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.c. 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.
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.IAP[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 (=nullist=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.IAP[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-C, 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, ASSELECTQ, 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
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 inter-
preted LISP code.

2. List of features implemented.
The following features of SCHEME are implemented:

variables
combinators
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
PROCE
ENCLOSE (The first argument to ENCLOSE must be
a lambda-expression)

FLUID
FLUIDBEGIN
FLUIDSETO
CATCH
STATIC

Any form whose CAR has the fexpr or fsubr property is
passed directly to LISP for evaluation. This is useful for
doing GEBINDEFS, 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 #%&$) or (SETQ #COMPILETRACE T).

Any atom with a non-null expr, subr, leexpr, 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 Y) 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
#ENVSSTACK
#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 cpcode is extracted from the ir and executed. This all happens in a function called INNERLCFE. The following instructions are implemented:

(PUSHI ccncst)
(PUSH ident)
(PUSH-ENV)
(POP-ENV)
(APPLY-EXCE lisp-fn)
(EVAL-EXPE lisp-form)
(APPLY number-cf-args)
(APPLY-CLOSURE lvars ccde)
(TEST ccde1 ccde2)
(STORE ident)
(SET-LABELS ids ((lvars . ccode)* ))
(GLOBAL-STORE id)
(PUSH-OBJECT flag (msg bvars . bcd)*)
(PUSH-ENV)
(POP-ENV)
(PUSH-FLUID id)
(FLUIDENV 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, à 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 @content)
(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 (LAMDA ()) X))
 (SET (LAMDA (V) (ASETQ X V))))

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

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

(DEFINE (RESETTABLE CELL X)
 (CLASS (CELL X)
 (RESET (LAMDA () (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)

Syntactic macros may be defined using the LISP DEFMAC R

(DEFMACR name tvar 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 tvar 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 (SPICE expr) produce segments which are
appended into the result. An example will clarify this.
If the value cf 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 `PRiEMPT`. For example, one may write:

```
(define PRiEMPT (X) (eLOCK (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 pnp [10353,1000]. To run it, type:

```
:*RU SCHEME[10353,1000]
*(READLCCF)
```

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

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

LISP 1.6[IUEUI MM-DD-YY]
```

```
*(CALTAL)
```

```
10
*(DSKIN (5C106 5003) (S3.LAP))
```

```
(AUG-2-79)
```

```
FINISHEI-ICALING
*(DECIMAL)
```

```
10
*(READLCCF)
```

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 PPW [50106,5003]. These include:

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

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

Report bugs, inaccuracies, and other problems to Mitch Ward 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.
This version features DO, ITERATE, CR, AND, and ANAPCAE

This version features interrupts
- See the code for RUSH for details.

This version features FLUIDSIO, FLUID, and FLUIDSETG

This version also features CATCH
- Also STATIC

(SETQ VERSION (QUOTE (AUG-2-79)))

(SETQ ITALIC T)
- Set input mode to cctal to please lap_.

(GCRTL)

Get CBRIDGE loaded, and set list of indicators

(GCRTL HBRIDGE)

(HCCOE (QUOTE (SCHEME-VALUE INSTR CCFILE SCHEME-MACRO SCHEME-SOURCE)))

Declare global variables for the compiler

(DEFPROP *STACK T SPECIAL)
(DEFPROP #IN T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #ENV T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)
(DEFPROP #ENVSTACK T SPECIAL)

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

(DEFPROP I T SPECIAL)

(IN BRIDGECE
NIL
(NHMG (IN STIR))
(PRINT (LIST (QUOTE SCHEME) (QUOTE VERSION)) VERSION)
TAG
(TERPRI)
(PRINT (QUOTE **))
(SETQ X (BUFFER (READ)))
(CHAR (ATOM X) (GO TAG))
(SETQ #ENV NIL)
(SETQ #ENV NIL)
(SETQ #ENVSTACK #(VAL-STACK-UNDERFLOW))
(SETQ #ENVSTACK #(.ENVSTACK-UNDERFLOW))
(SETQ #ENVSTACK #(.ENVSTACK-UNDERFLOW))
(DE GUARD-PRINT (X) (CONZ
  ((ATCH X) (PRINT X))
  ((BACK (CAR X) #CLOSURE OBJECT))
  ((PRINT (PRINT X))
  (T (PRINT X))
))

(DE ERRG2 (X) (ERRG2 (PRINT X) (ERR (PRINT SCHEME-ERROR))))

(SETQ #CSTACK #NIL)

(SETQ #ERABLE #NIL)

(DE PP (X) (ERRG2 (PRINT) (PRINT X 1 )
)

(SETQ #SCFAC #NIL)

| This is the inner loop. It performs an instruction fetch, advances the program counter (at the stack), and dispatches
| on the opcode.

