This document was produced by SDC and III in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency Information Processing Techniques Office & Subcontract 65-107.

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 200 Sixth Street / Cambridge, Massachusetts 02142

TM- 2710/320/01

AUTHOR
R. A. Saunders, III
J. A. Barnett, SDC
Donna Firth, SDC

TECHNICAL
L. Hawkinson, III
C. Weissman, SDC

RELEASE
S. L. Kameny, SDC
E. Fredkin, III

for J. I. Schwartz, SDC

DATE 1 February 1966

The LISP 2 Compiler

ABSTRACT

This document describes the operation of the LISP 2 compiler, as presently coded in LISP 1.5. The compiler translates LISP 2 intermediate language into LAP 2 input.

TABLE OF CONTENTS

Figure List

THE LISP 2 COMPILER

1. GENERAL

The LISP 2 compiler translates the S-expressions of LISP 2 Intermediate Language into the assembly language input of LAP 2. It is composed of three principal parts: the control flow and variable binding supervisor, herein called the controller; the arithmetic function translator; and the register allocater, called the mover. These parts, which are largely independent, interact in a fairly simple way.

The notion of multiple bindings of a variable at different levels of recursion is extremely important in this compiler. Numerous free variables describe the state of the compilation at any given time, and these variables are bound and set in a complex manner.

The reserved words of the LISP 2 Intermediate Language are treated by invoking a specially coded function for each such word. The functions accessed in this way are called INSTRUCTIONS. These functions interact strongly with the free variables and other functions of the compiler. This arrangement permits great flexibility in changing the language. Adding a new reserved word can be done quite quickly by anyone familiar with the compiler, and with relatively little danger of damaging existing functions.

2. THE CONTROLLER

The controller is the top level of the compiler. It operates recursively to dissect the various levels of declarations, function compositions, and so on. Its general flow is shown in Figure 1.

At the top are routines invoked by appearance of the reserved words (BLOCK, GO, etc.). These words call drivers which cause forms to be compiled in the appropriate context; the drivers in turn call COMEXP, which is sort of a master switch through which recursion takes place.

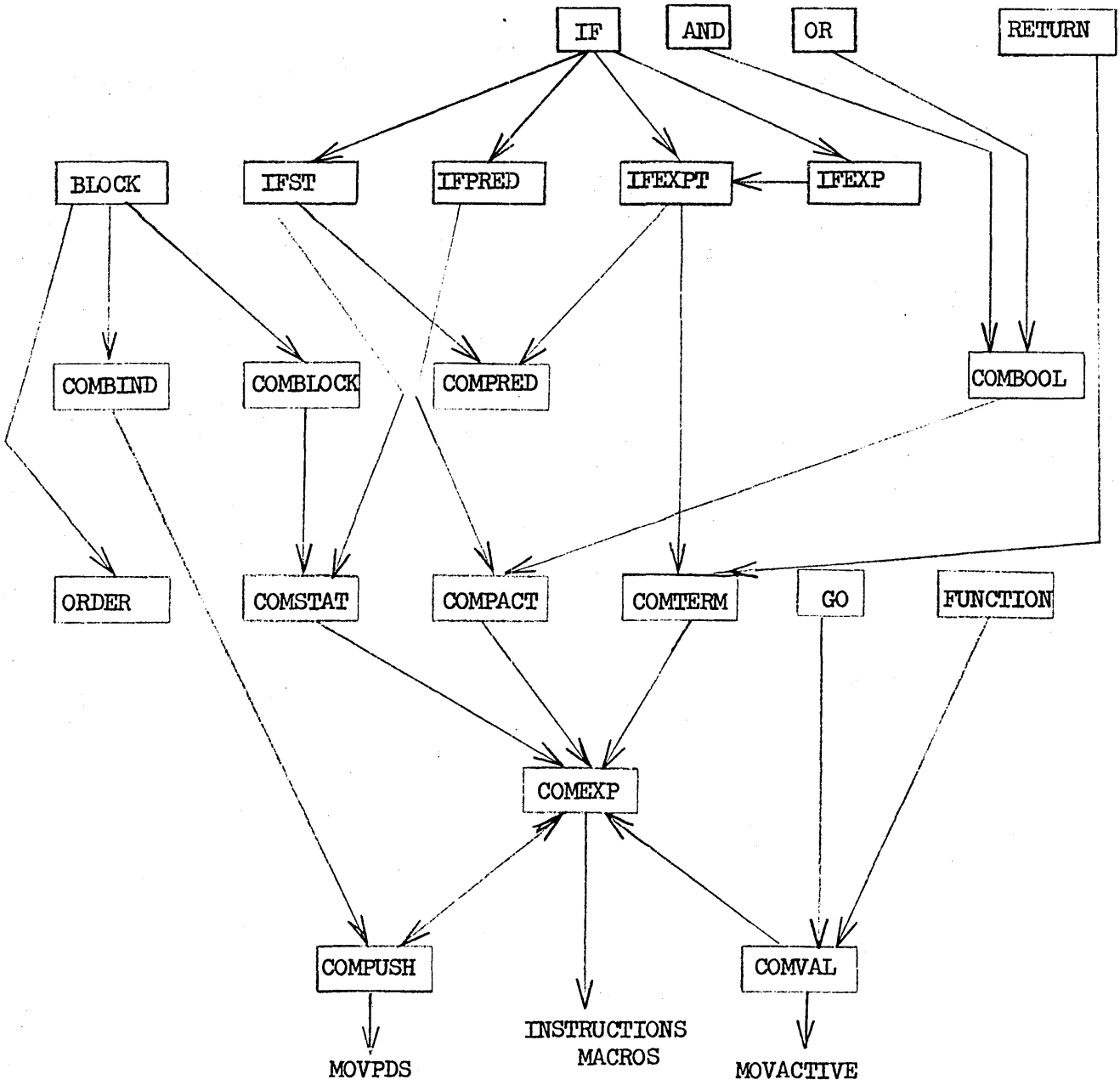

Figure 1. Controller Flow

The context is described by three fluid state variables, SCLASS, PCLASS, and TERGO. If the compilation of an expression is to produce a value, then the context is referred to as expression and SCLASS is NIL. If the compilation is of a statement in a block, where no value is required, then SCLASS is TRUE. PCLASS describes the result required at the expression level. If the result is to be used as a predicate, then PCLASS is TRUE; otherwise it is NIL. TERGO carries the label for the point at which IF branches in expression context reconverge. It is either a label, or (more usually) NIL. As the S-expression is dissected, SCLASS, PCLASS and TERGO are rebound as the context changes. All four combinations of SCLASS and PCLASS are possible because whenever the context is statement, there is always a higher level in which it is expression, and the nature of this expression controls PCLASS independently of SCLASS.

2.1 COMEXP

COMEXP has an explicit argument EXP, which is either a variable or a list of a function name (possibly reserved) and arguments. The result of COMEXP is to set an assortment of state variables describing where the results of the computation may be found. These variables, called the V-variables, include the following:

- VCLASS — Either LOC, meaning result in memory; ACTIVE, meaning in an active register; or DATUM, meaning a quoted item. May also be set to PREDICATE, although not by COMEXP.
- VTYPE — SYMBOL, REAL, INTEGER, OCTAL, BOOLEAN, or FORMAL.
- VREG — If VCLASS is ACTIVE, the register containing the value. If VCLASS is LOC, index modifiers required to access the value.

NAME	VALUE		
VCLASS	LOC (in core storage)	ACTIVE	DATUM (a "quoted" expression)
VTYPE	T1	T1	T1
VREG	THE INDEX MODE. OF VADDR, OR THE LIST OF DIR. & INDIR. INDEX	THE REGISTER HOLDING THE VALUE. (AC,B,L,1,...,8)	D.N.A.
VADDR	LAP Address	D.N.A.	THE ACTUAL S-EXPRESSION
VIND	NIL ≡ VADDR direct TRUE ≡ VADDR indirect	D.N.A.	D.N.A.
VBYTE	B1	B1	D.N.A.
VBLØT	C1	C1	D.N.A.
VINV	A1	A1	A1

D.N.A. ≡ Does not apply.
T1 ≡ Symbol, real formal, integer, octal, or boolean.
B1 ≡ Position within the word addressed.
NIL = full word
LH, RH, LA, RA, etc. are the same as the LAP modifiers (Reference III Memo No. 10, dated 23 August 1965. LAP II, author Michael Levin)
(i j) ≡ i is the right most bit, j is the number of bits.
C1 ≡ a list of the active registers destroyed in producing this result: AC, B, 1,...,8, etc.
A1 ≡ Special consideration in handling. MINUS, RECIP

These variables are bound by a driving function (usually to NIL) before calling COMEXP to compile an expression. COMEXP updates them appropriately to describe the final status of the compiled expression. In addition, if any moving function (MOVACTIVE, MOVELOC, MAKELOC) is utilized and changes the status of the expression then the variables will also be changed to reflect this.