(DE INNBFACCS () (PROG (OPCODE CDARC))
  (SETQ #B (CAR #CSTACK))
  (CDAR (PRINT #B))
  (SETQ CDARCS (CDAR #B))
  (SETQ CDARC (RETURN (CAR #VALSTACK)))
  (SETQ CDARC (CDARC #B))
  (T (SETQ #STACK (CDAR #CSTACK)))
  (SETQ #STACK (CDARC #B))
  (SETQ CDARC (SET CDARC #TIME))
  (SETQ #STACK (CDARC #B))
  (SETQ #STACK (CDARC #B))
  (SETQ #STACK (CDARC #B))
  (SETQ #STACK (CDARC #B)))
(CCCD ((REIIL CECDE) (REBCH (LIST SUNRECOGNIZE-INSTROCTION #II)))
  (APPLY (CAR CECDE) NIL)
  (GC FITCH)))

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

(DE RIGE
  (VARS ARGZ ENV)
  (CCCD ((EC (LENGTH VARS) (LENGTH ARGZ)) (CONS (CCBS VARS ARGZ) ENV))
    (I (ERROR (QUOTE WRONG-NO-OP-ARGZ--TIME))))

(DE VALUE
  (NAME ENV) (VALUE1 NAME (LCCUXP NAME ENV)))

(DE VALUE1
  (NAME SLCT)
  (CCCD ((EDIT SLCT)
    (CCCD ((EDITCP NAME) NAME)
      ((SETQ #GOT (GET NAME (QUOTE SCHEE-VALUE))) #GOT)
       ((SETQ #GOT (GETL NAME (QUOTE (SCEER--VALUE))) (CAR #GOT))
         (I (ERROR (QUOTE UBOUND-IDENTIFIER) NAME))))
     (T (CAR SLCT)))))

(DE LCCUXP
  (NAME ENV) (CCCD ((NULL ENV) (QUOTE NO-SLCT)) (T (LCCUXP NAME (CCAR ENV) (CCAR ENV) ENV))))

(DE LCCUXP)
  (NAME VARS VALS ENV)
  (CCCD ((NULL VARS) (LCCUXP NAME (LCC ENV))
    ((EC NAME (CAR VARS)) VALS)
     (T (LCCUXP NAME (CCAR VARS) (CCAR VALS) ENV))))

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

(DE ONE (X) (CAR X))

(DE TWO (X) (CADDR X))

(DE THREE (X) (CADDIER X))

(DE FOUR (X) (CALLCAR X))

(DE FIVE (X) (CADE (CCLDE X))

(DE FIRST* (X) (NAPCAR (QUOTE (LAMDA (U) (CAR U)))) X))

(DE SECOND* (X) (NAPCAR (QUOTE (LAMDA (U) (CADDR U)) X))

There are two SCHEME primitives.

(CLOSE-ENVIRONMENT)

(CLOSE-ENVIRONMENT (ENVIRONMENT))

(CLOSE-ENVIRONMENT (ENVIRONMENT))

The following functions implement MACLISP's "back quote" mechanism or Quine's quasi-quote.

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

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

(COMPILE (EXP) (COMPILE1 EXP NIL))

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

(COMPIL EXP T SPECIAL)

(COMPIL EXP T SPECIAL)

(COMPIL EXP T SPECIAL)

(COMPIL EXP T SPECIAL)

The following instructions, the listing is arranged by
language feature: for each type of SCHEME phase,
we list the code for compiling it, followed by the
code for executing the relevant machine instruction.

This is the compiler. It translates from SCHEME code
If in non-nil only if the current program is to
be followed by additional instructions.

(CCELIB (EXP PRES) (CCONS
 (FULL EXE) (LIST (CCONS NIL)))
 (CONS (EXP HT) (LIST (CCONS (HULL T))))
 (ITAL EXE) (CENT
 (CONS (EXP) (LIST (CENT 2PUSHI EXP)))
 (T (LIST (CENT 2PUSHI EXP))))
 (GETI (CARS EXP) (EXP) (DEFER LIST) (LOAD USER))
 (APPEND (CCELIB (CEN EXE) PRES)
 (LIST (LIST 3PUSHI-EXP)
 (CNR EXP) (LENGTH (CEN EXP))))
 (LIST (CNR EXP) (GET (CNR EXP) (MACRO))
 (CCELIB (AILI TEMP (LIST EXE)) PRES))
 (EXEC TEM (GET (CNR EXP) (CCONS TEMPS))
 (EVAL TEM))
 (EXEC TEM (GET (CNR EXP) (SCHEME-MACRO))
 (CCELIB (APPLY TEM (LIST EXE)) PRES))
 (GETL (CARS EXP) (FLUSH PSUSE))
 (LIST (LIST 3PUSHI-EXP))
 (T (APPEND
 (CCELIB (EXP PRES)
 (MK-APPLY PRES (LENGTH (CNR EXP)))))))

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

(DEPUSHI PUSI (PUSI INSTB)
 (DE PUSI) (SETQ #VALSTACK (CCONS (CARS #1S) #1W #VALSTACK)))

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

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

The user must build his own interrupt
processor; the installer one is
((DISABLE PREEMPT [X] X)
which just causes the computation to proceed.

Note that #ENABLED is initially set to nil,
so if you don't want to play interrupt games,
(DEFMFRG PUSH (ENV) INSTR)

(DE PUSH()) (CCBG
 ((ARE #ENAELE (GREATERP (DIFFERENCE (TINQ) #ALARM CLOCK))
   50))
  (SETQ #VALSTACK (CONS (VALUE (CAR #ENV) #ENV))
   (CONS (VALUE #PREEEDP #ENV)
      #VALSTACK))
  (SETQ #CSTACK (CONS 
  B ((PUSH-ENV) (APPLY 1) (EXEC-ENV))
  #CSTACK))
  (SETQ #ALARM_CLOCK (TINQ))
  (X (SETQ #VALSTACK (CONS (VALUE (CAR #ENV) #ENV)
      #VALSTACK))))

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

(DEFMFRG PUSH-ENV (PUSH-ENV) INSTR)

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

; The format for a pop-env is (EXEC-ENV).

(DEFMFRG EXEC-ENV (POP-ENV) INSTR)

(DE POP-ENV NIL (EXEC2
  (SETQ #ENV (CAR #ENVSTACK))
  (SETQ #ENVSTACK (CONS #ENV #ENVSTACK))))

; The format for the APPLY-EXOP instruction is
; (APPLY-EXOP fn number-of-args).
; fn must be a primop (an opext, unbr, leqbr, lieqbr, or macro).
; When this instruction is executed, the top of
; the stack should look like args, argn-1, ..., arg1.
; There n arguments are removed from the stack
; (the valstack), and (fn arg1 ... argn) is pushed onto it.

(DEFMFRG APPLY-EXOP (APPLY-EXOP) INSTR)

(DE APPLY-EXOP NIL (EXEC (ARGS)
  (SETQ ARGS (REMOVE-FROM-VALSTACK (CALDB #ENV)))
  (SETQ #VALSTACK (CONS (APPLY (CAR #ENV) #ARGS) #VALSTACK))))

; REMOVE-FROM-VALSTACK removes n items from the top of
; the valstack and reverses them, leaving the
; last n items on the top of the stack (a la Reverse).

(DE REMOVE-FROM-VALSTACK (N) (EXEC (N NIL NIL TEMP)
  (SETQ X #VALSTACK))
(SETQ XTRAIL NIL)
(CODE ((LFORM 3) (SETQ #VALSTACK X) (RETURN XTRAIL)))
(SETQ TEMPE (CEIL X))
(REPLACE 1 XTRAIL)
(SETQ XTRAIL X)
(SETQ X TEMPE)
(SETQ X (SUP N))
(SETQ TCF))

The format of the EVAL-FIXREF instruction is

(VVAL-FIXREF form)

form is evaluated and pushed onto the valstack.

(DEPERSF-EVAL-FIXREF (EVAL-FIXREF) INSTB)

(EV EVAL-FIXREF NIL (SETQ #VALSTACK (CONS (EVAL (CALL #IF) #VALSTACK))))

RE-APPLY is the function which finally exits a

PUSH-ENV--EVAL-ENV pair when necessary.

(RE-APPLY (EVAL N) (CCMC
(EFS! (LIST 1 (PUSH-ENV) (LIST RE-APPLY N) 2 (EVAL-ENV)))
(T (LIST (LIST RE-APPLY 3)))))

COMPLES compiles code which causes the expressions
in exp to be evaluated in sequence, left-to-right.