Figure 2. Some Fluid State Variables (V-variables) for the LISP 2 Compiler

VADDR If VCLASS is LOC, the LAP address of the result.
If VCLASS is DATUM, the actual quoted item.

VIND If VCLASS is LOC, denotes that VADDR is taken indirect.

VBYTE If VCLASS if LOC or ACTIVE, the bit positions in the word. NIL denotes entire word.

VINV Inverted status. Result found is negative, reciprocal, or other function of desired result.

COMEXP first separates the variable case from the function-and-arguments case. Given a variable, its declaration determines the setting of the V-variables. For the function case, the declaration is first found. It may be a macro expansion, in which case COMEXP is applied to the expanded form. It may be declared as INSTRUCTIONS, in which case the associated program is applied. If it is a CAR-CDR chain, the appropriate CARs and CDRs are generated by MAKECARCDR. It may be an array, in which case code to calculate the subscripting is compiled by COMSUB. Or it may be FORMAL, which causes the arguments to be compiled and the appropriate calling sequence to be generated. The V-variables are then set to reflect the returned value of the function.

2.1.1 COMVAL and COMPUSH

COMVAL and COMPUSH are used to compile an expression so that the result appears in a specified place and is of a specified type. If no suitable type is specified, an appropriate one is established. They bind SCLASS, PCLASS, TERGO and the V-variables to NIL, then call COMEXP. The result is moved to the indicated place.

2.2 FUNCTION

FUNCTION is the topmost part of the compiler. It arranges the binding of arguments of the function, determines types, and compiles the expression for

the function's value. Because the context of the function's value is expression, SCLASS, PCLASS, and TERGO are bound to NIL. The following are also bound:

ALIST — A list of bound variables and their declarations, bound here to the function variables (via COMBIND).

LISTING — A list, in reverse order, of the LAP instructions generated for the compiled function.

REMOTES — A list, in reverse order, of remote code LAP instructions for the compiled function, such as SWITCH arrays.

REFLIST — A list of all identifiers used free. Used as an argument of LAP.

FNAME — The name of this function.

FTYPE — The type of the value of this function.

XTYPE — The type that the compiler is requested to produce for the value of this function.

VBLOT — Described in Section 2.1

The listing thus generated is reversed, attached to the reversed remotes, and passed off to LAP 2 to be assembled.

2.3 Blocks

There are two sorts of blocks: block-statements and block-expressions. They may be distinguished by the context in which they occur. They differ in their effects--RETURN, for example, which returns to the next higher expression level, goes one level in block-expressions but goes two or more in block-statements.

When BLOCK is recognized by COMEXP, the INSTRUCTIONS for BLOCK are invoked. A block-statement binding no variables is a compound statement and causes only compilation of the contained statements, via COMBLOCK. Otherwise more elaborate treatment is required. The ALIST is rebound to its previous value; COMBIND is then applied to the block variables to set the variables onto the ALIST.

There are now several possibilities: If SCLASS is TRUE, it suffices to apply COMBLOCK to the statements. If PCLASS is TRUE, the context is predicate. The result required in this case is a transfer of control to one place or another depending on whether the block evaluates to TRUE or FALSE. These places are marked by labels in the state variables TGO and FGO, respectively. Of TGO or FGO at least one must carry a label at all times; one of them may be NIL, indicating a fall-through case. If a fall-through case exists, a label is generated for it, and set into the appropriate variable. COMBLOCK then compiles the statements. If a fall-through label has been generated, it is now attached to the listing. VCLASS is now set to PREDICATE to tell the higher-level functions that all transfers required have been generated. TGO is set to NIL to avoid a spurious transfer at the end. Note that if no fall-through label is used, the above is unnecessary, as a higher routine will do both of these.

The remaining cases are expression cases. Here the concept of terminals becomes important. Consider the case:

(SET I (BLOCK (J)... (RETURN 3.0)))

where I is an integer variable. Conversion is required of the real datum to set the integer variable. This is done by collecting all RETURNS on a list of terminal points called TERMINS with the V-variables for the returned values and pointers to their places in the assembled code. Eventually COMTERMINS is called to search the TERMINS list and insert conversion codes into the assembled code as required. The RETURNS

merge at a label in TERGO. TERGO is first checked to see if it is NIL. This indicates that an outer block-expression will take care of the terminal confluence, so COMBLOCK is called, and that is the end of it. If TERGO is NIL, a confluence point must be created, so TERGO is rebound, a label set into it, and TERMINS, the list of terminal fix-up points, is bound to NIL. COMBLOCK is called to process the statements, the terminal label TERGO is ATTACHed, and then COMTERMINS is invoked to insert the fix-ups at the terminal points.

After compiling the contents of the block, various fix-ups are necessary. COMGOES is called to arrange any branches across fluid restores. If the context is expression, all branches must be completely defined, and an error arises if any are not. For any other context, incomplete branches are NCONCed onto the GOLIST, to be attended to later. A complete discussion is presented in Section 2.5.

2.4 Variables

The management of variable bindings and usage is assigned to functions COMBIND, ORDER, GETFREEV, GETBOUNDV, and GETDEC. The problem is to remember what variables are bound at any given time, what their types are, and whether they are fluid, locative, or both. There are two mechanisms for this--the A-list and the fluid-function-variable storage. The A-list, much like the A-list in LISP 1.5, carries data about current local variable bindings. The FFV storage is examined to determine free variable data, and also examined to determine fluid binding to be established.

The process begins when COMBIND is called, with a list of variables to be bound. Depending on where it is called from, COMBIND does various things. First the individual variable declarations must be processed, for the syntax allows numerous forms of declaration. ORDER is called to unscramble the declaration and sets six fluid variables:

DV	to the variable name, with tailing if present
DT	to the type, if given
DF	to FLUID if it is explicitly fluid
DL	to LOC if it is explicitly by location
DA	to ASSIGNED if it is of assigned type
DI	to any initialization

The next task is to compare the variable with the stored variable information. This is done by GETBOUNDV. If no declaration exists, the variable will be taken as FLUID if it is so declared locally. If a declaration does exist, the variable will be taken as fluid if either a local or a global declaration of FLUID exists. Should the variable turn out to be FLUID in the current section, it is tailed with the section name and returned through the fluid variable DV.

Back in COMBIND, any declaration found must be compared with the local declaration to determine discrepancies and fill in any missing information. Two lists are then assembled: one for the DECLARE statement to LAP, and one for an entry on the ALIST. The former contains only the items required for LAP. The latter has four items per entry: the name, the type, whether it's LOC, and whether it's FLUID in the current section.

The variables are inspected, bound on the pushdown list to their preset values, and then declared for LAP, with fluid binds inserted as required. The A-list is then extended, and COMBIND is done.

When a variable is referenced, the information about it must be extracted by COMEXP. For this purpose, GETFREEV is called. GETFREEV looks for the variable on the A-list and if it is not there, defaults through the sections on SLIST. Should reference to a fluid cell arise, the variable and its data are put on the REFLIST, which becomes one of the LAP inputs. This permits assembly of compiled code at any future time when the declaration used by the

compiler may not exist. If the variable is FLUID in other than the current section name, it is tailed with the appropriate section and returned through DV.

2.5 GO

The GO processor is a complex part of the compiler. The principal problem is that GOes out of a block which binds fluid variables are possible, and when they occur, it is necessary to restore the previous bindings of the fluid variables. The code for fluid-restores is generated by the LAP pseudo-instruction (END). The compiler must, therefore, see that (END) happens at the right time.