If exp is nil, then the last expression need not
preserve the environment; this is allowed-tail-recursion.

All the other expressions need to preserve
the environment no matter what.

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

(CONS (EXPS FRM) (LAPEPEN (LAPEPEN
*FUNCTION (EVALPLICATE Z)
(CODE
((AND (NULL (CDR Z)) (NULL FRMS))
 (CCMC
 (CONS (CAR 2) NIL)
 (T (CONSPILE1 (CAR 2) T)))))
FRMS)))

(LAPEPEN (X) (CSCD
((NULL Z) NIL)
(T (LAPEPEN (CAR X) (LAPEPEN (CEIL X))))))

The format for an APPLY instruction is

(REP L)
where \( n \) is the number of arguments.

When this instruction is executed, the top of the valstack should look like:

\[ \text{arg}, \text{arg}_n, \ldots, \text{arg}_2, \ldots, \text{arg}_1, \text{fn} \]

where \( \text{fn} \) is any functional object.

These \( n-1 \) items are removed from the valstack, and \( \text{fn} \) is pushed onto it.

```
(DEFFPROG APPLY (EXECUTE-APPLY) INSTR)

(CE EXECUTE-APPLY #\{FREG (FREG ARG5)\}
  (SETQ ARG5 (ERROR-PROC-VALSTACK (CADDR #IR)))
  (SETQ F5 (CAR #VALSTACK))
  (SETQ #VALSTACK (CDR #VALSTACK))
  (CCDF)
  (LATCH F5) (SETQ #VALSTACK (CONS (FUNCALL F5 ARG5) #VALSTACK)))
  (EQ (CAR F5) ECLOSURE)
  (SETQ #CSTACK (CONS (THREE F5) #CSTACK))
  (SETQ #ENV (MAKE-TWO F5 ARG5 (PADDR F5)))
  (EQ (CAR F5) ECLOSURE) (EXECUTE-OBJECT (CADDR F5) (CAR ARG5)
    (CDR ARG5)))
  (EQ (CAR F5) ECLOSURE) (EXECUTE-OBJECT (CADDR F5) (CAR ARG5)
    (CDR ARG5)))
  (EQ (CAR F5) ECLOSURE) (EXECUTE-OBJECT (CADDR F5) (CAR ARG5)
    (CDR ARG5)))
)

(CE PRINC #F) (CLET #F (QUOTE (EXPR LEXPR SEXPR MACRO)))

(CE FUNCALL
  (FREG ARG5))
  (CONS (CLET #F (CQUOTE (EXPR LEXPR SEXPR MACRO))) (APPLY F5 ARG5))
  (T (FUNCALL #F ARG5)))

FICAL is used to call macros which have been bound.

It changes each element of args by putting a "quote" in front. It does this causing the cons cells from which args is assembled. qi is a statically allocated list of "QUOTE NIL"'s which are used.

For this quoting, when the arguments are all quoted, the function is called on (again using the statically allocated cons-call box to save on cons-ing), and the resulting list is EVALed. After evaluation, the statically allocated items are nilled out.

```
(CE FUNCALL #F ARG5) (EROG (CL FCL BOX X QQL)
  (SETQ CL #\{(QUOTE NIL) (QUOTE NIL) (QUOTE NIL)\})
  (SETQ FCL CL)
  (Y PQL points to the list of unused quote-calls.
  (SETQ BOX #\{NIL\})
  (SETQ X ARG5)
  (Y X walks down ARG5).

ECDF (CCDF ((NULL X) (GO TAG)))
  (NILAC (SCAR FCL) (SCAR X))
  (NILAC X (CAR PQL)))
(SETQ X (CBS NIL))
(CCBS (BULL (CBS PCL)) (BPLACE PCL (CBS 3QUOTE NIL)))
; if PCL has no place to go, extend QL
(SETQ PCL (CBS PCL))
(GC LCPF))

TAG
(BPLACE CBS PB)
(BPLACE CBS ARGS)
(SETQ X (XVAL CBS))
(BPLACE (BPLACE CBS NIL) NIL)
(SETQ CCL CL)
; QCL walks down CL until it hits PCL
LCPF2
(CCBS ((EC CCL PCL) (RETURN T)))
(BPLACE (CLASH CCL) NIL)
(SETQ CCL (CBS CCL))
(GC LCPF2))

I'm an OBJECT (the result of evaluating a
CLASS expression) is represented as follows:

I OBJECT - cllist

I cllist ::= (rcvr *)

I rcvr ::= ((msg bvars . code) . env)

(DEF EXECUTE-OBJECT (CLIST MSG ARGS) (CBS ()))
TCP
tCBS
+ (CL CBS (CBS CLIST))
(SETQ #STACK (CBS (CBS CLIST) #STACK))
(SETQ #ENV (CBS (CBS CLIST) CLIST) (CBS CLIST))
(RETURN T))
(SETQ CLIST (CBS CLIST))
(GC TCP))

I LABEL clist body) compiles into
I (SHUSH-CLOSURE bvars code), where code is
I the object code for body. This instruction
I pushes
I (CLGSH bvars code environment)
I onto the valistack.