The compiler keeps two lists, GOLIST and LABELS, which are re-bound at every block entrance. LABELS is a list of all labels seen so far in the block, and is updated by ATTACHLAB (called from COMBLOCK). GOLIST is a list of entries for GOes to points not on the list LABELS. Each entry consists of an indicator (GO, ADDR, or DECR) consed onto the entire listing at that point. GO is used for unconditional branches (i.e., BUC) where the entire instruction can be replaced; ADDR is used for conditional branch instructions (BOP, BOZ, etc.) where the address can be altered but the op-code must be preserved; and DECR is used for BSX instructions where the decrement may be changed.

2.5.1 COMGOES

COMGOES is called by BLOCK and IFEXP to generate fluid restores for GOes across block boundaries. Its argument is either a list of fluid variables (from BLOCK) or TRUE (from IFEXP). When called from IFEXP, all that is required

is to determine whether all GOes are to defined labels, as GOes out of an expression are not legal. For BLOCK, more is required. The GOes are matched against the labels and any undefined GO is put on a list L. If there are no fluid variables to restore, it suffices to attach (END) and return L. Otherwise, more must be done.

If fluid variables are to be restored, control must pass through an (END). Representative code for (GO X) might be:

```
        ...
        (BSX G1 4 X)      (a)
        ...
        (BSX G1 4 G2)     (b)
G1      (END)             (c)
        (BUC  0  4)       (d)
G2      ...               (e)
X                         (f)
```

The fluid restores here have been made into a closed subroutine. If control can not drop into the place where (END) is attached, the BSX instruction (b) and the label (c) are not necessary and are omitted. The branch instruction (a) starts out on the GOLIST (and after the check pass, on L) as:

(... (GO . ((BUC X) ...)) ...)

The loop at C goes through the list L. A BSX instruction is fabricated, and if the entry is a GO entry, it is put into the listing in place of the BUC instruction. If it were an ADDR entry, such as BOX, the BSX instruction cannot replace it, so the BSX instruction must be put elsewhere, and the BOZ address modified to point to it. If a suitable BSX exists elsewhere on the GOLIST (see GOMEMBER), it is used; otherwise, the BSX is attached as remote code. Finally, a DECR entry (as from a GO crossing two or more fluid-restores) is treated the same as an ADDR entry, except that the decrement (instead of the address) of the original BSX is modified to point to the new one. Finally, a new GOLIST is constructed of all the BSX instructions to be passed back to BLOCK for use at the next outer level.

2.5.2 Switches

A switch declaration generates remote code consisting of a dispatch branch on the accumulator, labeled with the switch name, followed by a dispatch table of BUC instructions. The latter are entered on the GOLIST. The switch name is entered on the list LABELS.

A switch call is recognized by the INSTRUCTIONS for GO. The switch index is COMVALed into the accumulator as an integer; then a branch to the switch name is attached. The handling of fluid restores is identical to that for ordinary GOes.

2.6 IF's, COMPRED, and BRANCHER

The conditional branching logic is one of the longest parts of the compiler. The INSTRUCTIONS for IF determine what context exists, and invoke the appropriate routine. The format for IF is the identifier IF, followed by alternating predicates and expressions or statements. The general approach is to label each predicate and compile the predicates with conditional branches to the next predicate, so that they will be evaluated in order.

The compilation of predicates is done by COMPRED. COMPRED takes three arguments: the predicate expression, a label to which to transfer on truth, and a label to which to transfer on falsity. COMPRED throws the context to predicate expression, and calls COMPACT to do the work. COMPACT binds TGO and FGO, mentioned earlier; at least one of these will be non-NIL; NIL represents a fall-through case. COMPACT actually compiles the predicate, using COMEXP; the V-variables are then used to determine the appropriate branching.

BRANCHER generates the conditional branch instructions. An arbitrary set of branch instructions can be generated, according to the argument of BRANCHER. To illustrate the possibilities, consider the following example:

```
(IF  (LQ I J)  (ZIP))
```

which should generate the following code:

```
        (LDA I)
        (SUB J)
        (BOZ G1)
        (BOP G2)
G1      (CALL ZIP)
G2       ...
```

When BRANCHER is called, the LDA and SUB will have been generated, TGO will be NIL, and FGO will be set to GEN (in this case G2) by IFST. The argument of BRANCHER, when entered from COMREL, is ((TGO (BOZ)) (FGO (BOP)(BOM))). This is interpreted, "On zero, result is true. Otherwise, on positive, result is false; equivalently, on negative (minus), result is true." Note that to reverse the sense of a series of conditional branches, on all but the last, TGO and FGO are interchanged, while on the last, the sense of the branch is reversed. This is the basis of the operation of BRANCHER.

It is sometimes necessary to create a label, if TGO and FGO are not both labels. That is the case in this example, and DGO holds the label. The BLIST entries are scanned for TGO and FGO. If one of these is found, and that variable holds a label, the label is the branch address. If the variable is NIL, for a fall-through case, then a label must be generated (e.g., G1) for the fall-through point.

The syntax of BLIST is:

```
BLIST = (entry*)
entry = (entry-1)|(entry-2)|(entry-3)
entry-1 = gopoint branch | gopoint branch branch
gopoint = TGO | FGO
branch = (BUC) | (BOP) | (BOM) | (BOZ)
entry-2 = field field entry-1
entry-3 = field field field
```

At present, entry-2 and entry-3 are not used. In a list of entry-1's, only the last will have the second form, in which the branch of the reverse sense is given.

2.6.1 IF Statements

IF's in statement context invoke IFST, which does considerable optimizing of GO statements in the IF. A confluence point for the IF is kept in XGO, and one will be created if a suitable one does not exist already. A GO statement determines an explicit label TGO for COMPACT. In the absence of such, a label is generated, passed to COMPACT as FGO, and then attached following the statement. Statements other than simple GOes are compiled by COMSTAT which calls COMEXP with SCLASS bound TRUE.

2.6.2 IF Predicates

IF in predicate position invokes IFPRED. TGO and FGO will be bound, and if one of them is NIL, a label will be created for the fall-through case. Expressions of TRUE and FALSE are recognized as special cases, generating predicate branches directly to the appropriate label. Otherwise, a predicate branch to the next predicate is generated, followed by a predicate branching on the expression to the appropriate labels.

2.6.3 IF Expressions

Two conditions may apply to an IF in expression context: a terminal confluence may or may not have been established. For example, in the expression

(SET A (IF P1 (IF P2 Q R) S))

the outermost IF will establish a terminal point which the innermost IF will also use. The cases are differentiated by examination of TERGO, which contains the label for the confluence point. If no confluence point exists, one is created in IFEXP; TERMINS, LABELS, and GOLIST are also bound. IFEXPT, which takes the case where a confluence point exists, is then called. It uses a function COMTERM to compile a terminal expression. The V-variables for the compiled expression are saved on the LISTING, and all points of LISTING at which this has been done are added to the list TERMINS of terminal exits. When the outermost terminal context is complete, COMTERMINS is called. This takes the list TERMINS, decides what type the terminal expressions should become, generates code to move to this type to the accumulator and splices this code into the LISTING. COMGOES is also called to tie up the GOes.

2.7 AND, OR

AND and OR call a common routine COMBOOL for compilation. COMBOOL creates a label for any fall-through case, then calls COMPACT with the appropriate labels to compile each argument. After the last argument is compiled, any generated label is attached, and VCLASS is set to PREDICATE to indicate that the value of the AND or OR lies in the resulting branch instructions.

2.8 EQ, EQUALN, EQN

EQ is long because there are a large number of possible cases. If the items being compared are both numbers (OCTAL, INTEGER, or REAL), the arithmetic difference is calculated, and branch instructions are attached as appropriate. If both items are BOOLEAN or both FORMAL, they are cheated to OCTAL and treated as numbers. If either item is SYMBOL, a check is made to determine if one item is a SYMBOL identifier. If so, open code using a BXE instruction is generated; otherwise, a calling reference for a symbolic equal subroutine EQUAL. is generated.

EQ and EQUALN call EQHLP with the name of an appropriate function (EQUAL. or EQUALN.). EQHLP calls MAKEPRED if the context is not predicate, otherwise compiles the arguments. Depending upon the types of the arguments, various possibilities arise. For most cases, EQNIL, EQXOR, or EQSUB are used. For arguments of type SYMBOL, a call to the appropriate function will be generated; however, should one of the arguments be a datum identifier, a BXE instruction is generated.

EQNIL is used by EQHLP when, because of type incompatibility, the result is NIL. It takes a list of CLUNKS (i.e., lists of V-variables and LISTING as returned by COMARGS) and attaches the generated code to the LISTING, so that side effects will work.

EQSUB is used when the comparison is to be done by subtraction. It takes a list of CLUNKS (as returned by COMARGS), negates one of the elements, and calls COMARITH with the functional arguments for PLUS. The result is then moved to the accumulator. BRANCHER is called to test the result.

EQXOR is used when the comparison is to be done by exclusive OR. It takes a list of CLUNKS, cheats the elements to OCTAL, and calls WRDHLP, with INSTRUCTION bound to XOR, to generate the XOR instruction. BRANCHER is then called to test the result.

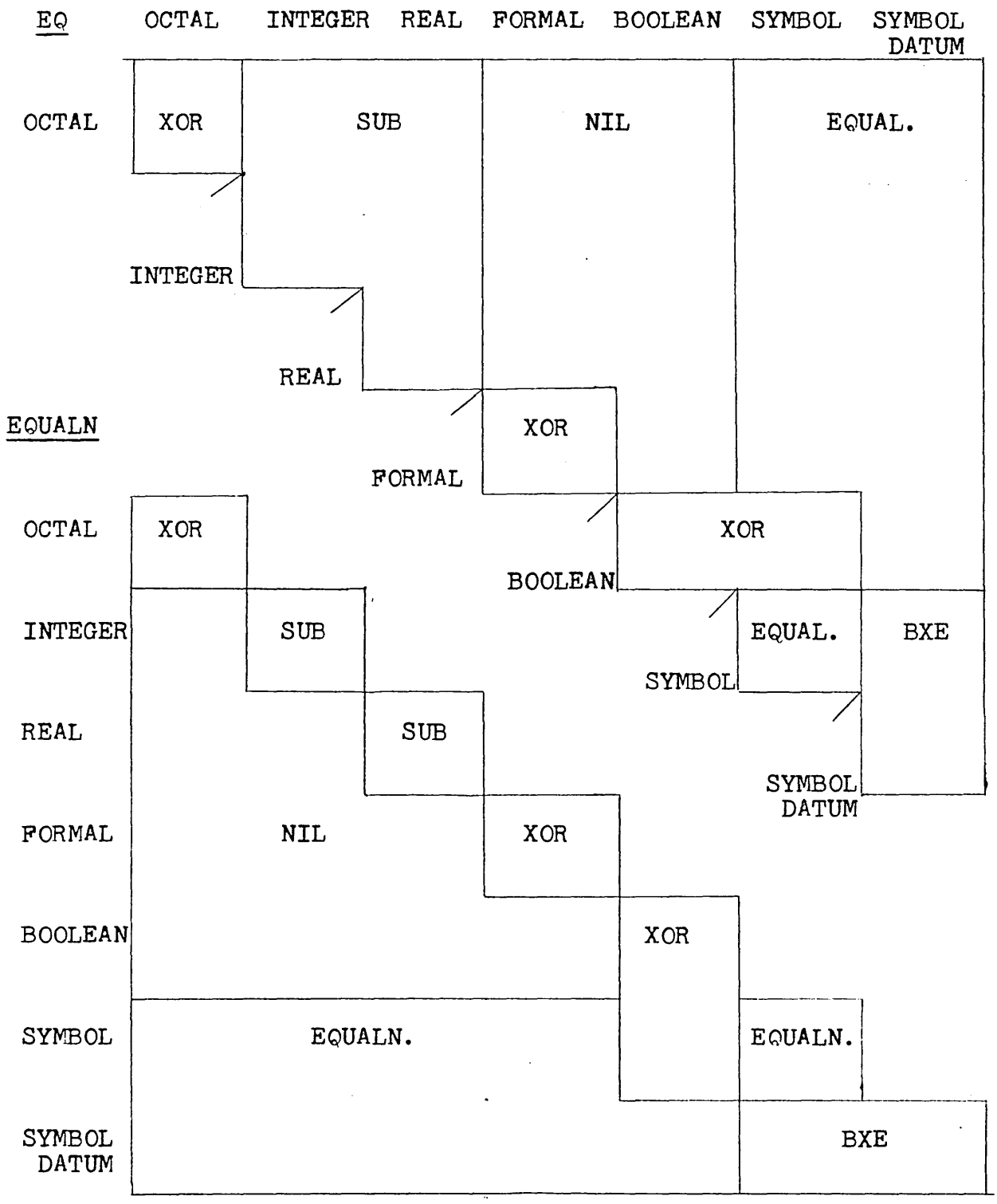

Figure 3. Comparisons Used For EQ, EQUALN

EQN is an INSTRUCTIONS. The context is made predicate by MAKEPRED if it is not so already. The arguments are then compiled as SYMBOL by COMTOP, and passed to EQXOR.

2.9 LS, GR, LQ, GQ

These comparators use COMREL with a suitable set of branches as an argument. COMREL makes a predicate if predicate context does not already exist; COMVALes the difference of the arguments into the accumulator; then calls BRANCHER to attach the instructions to test the accumulator.

2.10 NOT, NULL

NOT and NULL make a predicate if not in predicate context, then interchange TGO and FGO and compile the argument using NOTF.

2.11 Miscellaneous INSTRUCTIONS

QUOTE QUOTE sets the V-variables to SYMBOL, DATUM, the S-expression to be quoted.

LABEL Legal in statement context, LABEL attaches the label to the list of labels via ATTACHLAB, then compiles the associated statement with COMEXP.

RETURN Legal in statement context, RETURN may be used in either predicate or nonpredicate position. If predicate, COMPACT is used to compile the predicate. Otherwise, COMTERM compiles the value, and a branch to the terminal confluence if generated.

2.12 Miscellaneous Functions

IFGO IFGO is a predicate used by IFST. If the argument is of the form (GO label), the value is true, otherwise the value is false.

MAKEPRED MAKEPRED is invoked by various predicates in expression context. It compiles the constructed form: (IF p TRUE NIL).

COMSUB COMSUB is the subscript calculator for COMEXP. The subscript value is compiled into the accumulator, and a full locative made by twin-mode addition of the array head address (at run time). The V-variables are set appropriately.

ANYVARS BLOCK uses ANYVARS to determine whether any variables (excluding switches) are bound by a block. The argument is the declaration list.

COMSWITCH COMBIND uses COMSWITCH to process switch declarations. The arguments are the list of labels and the switch name. The switch name is attached as a label in remote code, followed by a dispatch branch. The dispatch table of individual branches to labels is then constructed. These are entered on the GOLIST. The switch name is entered on the LABELS.

LASTBRANCH LASTBRANCH is a predicate used by COMGOES, COMPACT, and IFST. If the last entry of LISTING is not a label, but is an unconditional branch instruction, then this point in the listing cannot be reached by code generated so far, and LASTBRANCH returns TRUE.

GOGET COMGOES uses GOGET to extract the label from an entry on the GOLIST. See Section 2.5.

GOMEMBER GOMEMBER is used by COMGOES as a semi-predicate. It searches the GOLIST for a BSX instruction identical to one required by COMGOES. If one is found, it is labeled if it does not already have a label, and the label is returned. If none is found, the value is NIL.

ATTACH ATTACH puts one item on the LISTING, via CONS. LISTING is kept in reverse order.

ATTACHGO ATTACHGO puts an unconditional branch instruction on the LISTING. If the branch is to an undefined label, an entry is also made on the GOLIST.

ATTACHLAB ATTACHLAB attaches a label to the LISTING, and also enters it on the list of LABELS.

REMOTE REMOTE is used to attach remote code. The list REMOTES is kept in reverse order. REMOTES is merged with LISTING when the two are reversed at the end of compilation.

BLOTTO BLOTTO sets VBLOT to indicate that no useful values can be depended upon to remain in active registers.

BLOTCH BLOTCH takes a single register name and adds it to VBLOT if it is not already there.

FVTYPE FVTYPE determines the type of a datum. Its value is INTEGER, REAL, or SYMBOL.

ITYPE COMBIND uses ITYPE to determine an initialization suitable to a particular type of block variable.