(EXITPROP LABEL (CCBS-LABEL) COMPILE)
(DEF CCBS-LABEL () (LIST LIST SHUSH-CLOSURE
(CCBS EXP)
(COMPILE1 (CBS EXP) NIL)))

(EXITPROP PUSH-CLOSURE (PUSH-CLOSURE CLIST))
(DEPUSH-CLOSURE () (SETQ #VALSTACK (CCWS
 (LIST #CLOSURE (CAER #1R) (CAER #1R) #ENV)
 #VALSTACK)))

(DEPUSH IF (CCSF-IF) COMPILE)

(DE CCSE-IF NIL (APPEND
 (CCSFIF1 (CAER EXP) T)
 (LIST (LIST 8TEST
 (CCSFIF1 (CAER EXP) PNE))
 (CCSFIF1 (CAER EXP) PNE))))

1Y The format for a TEST instruction is
1Y (TEST code1 code2). Either code1 or
1Y code2 is executed, depending on the
1Y top of the valstack. The boolean is
1Y consumed.

(EFFPSC TEST (INTERPRET-TEST) INST)

(DE INTERPRET-TEST NIL (PROC2
 (CCHE ((CAER #VALSTACK) (SETQ #CSTACK (CCHE (CAER #1R) #CSTACK)))
 (T (SETQ #CSTACK (CCHE (CAER #1R) #CSTACK)))))
 (SETQ #VALSTACK (CAER #VALSTACK) ))

(EFFPSC &GTCE (CCSF-CGTE) COMPILE)

(DE CCSE-CGTE NIL (LIST (LIST BRUSHI (CAER EXP))))

(EFFPSC &ASFC (CCSF-ASFCT) COMPILE)

(DE CCSE-ASFCT ()
 (APPEND
 (CCSFIF1 (CAER EXP) T)
 (LIST (LIST 8STCSE (CAER EXP)) ))

1Y The format is (STCSE identifier)

(EFFPSC STCSE (ST-STORE) INST)

(DE ST-STORE ()
 (PROC (SITQ)
 (SITQ SLOT (LOOKUP (CAER #1R) #ENV))
 (CCON ((AHEX SLOT) (EFFPSC (CAER #1R) (CAER #VALSTACK) #SCHERE-VALUE))
 (T (BELAT SLOT (CAER #VALSTACK) )) )

(EFFPSC LABELS (CCSF-LABELS) COMPILE)

(DE CCSE-LABELS ()
 (APPEND
 (CCON (PNE (1PUSH-ENV)) (T NIL))
 (LIST (LIST BSIT-LABELS (FIRST* (CAER EXP)))))

1Y
(CCB-LABELS (SECOND* (CASCADE EXP))
(CCBPOOL (CASCADE EXP) #IL)
(CCB (EXP @ (((POP-EXP))) (T NIL)))
)

(DEF CCB-LABELS (LEIFS) (CCB
 (FULL LEIFS) #IL)
 (T (CCB
 (CCBS (CASCADE LEIFS) (CCBPOOL (CASCADE [LEIFS]) #IL)
 (CCB-LABELS (CCB LEIFS)))))))

;Y The format is (SET-LABELS id ((bvars . code)*))
(DEFPROP SET-LABELS (CC-SET-LABELS) INSTR)

(DE DO-SET-LABELS () (PRGC2
 (SETQ #ENV (CCBS (CCBS (CASCADE #IS) #IL) #ENV))
 (REPLACE (CAR #ENV) (CLOSE* (CASCADE #TB))) ))

(DE CLOSE* (LEIFS) (CCB
 (FULL LEIFS) #IL)
 (T (CCB
 (LIST BCLOSURE (CASCADE [LEIFS]) (CASCADE EXP) #ENV)
 (CLOSE* (CASCADE EXP)))))))

(DEFPROP *DEFINE (CCB-DEFINE) CCPOOL)

(DE CCB-DEFINE () (AFFEBE
 (CCBPOOL (CASCADE EXP) #EXP)
 (LIST (LIST BCGLOBAL-STORE (CASCADE EXP)))))

;Y The format is (GCCELL-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 GCCELL-STORE (GLOBAL-STORE) INSTR)

(DE GCCELL-STORE #IL (PRGC3
 (EUTBEC (CASCADE #EB) (CAR #VALSTACK) BCSCHEME-VALUE)
 (SETQ #VALSTACK (CCBS (CASCADE #IS) (CASCADE #TB) #VALSTACK)))))))

;Y The syntax for CLASS is
;Y (CLASS basis . (msg lambda-exp)*)

;Y An example is
;Y (CLASS BAR (FOO (LAMBDA (X Y) ...) (BAR (LABELA ()) ...)))
A CLASS expression is like a LAMBDA expression in that
it evaluates to an OBJECT (like a closure). The object
(sorry about that pun) code is

(PUSH-BJECT flag. (msg locals . bcd) * )

(CBREF-CLASS [COMP-CLASS] COMPIL)

(DEF CCREF-CLASS () (AFFINE
    [CCREF
        (CAEB EOB) (COMPILE1 (CAEB EOB) T))
        IF the basis is non-nil, get it on the stack
        (T NIL)
    ]
    (LIST
        AFFINE [LIST DEUSH-JECT (NULL (NULL (CAEB EOB)))
            (CCREF-CLASS1 (CCREF EOB)) ] ) )

(DEF CCREF-CLASS1 (LC) (IXEAPCCL @CCREF-CLASS2 L))

(CCCB
    ((NULL L) NIL)
    (T (CCCS
        (CCREF-CLASS2 (CAR L))
        (CCREF-CLASS1 (CDR L)) ) ) )

(DEF CCREF-CLASS2 (X) (CCCS
    (CAR X)
    (CCCS (CALCER X)
        (CCPFEETE1 (CADER (CAR X)) NIL) ) )

(IZEPF PUSE-JECT (EXECUTE-PUSH-JECT) INSB)

(EXE CCREF-JECT () (PROG (C ENV)
    (SETQ ENV (CCCS (CCS (SELF ENV) NIL) #ENV))
    (SETQ C (CLOSE-CLIST (CAR #IR) (CDR #IR) ENV))
    (SETQ #VALSTACK (CONS (CONS #OBJECT C) #VALSTACK))
    (SELACD (CAR ENV) (LIST (CAR #VALSTACK))) )

(DEF CLOSE-CLIST (FLAG CL ENV) (CCCS
    (NULL CL)
    (CCCS
        (FLAG (PROG2 NIL (CCAR #VALSTACK) (SETQ #VALSTACK (CDR #VALSTACK)))
            (T NIL))
        (T (CCCS
            (CONS (CAR CL ENV)
                (CLOSE-CLIST FLAG (CDR CL ENV)) ) ) )

IY Here is the stuff for fluid variables

IY The format for PUSH-FLUIDS is (PUSH-FLUID IDENTIFIER)
(DEFFLUSH-FLUSH (FLUSH-FLUSH) INSTR)

(DE FLUSH-FLUSH ()
  (SETQ #VALSTACK (CONS
    (VALUES (CAAR #IN) #PENV)
    #VALSTACK))))

| The format is (FLUSH-FLUSH) and (FLUSH-FLUSH) |

(DEFFLUSH-FLUSH-FLUSH (FLUSH-FLUSH) INSTR)

(DE FLUSH-FLUSH () (SETQ #ENVSTACK (CONS #PENV #ENVSTACK)))

(DEFLUSH-FLUSH-FLUSH (FLUSH-FLUSH) INSTR)

(DE FCX-FLUSH () (FORC2
  (SETQ #ENV (CAR #ENVSTACK))
  (SETQ #ENVSTACK (CDR #ENVSTACK))))

(DEFLUSH-FLUSH-FLUSH (CCFL-FLUSH-FLUSH) COMPILE)

(DE CCFL-FLUSH-FLUSH () (LIST
  (VALUES #ENVSTACK (CAAR EXP))))

(DEFLUSH-FLUSH-FLUSH (CCFL-FLUSH-FLUSH) COMPILE)

(DE CCFL-FLUSH-FLUSH () (CCFL-FLUSH-FLUSH)
  (FIRST* (CAAR EXP))
  (SECOND* (CAAR EXP))
  (CICLUS EXP)))