TYPEP TYPEP is a predicate used by ORDER to determine whether a thing is a type.

FTYPER FTYPER determines the "f-type" for a complete type specification.

3. THE ARITHMETIC PROCESSOR

The arithmetic processor includes routines for arithmetic operations, and also handles logical operators, type conversion, type cheating, and other odds and ends. The principal compiling function used is COMEXP1, which binds SCLASS, PCLASS and TERGO to NIL, then calls COMEXP. The general organization of this part is shown by the flow chart (next page). Following are descriptions of the various functions in this part.

3.1 Macro Expansions: FOR, LIST, DIFFERENCE, RECIP

RECIP RECIP is a macro.

(RECIP exp) = (QUOTIENT 1. exp)

FOR FOR is a macro that expands exactly as specified by the Intermediate Language document, with one exception.

f = ({a_1 |empty}STEP a_2 UNTIL a_3 {{WHILE|UNLESS}p|empty})

If a_2 is a numeric datum, then advantage is taken of this fact by generating:

```
(BLOCK ()
  {(SET v a1) |EMPTY}
L1 {(IF{(NOT p) |p}(GO L2))|empty}
   (IF({LE|GE} (SET v(PLUS v a2))a3 (GO L1))
    L2)
```

TERM TERM is a function of no arguments used by the FOR macro. TERM looks for the phrase WHILE exp or UNLESS exp in a FORELM. If found, an appropriate statement is tacked onto the output of the FOR macro and the phrase is deleted from FORELM.

FORX FORX is a function of one argument, used by the FOR macro. The argument is tacked onto the output list being compiled by the FOR macro.

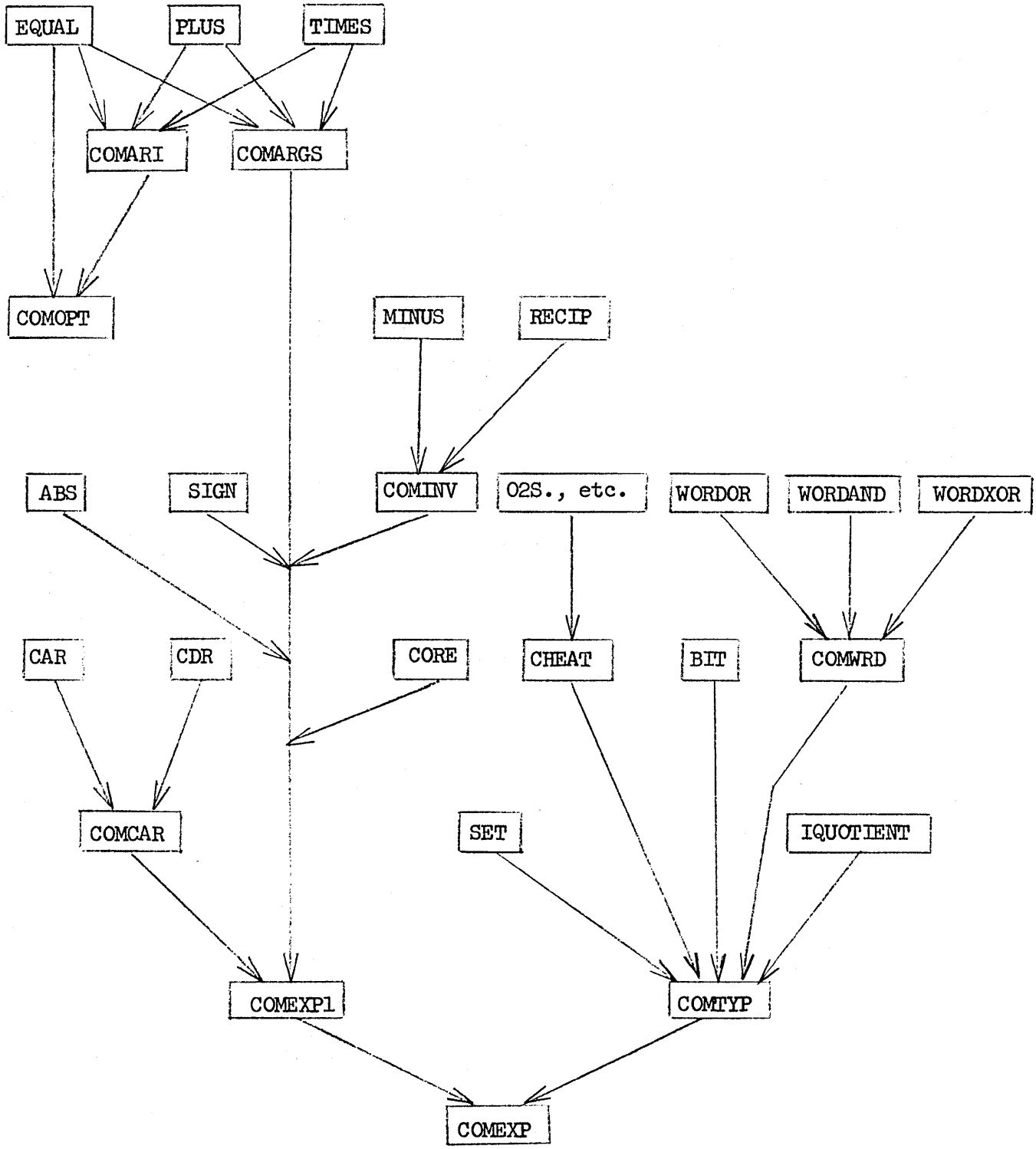

Figure 4. Arithmetic Flow

LIST — LIST is a macro that expands into a series of CONS's.
Example: (LIST A B C)
= (CONS A (CONS B (CONS C NIL)))

DIFFERENCE — DIFFERENCE is a macro that expands in terms of PLUS and MINUS.
Example: (DIFFERENCE EXP_1 EXP_2)
= (PLUS EXP_1 (MINUS EXP_2))

3.2 The Arithmetic Package

The INSTRUCTIONS invoked by arithmetic functions each compile their arguments, generally by calling COMARGS or COMTYP. These call COMEXP, and return fragments of listing and V-variables for each argument. The INSTRUCTIONS then call the optimizer.

The optimizer is in two parts, a parser and a sequencer. The parser (COMARI) separates arguments by type--REAL, INTEGER, or SYMBOL. The sequencer (COMOPT) then determines the appropriate order for arguments of each type, and interspersses the appropriate combining instructions. COMOPT may be recalled to combine the integer part with the real part. Symbol arguments are not treated; calls are generated by the INSTRUCTIONS to appropriate run time symbol arithmetic functions.

COMEXP1 — COMEXP1 is a function of one argument, EXP. PCLASS, SCLASS, TERGO and XTYPE are rebound to NIL. Otherwise, COMEXP1 acts exactly as does COMEXP.

COMTYP — COMTYP is a function of two arguments, EXP and XTYPE. PCLASS, SCLASS and TERGO are rebound to NIL. COMTYP is an active top-driver of COMEXP. If the type of the compiled argument does not agree with XTYPE, then it is converted to XTYPE. Octal and integer types are treated as special cases.

COMTOP COMTOP is a function of two arguments, XTYPE and EXP. It calls COMTYP with V-variables and LISTING rebound. It returns CLUNK of the result, i.e., a list of V-variables and LISTING.

COMARGS COMARGS is a function of no arguments. Each item in an argument position of EXP is fed to COMEXP1 with the V-variables and LISTING rebound. The CLUNK of each compilation is tacked onto a list which is returned.

Example: EXP = (fcn arg_1 arg_2)
COMARGS = ((v's and LISTING for arg_2) (v's and LISTING for arg_1) ())

As is illustrated by the example, the compilations are in reverse order and a NIL is the last element of the list.

COMDAT COMDAT is a function of one argument, a list in the format returned by CLUNK; VCLASS in the list must be of type DATUM. If VINV is not NIL, appropriate arithmetic is done so that VINV will be NIL. The value of COMDAT is a list in the CLUNK format with the datum "smoothed." If VINV contains RECIP, the type will become REAL.