(DE CCFL-FLUSH-FLUSH (VALUES VARS DO NOT) (APPEND
  (CCFL-FLUSH-FLUSH) (VALUES DO NOT)
  (VALUES #ENVSTACK (VALUES #ENVSTACK))
  (VALUES (VALUES #ENVSTACK) (VALUES #ENVSTACK))
  (VALUES (VALUES #ENVSTACK) (VALUES #ENVSTACK))
  (VALUES (VALUES #ENVSTACK) (VALUES #ENVSTACK))
  (VALUES (VALUES #ENVSTACK) (VALUES #ENVSTACK)))

| The fluid/indirect instruction has format |

| FLUSH-FLUSH a value code |

(DEFFLUSH-FLUSH-FLUSH (EXECUTE-FLUSH-FLUSH) INSTR)

(DE EXECUTE-FLUSH-FLUSH () (FORC NIL
  (SETQ #ENV (KIND (CADDR #IN))
    (REMOVE-FLUSH-VALSTACK (CAAR #IN))
    #PENV))
  (SETQ #CSTACK (CONS (CAAR #IN) #CSTACK))
)

(DEFLUSH-FLUSH-FLUSH (CCFL-FLUSH-FLUSH) COMPILE)

(DE CCFL-FLUSH-FLUSH ()
  (APPEND
  (VALUES #ENVSTACK))
(DEFLISTSTORE (EX-FLUIDSTORE) INSTR)
(DE EX-FLUIDSTORE () (ERCG (SLCT)
  (SITC SLOT (LOOKUP (CRED #IB) #PENV))
  (CRED)
  (RETRD CRED #IB) (CRED #VALSTACK) #SCHNE-VALUE))
  (T (REPLAC SLCT (CRED #VALSTACK)) ) )))

<table>
<thead>
<tr>
<th>This is the end of the fluid variable stuff</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Here is the code for CATCH</th>
</tr>
</thead>
</table>

(DDEFPRO CATCH (CCSP-CATCH) CORFILE)

(DE CCESS-CATCH () (APPHNC
  (CCED (PRES #((PUSH-ENV)) (T NIL))
  (LIST (LIST SCATCH (LIST (CRED EXP))
  (CCSPFILE1 (CRED EXP) PRES))
  (CRED (PRES #((POP-ENV)) (T NIL))))))

<table>
<thead>
<tr>
<th>The format is (CATCH (id) code)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>De CCESS-CATCH () (EX-CATCH) INSTR</th>
</tr>
</thead>
</table>

(DDEFPRO CATCH (EX-CATCH) INSTP)

(DE EX-CATCH () (ERCG NIL
  (SITC #ENV (BIND (CRED #IB)
  (LIST (COLLECT-CONTINUATION))
  (T BIND takes lists of args and vals
  #ENV))
  (SITC #CSTACK (CADDR #IR) #CSTACK)) ) )

| (DE COLLECT-CONTINUATION ()
|--------------------------------|

| (LIST &CONSMULTIARG #ENV &EMV #ENVSTACK #ENVSTACK #VALSTACK
| #CSTACK) ) |

| (DE START-CONTINUATION (FN ARGS) (ERCG NIL
|-----------------------------------------------|

| (SITC #ENV (CADDR FN))
| (SITC #EMV (CADDR FN))
| (SITC FN (CADDR FN))
| (SITC #ENVSTACK (CADDR FN))
| (SITC #ENVSTACK (CADDR FN))
| (SITC #VALSTACK (REPLACE VARV (CADDR FN)))
| (SITC #CSTACK (CADDR FN)) )))

| This is the end of the CATCH stuff |

| De is used to define syntactic macros (magic words). |
(DEF DSN (X) (PROC
  (OUTER (C A X)
    (LIST [HEADA (LIST (CAAR X)) (CADDR X))
      [SCEMIP-SACHE] (CAR X))
  )
)

(ESM SELL 2
  (CCSE ((NULL (CDDR 2)) NIL)
    (NULL (CDDR 2)) (CDDR 2))
  T (BUILD (LAMBDA (A B) (B)) (VAL (CDDR 2)) (LAMBDA NIL (SLLCK (SPICE (CDDR 2)))))
)

(ESM LET 2 (BUILD (LAMBDA (VAL (FIRST* (CADDR 2))) (BLOCK (SPICE (CDDR 2)))) (SPICE (SECOND* (CADDR 2))))
)

(ESM TEST 2
  (BUILD (LAMBDA (F P A) (IF P ((F) P) (A)))
    (VAL (CADDR 2))
    (LAMBDA NIL (VAL (CADDR 2)))
    (LAMBDA NIL (VAL (CADDR 2)))
  )
)

(ESM CCED 2
  (CCED ((NULL (CDDR 2)) (QUOTE NIL))
    (NULL (CDDR 2)) (CDDR 2)
    (BLOCK (LAMBDA (V B) (IF V V (B))) (VAL (CAR (CDDR 2)) (LAMBDA NIL (COND (SPICE (CDDR 2)))))
    (SPICE (CADDR 2)) (QUOTE ->)
    (BLOCK TEST (VAL (CADDR 2)) (VAL (CADDR (CADDR 2)) (CCED (SPICE (CDDR 2))))
    (SPICE (CADDR 2))
    (LAMBDA NIL (VAL (CADDR 2)))
  )
)

(ESM LIST 2 (COND ((NULL (CDDR 2)) (QUOTE NIL)) (T (BUILD CCEDS (VAL (CADDR 2)) (LIST (SPICE (CDDR 2)))))))

(ESM STATIC 2 (CADDR 2))

(ESM DEFINE 2 (CONE
  (AND (EQ (LENGTH 2) 3) (ATCH (CADDR 2)))
  (BUILD BLOCK
    (REFERENCE (VAL (CADDR 2)) (VAL (CADDR 2)) SCSHERE-SOURCE)
    (REFERENCE (VAL (CADDR 2)) (VAL (CADDR 2)))
  )
  (ATCH (CADDR 2))
  (BUILD DEFINE (VAL (CADDR 2)) (LAMBDA (VAL (CADDR 2)) (SPICE (CADDR 2))))
  (T (BUILD DEFINE (VAL (CADDR 2)) (VAL (CADDR 2)) (SPICE (CADDR 2))))
)

(ESM DO 2 (CC 1 (BUILD-CO-TAPE (CADDR 2))
  (FIRST* (CADDR 2))
  (CADDR 2)
  (CADDR 2)
  (CADDR 2))
)

(BE BUILD-CO-TAPE (PARAMS) (MAPCAR
(*FUNCTION (LAMBDA (X)  
  (CONS (GENSYM) (CONS (GENSYM) (CONS (GENSYM) X)))))

(DEFFUN TABLE T SPECIAL)  
(DEFFUN VARS T SPECIAL)  
(DEFFUN TEST T SPECIAL)  
(DEFFUN CHERE T SPECIAL)  
(DEFFUN ECITY T SPECIAL)

(DE DC7 (TABLE VARS TEST DOWE BODY)  
  (BUILD LET ((TS (LAMBDA (VAL VARS) (VAL TEST)))  
    (EN (LAMBDA (VAL VARS) (BLOCK (SPICE DOWE))))  
    (BE (LAMBDA (VAL VARS) (BLOCK (SPICE BODY))))  
    (SELICE (BFCAR  
      (FUNCTION (LAMBDA (E)  
        (BUILD (VAL (CAR E)) (LAMBA ())  
          (VAL (CAED (CEED E)) (C)))))  
    ))  
  ))  
(DELC (LAMBA (E)  
  (BUILD (VAL (CAED E)) (LAMBA (VAL VARS)  
    (VAL (CAED (CEED E))) (C)))))

(LICEP  
  (IF (TS (SELICE (THIRD T TABLE)))  
    (EN (SELICE (THIRD T TABLE)))  
    (BLOCK  
      (BD (SELICE (THIRD T TABLE))))  
  (LCGC  
    (SELICE (BFCAR  
      (FUNCTION (LAMBDA (E) (CONS (CAR E) NIL))  
        (TABLE))))))

(IN FORMAT FOR TABLE WERT) AS (E1 E1 E1 V1 V1 V1)

(DE THIRD* (L) (CONS ((NULL L) NIL)  
  (L) (CONS (CAEDB L) (THIRD* (CEED L)))))

(DSM ITERATE 2 (BUILD LABELS  
  (((VAL (CAED Z)) (LAMBA (VAL (FIRST* (CAEDB Z))))  
    (BLOCK (SPICE (CEED Z))) (THIRD* (CAEDB Z)))))
  (((VAL (CAED Z)) (SPICE (SECOND* (CAEDB Z)))))

(ESH GR 2 (CCCE (NULL (CEED Z)) NIL)  
  (NULL (CEED Z)) (CEED Z)  
  (B (BUILD CORE (NULL (CAED Z))) (IF (FOR (SPICE  
    (CEED Z)))))))