COMOPT COMOPT is a function of five arguments. Argument 1 is a list in the format returned by COMARGS. Arguments 2-5 are functional arguments. COMOPT is a sequencer for combinatorial functions such as PLUS, WORDOR, etc. Two lists are made from argument 1. The first list is of those things which are locatives not modified by an index register and which can be moved "easily" towards an active register (usually, "easily" means in one instruction). This first list is called the LOCLST. All other things are put on the ACTLST. (Datums are considered LOCS.)

The question as to whether a thing may be easily moved is answered by a formal predicate argument (argument 4). If an item is locative but cannot be moved easily, a formal argument is used to make the item active (argument 3). If there are no entries on the ACTLST,

then the V-variables (with the exception of VBLOT) are set to NIL and a function peculiar to the code being generated is called (i.e., an adder for PLUS or a multiplier for TIMES) (argument 2).

If there is something on the ACTLST, then this procedure is followed: The first item on the ACTLST is INHERITed and the listing is put on the latest binding of LISTING. Argument 2 (as described above) is then called. If nothing remains on the ACTLST, COMOPT returns. If there is an item, then the present value computed by argument 2 is moved to the pushdown list with the aid of argument 5 and LOCLST is set to the value (of argument 5). The item from the ACTLST is now INHERITed and argument 2 is called. This process continues until nothing remains on the ACTLST.

COMARI

COMARI is a function of eleven arguments:

argument 1	initial datum
argument 2	a list in the format output by COMARG
argument 3	a formal argument which is a function of two arguments
argument 4-7	a set of four formal arguments for use by COMOPT with fixed point (octal or integer) input
argument 8-11	a set of four formal arguments for use by COMOPT with floating point (real) input.

COMARI is a parser used chiefly by PLUS and TIMES. The list of compiled arguments (argument 1) is parsed into three lists.

1) SYMLST — the list of all arguments with VTYPE = SYMBOL
2) REALST — the list of all arguments with either VTYPE = REAL or VTYPE either INTEGER or OCTAL and RECIP a member of VINV
3) INTLST — the list of all arguments with VTYPE = OCTAL or INTEGER and RECIP <u>not</u> a member of VINV.

Before the parsing is made by type, all datums are taken from the original list and combined by the use of argument 2 (*PLUS, *TIMES, etc.).

If the result of the datum combination is equal to argument 1, the identity element of the group (i.e., $\emptyset$ for PLUS, 1 for TIMES) it is discarded; otherwise, it is tacked onto either REALST or INTLST, whichever is appropriate.

The INTLST is then passed to COMOPT. If anything is on the REALST, the INTLST is "floated" and tacked onto the REALST, which is then given to COMOPT. The final value of COMOPT is INHERITed and LSTLSTed.

The value of COMARI is the SYMLST.

PLUS PLUS is an instruction of no arguments. COMARI is used to compile code for all but the case of SYMBOL arguments. Upon returning from COMARI, the SYMBOL arguments, if there are any, are handled one at a time with calls to one of these: SPLUS, SPLUI, SPLUR, SMINS, SMINI,or SMINR.

PLIMVP,PLRMVP are mover predicates for COMOPT. They use MOVPRD to determine whether a move is "easy" or not. MOVPRD calls EXHOCKY to determine this.

PLSMOV does moving of items which are not "easy" to the accumulator.

PLSPDL moves temporary results to the pushdown stack. If, because of byte activity, the recovery of the result is not "easy," the pushing is by full word; otherwise byte activity is preserved.

PLIALG and PLRALG are formal arguments for COMOPT to generate the arithmetic combining insturction for PLUS. They call PLSALG to do the combination of entries on the LOCLST with anything which may already be active.

TIMES TIMES is an instruction of no arguments. COMARI is used to compile code for all but the case of SYMBOL arguments. Upon returning from COMARI, the SYMBOL arguments, if there are any, are handled one at a time with calls to one of these: STIMS, STIMI, STIMR.

MPIALG, MPRALG are formal arguments to COMOPT to generate the arithmetic combining instructions for TIMES. They call MPYALG to do the combination of the entries on the LOCLST with anything which may already be active.

COMINV COMINV is a function of three arguments:

argument 1	the inversion being compiled (MINUS or RECIP)
argument 2	the inversion not being compiled (MINUS or RECIP)
argument 3	a format argument of one variable that will perform the conversion for data.

(CADR EXP) is compiled using COMEXP1.

If VCLASS is datum, then all inversions are taken care of at compile time; if not, VINV is set to take care of the required inversion. A minus of a minus will be a no-op. (RECIP is not presently implemented in this manner.)

MINUS MINUS is an INSTRUCTION using COMINV.

ABS ABS is an instruction of no arguments. (CADR EXP) is compiled using COMEXP. If VTYPE is SYMBOL then the SYMBOL pointer is moved to the AC and SYMABS is called. If TYPE is not SYMBOL, in-line

code is generated. If VCLASS is datum, the absolute value of the datum is found at compile time.

SIGN — SIGN is an instruction of no arguments. (CADR EXP) is compiled and moved to the AC. If VTYPE is SYMBOL, SYMSGN is called. If not, in-line code is generated.

DIVIDE. — DIVIDE. is a function of two arguments, a type and an instruction for use by EXHOCKY. (CADDR EXP) is compiled and top driven to the specified type by COMTYP. If VREG is true or £XHOCKY is false, the argument is moved to the pushdown stack. (CADR EXP) is then COMVALed to the specified type in the AC. The first compiled argument is INHERITed so that the calling instruction may perform the necessary division.

IQUOTIENT — IQUOTIENT is an instruction that calls DIVIDE, then ATTACHes appropriate instructions to do an integer division. The result of an integer division is the full-word B register.

QUOTIENT — QUOTIENT is an instruction that calls DIVIDE, then ATTACHes appropriate instructions to do a floating point division. The result of a floating point division is the full-word AC.

3.3 Partial Word and Logical Operations

WORDOR, WORDAND, WORDXOR — These use COMWRD with an appropriate argument to compile code.

COMWRD — COMWRD is a function of three arguments.

argument 1 ≡ group identity element
argument 2 ≡ an instruction mnemonic
argument 3 ≡ a formal argument used for data collection

COMWRD compiles all expressions in argument position of EXP. The compilations are top driven to type OCTAL by COMTYP. All arguments of class datum are collected by argument 3 to produce a single datum. If this single datum is equal to argument 1 then it is disregarded. The compiled list of arguments is now passed to COMOPT. If either no arguments were present or all were datum that degenerated to argument 1, then argument 1 is used as the run time value of this function. Otherwise, the value is produced by the code from COMOPT.

WRDHLP WRDHLP is used by COMWRD. It calls COMOPT with functional arguments appropriate to word operations.

CAR CAR is an instruction of no arguments that calls COMCAR with an argument (24 24).

CDR CDR is an instruction of no arguments that calls COMCAR with an argument (0 24).

PROP PROP is an instruction of no arguments that calls COMCAR with an argument (0 18).

COMCAR COMCAR is a function of one argument, a list of two intergers in VBYTE format. (CADR EXP) is compiled. If VTYPE is not SYMBOL, an error message is issued.

VINDX is used to determine if indirect addressing is permissible through the argument. If it is, VIND is set TRUE and VBYTE is set to the argument of COMCAR. If not, the argument is moved to the accumulator and the V-variables are set to reflect that this is a SYMBOL in location 0, AC and that it lies in the portion of the word indicated in the argument to COMCAR.

CORE CORE is an instruction of no arguments. (CADR EXP) is compiled. If VCLASS is DATUM, the datum is used as the absolute address of the value of CORE (at execution time). If VINDX is true, then the V-variable is set to indicate an indirect address through the LOC. If none of the above are true, the expression is moved to the accumulator, and the V-variables are set to indicate address 0, AC.

In all cases, the V-variables are set to full word, octal locative.

BIT BIT is an instruction of no arguments. EXP is assumed in the format (BIT A_1 A_2 A_3).

If either A_1 or A_2 are not numeric data, then the function BITS in section SYS is invoked.

A_3 is compiled by COMTYP and driven to OCTAL. VBYTE is now set to reflect the active bits of the result. If it is necessary because of a prior condition of VBYTE, a move will be made by MOVACTIVE.

3.4 SET and LOCSET

SET SET is an instruction of no arguments. SET is the assignment logic of the LISP 2 Compiler. The action is to assign the value of the right-hand side of an equation to the left-hand side.

The following division of responsibility is made:

1) If the moving of the right side to the left side will clobber a thing wanted to be saved, the mover MOVLOC will be responsible for protecting the necessary register(s).

2) If the code generated by the left side will cause the clobbering of something to be saved, then it is SET's responsibility to do the protecting.

Three different cases are taken care of by SET:

1) Statement context. The right-hand side is compiled by COMTYP and top driven to the type of the left-hand side.

2) Expression context and XTYPE equals the type of the left-hand side: The right hand side is compiled by COMTYP and top driven to the type of the left-hand side. Facilities are set up to remember the value of the right-hand side.

3) Expression context and XTYPE is not the type of the left side: The right-hand side is compiled by COMEXP1 with no top driver. Facilities are set up to remember the value of the right-hand side.

In expression context, the value of the right-hand side is left in an active register whenever no extra work is involved.

LOCSET LOCSET is an instruction of no arguments. The left and right-hand side are compiled and checked for a locative variable and a full locative, respectively. The types of the left and right-hand side are checked. If they are not the same, an error message is given. (Octal and integer are not considered the same types.)

MAKELOC is called to make the locative pointer, then a STF instruction is ATTACHed onto the listing.

3.5 Type Conversion and Cheaters

S20., R20., B20., I20., F20., 02S., 02B., 02I., 02F., 02R.

These are the "cheater functions," (actually INSTRUCTIONS). They call CHEAT with appropriate arguments.

CHEAT CHEAT is a function of two arguments, a type to be cheated from and a type to be cheated to.

CHEAT compiles (CADR EXP) using COMTYP with XTYPE = "type cheated from." VTYPE is then set equal to "type cheated to."

Three special cases are handled:

- to INTEGER
- from INTEGER
- to REAL

For these special cases, the expression will be driven to a full word quantity if it is not one already.

3.6 Miscellaneous Service Functions

VLIST — VLIST is a function of no arguments. The value of VLIST is a list of the latest bindings of the V-variables. (VCLASS VTYPE VREG VADDR VIND VBYTE VBLOT VINV)

VSET — VSET is a function of one argument, a list in the same format as made by VLIST or CLUNK. The value of VSET is NIL. The action of VSET is to restore each V-variable binding to the appropriate part of the argument list.

CLUNK — CLUNK is a function of no arguments. The value of CLUNK is a list in the same format as VLIST with an additional element which is the latest value of LISTING.

INHERIT — INHERIT is a function of one argument. INHERIT works the same as VSET except that VBLOT is set to the UNION of the present VBLOT and the VBLOT in the list.

GVCLASS, GVTYPE, GVERG, GVADDR, GVIND, GVBYTE, GVBLOT, GVINV are functions of one argument, a list in the format as returned by VLIST or CLUNK. The value of GV__ is the appropriate V-variable in the list.

LSTLST — LSTLST is a function of one argument, a list of LAP instructions. LISTING is set to the NCONC of the argument and the present value of LISTING. The list must be in reverse order (as is LISTING).

COMLCK — COMLCK is a function of one argument. If the length of EXP does not equal the argument, an error message is issued, the V-variables set to DATUM 0Q and a value of TRUE is returned. Otherwise NIL is returned.

RESTORE RESTORE takes an argument in CLUNK format (V-variables and listing). The V-variables are set and the listing LSTLSTed (NCONCed onto LISTING).

VINDX VINDX is a predicate function of no arguments. If the present bindings of the V-variables will allow indirect addressing to be added to a locative description, then the value of the function is TRUE, otherwise the value is NIL.

4. THE "MOVER"

The "Mover" is a family of functions that are used within the LISP 2 compiler to generate LAP code for moves between machine registers. The present state (present location in the computer) is given by the setting of the V-variables (described in Section 2.1) and the desired state (where to move to) is given by the arguments to the function called. The function arguments are counterparts of the V-variables (and will be called the X-variables) such that at the end of the "move" the V-variables have been changed to the X-variables.

The desired class is never specified explicitly but is implicit in the function called. The function MOVACTIVE always produces VCLASS = ACTIVE and the function MOVLOC always produces VCLASS = LOC. No function produces VCLASS = DATUM. XINV is never specified; the assumption is that VINV is to be considered in the move and always left in the NIL state when through. The other X-variables are specified only if they are necessary for the complete description of the desired class.

The V-variables and X-variables not only describe the machine location where something is to be found, but also describe the type of data there and the position within the machine register if it is not a full word. When a move is made, type conversion, packing, and unpacking will be done if it is necessary for the transformation from the V-variables to the X-variables.

The "Mover" does not consider what registers it may destroy in a move. It arbitrarily selects and uses any register that it needs. The registers that are destroyed are listed, and it is the responsibility of calling functions to save and restore such registers if they are needed after the move. Errors are recognized and appropriate messages are printed, but some move will be made regardless. All LAP instructions generated are put on the LISTING.

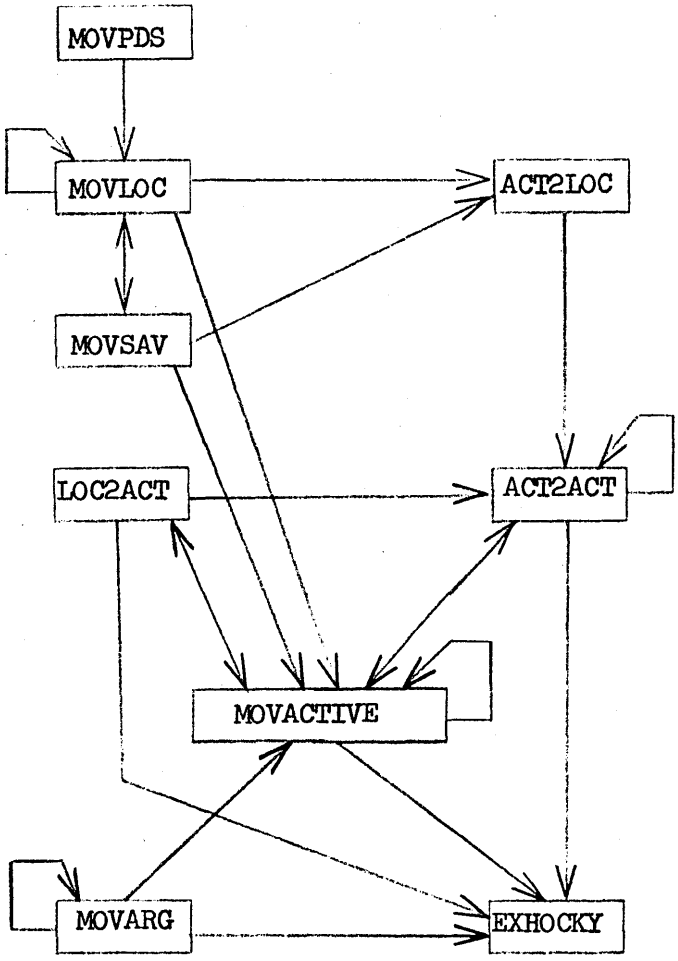

Figure 5. The "Mover" Flow

4.1 Type Conversion

The basic type conversion data comes from function CONVP. Conversions are generated by function CNVL2AC. The values of CONVP are shown below.

Values of CONVP

VTYPE	XTYPE					
	OCTAL	INTEGER	REAL	SYMBOL	BOOLEAN	FORMAL
OCTAL	V	OI	IR	FUNCTION OCT2SYM	TRUE	NL
INTEGER	MZ	V	IR	FUNCTION INT2SYM	TRUE	NL
REAL	FUNCTION OCTROUND	FUNCTION ROUND	V	FUNCTION REAL2SYM	TRUE	NL
SYMBOL	FUNCTION SYM2OCT	FUNCTION SYM2INT	FUNCTION SYM2REAL	V	SP	FUNCTION SYM2FORM
BOOLEAN	NL	NL	NL	V	V	NL
FORMAL	NL	NL	NL	FUNCTION FORM2SYM	TRUE	V

The value retrieved from this table is used in two different ways depending on whether VCLASS is DATUM (a compile-time conversion) or VCLASS is ACTIVE or LOC (a run-time conversion.)

Value of CONVP	Value of CNVL2AC	Result, VCLASS=ACTIVE or LOC
FUNCTION name	5	Generate instructions to call the named function
TRUE	1	Change the V-variables to DATUM, BOOLEAN, TRUE
IR	0	Generate open code to float the integer or octal
MZ	NIL	Generate open code to test and change minus zero
SP	NIL	Generate open code to test the value of the symbol
OI	3 or NIL	Make the move first then change VTYPE to XTYPE
V	2	Change VTYPE to XTYPE
NL	0	Not legal, but VTYPE is set to XTYPE

Value of CONVP	Value of CNVL2AC	Result, VCLASS=DATUM
FUNCTION name	5	Apply the equivalent LISP 1.5 function to VADDR
TRUE	1	Change the V-variables to DATUM, BOOLEAN, TRUE
IR	0	Apply LISP 1.5 FLOAT to VADDR
MZ	NIL	Set VADDR to 0 if it is -0.
SP	NIL	Test VADDR, and if not NIL set to TRUE
OI	3 or NIL	Change VTYPE to INTEGER
V	2	Change VTYPE to XTYPE
NL	0	Print error message

For the compile-time numeric conversions there is an additional check on VADDR to determine if it really is a number before doing the conversion.

4.2 MOVPDS

MOVPDS sets up an address on the push-down stack and has MOVLOC move to that address. VADDR is then set to POP.

4.3 MOVLOC

MOVLOC generates instructions to store into a core location specified by XADDR, XREG, XIND. If XREG will be destroyed by the move it is saved by MOVSAV before generating any instructions.

Type conversion is done first if necessary in the same fashion as MOVACTIVE. Then, if VCLASS is ACTIVE, the function ACT2LOC is called to finish the move. In the case of DATUM or LOC, an active register is chosen on the basis of bit activity required. A MOVACTIVE is done, then ACT2LOC is called.

4.4 ACT2LOC

ACT2LOC moves from an active register to core. It determines if the move can be made in one instruction of the STX, STF, or STA class. If so, then the instruction is attached to the listing, the V-variables are changed to LOC of VTYPE in XADDR, XREG, XIND at XBYTE, and exit is made. Otherwise an intermediate register is used to position and/or pack the bits to be stored, the V-variables are reset, and exit is made.

4.5 MOVSAV

MOVSAV computes the effective address of XADDR, XREG, XIND and saves it on the push-down stack. A MOVLOC is then made going indirectly through the saved address. The push-down stack is balanced afterward so that the number of pushes and pops match. The V-variables are all reset to NIL since XREG has been destroyed and there is no way to recapture what has just been stored.

4.6 LOC2ACT

LOC2ACT is called to load a LOC into an active register. If VCLASS is discovered to be ACTIVE (by using L2AP.) then ACT2ACT is called instead. A LDA or LDX

instruction is requested from EXHOCKY which,if obtained, is attached to the listing and exit is made. If not obtained, an intermediate register (AC, B, or L) is chosen,and the move is made in two steps. The register is chosen on the basis of the bit activity required to complete the move. Having chosen, a MOVACTIVE is done and ACT2ACT is then called to complete the move.

4.7 ACT2ACT

ACT2ACT divides its work into three parts--a change of registers only, bits within a register only, or both. (The case of neither of these being required may be discovered at various points; a prompt exit results.)

When the move requires only a change of registers, then either a full word load or store instruction (whichever is more convenient) is generated and exit is made.

When only bit position within the same register is to be changed, the question of which register is involved becomes important because of the nature of the various machine registers. If the goal can be achieved by self-loading with appropriate instruction modifiers, it is so done. If not, then the part of the program beginning at ABSHFT sets up shifting instructions, followed by masking if necessary to remove excess bits. It is necessary in some cases to move to the AC or B registers to do the bit manipulation.

If both registers and bit position require changes, then the registers involved are examined to see which change should be made first (if both can not be done simultaneously, according to EXHOCKY). For example, if shifting is required, it must be done at the point in the move when the AC or B register are in use. Consequently, quite a bit of worrying is done about which registers are on each end of the move, and whether an intermediate register has to be chosen to accomplish the bit manipulation. The move is generally done in parts. Either XREG or XBYTE is fixed up as required, and the remainder of the work is done by the part of the function that handles only one kind of change.

4.8 MOVACTIVE

MOVACTIVE generates instructions to move from the present position, given by the V-variables, to the position described by its arguments, XTYPE, XREG, and XBYTE. If the two positions are in fact the same, no instructions are generated. In all cases the V-variables will be reset to ACTIVE of XTYPE in XREG at XBYTE; any registers used will be blotted, and errors that are discovered will be printed (with default conditions chosen so that some move is made regardless of errors).

The function does type conversion first and then uses subsidiary functions to finish the move; the most important of which is ACT2ACT for the ACTIVE to ACTIVE case. Class LOC is loaded into an active register as profitably as possible, and then ACT2ACT completes the move.

MOVACTIVE itself only does type conversion and the DATUM moves. For DATUM the conversion is done at compile time using the function CNVD. An instruction of the LDA or LDX class is obtained from EXHOCKY for loading the converted DATUM and is attached to the listing. XREG is blotted and the V-variables are reset. The DATUM case is then finished and exit is made.

For type conversion of a LOC or ACTIVE the function CNVL2AC is utilized. This function attaches instructions to the listing which at run-time would load the AC and call the appropriate function for type conversion; in some cases, the conversion is open-coded.

4.9 MOVARG

MOVARG manufactures a LAP instruction of a certain class if the requested move can be made in one instruction. If it can be done, the CDR of the instruction is the value of the function and the V-variables are not changed. If it is impossible, then MOVACTIVE is called to make the move, instructions are attached to the listing, the V-variables are changed, and the value of MOVARG is NIL.

The instruction class BXE is given to the function MDECR; all others are given to EXHOCKY. If type conversion is required, MOVACTIVE is called except when the code obtained from CONVP is OI, V, or TRUE. The table presented in Section 4.1 gives the CONVP codes.

4.10 EXHOCKY

EXHOCKY is a semi-predicate which determines if the entity described by the V-variables can be moved to an active register in exactly one instruction. The V-variables are tested but are never reset.

The arguments are C, an instruction class, and B, a bit descriptor. The particular active register is unimportant and unknown, since the instruction class implicitly contains this information. The function value is either a partial LAP instruction or NIL, indicating that the move can not be done in one instruction. Possible values of C are LDA, LDX, and FAD. The bit descriptor B is usually a list of two integers where the first integer is the right-most bit to be filled and the second is the number of bits to be filled.

For the LDA class if the V-variables are ACTIVE or LOC, only one question is significant. If the move can be made without shifting and/or masking, then it can be done in one instruction, and all that is required is to make up the instruction modifiers. This is done at A0 and the instruction is returned. Should shifting or masking be required, NIL is returned.

If the V-variables are for a DATUM move, more can be done. At DA the various types of DATUM are separated and subsidiary functions (NADDR, SYMOD, and SHFTRA) are called to make up the instruction. These functions all have the property that if the datum can be converted, shifted, or masked into the form desired, this is done; and then the LDA instruction for the new datum is manufactured. If it is impossible, NIL is returned.

The FAD class is quite simple since only full words are permitted. For classes ACTIVE and LOC, the type must be REAL. If it is REAL, the appropriate instruction modifiers are computed and the instruction is returned; otherwise NIL is returned. For DATUM, the type may also be INTEGER or OCTAL, in which case a compile-time conversion is done and the instruction is returned. Otherwise the result is NIL.

The LDX class requires B, the bit description, to have 0 as the right-most bit. For the ACTIVE or LOC cases VBYTE must be (0 18) or (24 18). If this is so, the LDX instruction with the appropriate modifiers is made up and returned; otherwise the result is NIL. For a DATUM, VBYTE is forced to be (0 18) if possible, and the instruction is returned.

4.11 MAKELOC

MAKELOC is used to compute the address of a locative variable. In order to have an address the variable must be in core, hence VCLASS not LOC is an error. (Locative is to be distinguished from LOC, one being the transmission mode of a LISP 2 entity and the other a setting of the V-variables used by the compiler.)

Three cases are recognized as computable: any indirect address, a direct address of zero modified by the AC as an index register, or a direct address of zero modified by any other index register. In these cases the instruction to load the AC with the address is attached to the listing and the V-variables are set to ACTIVE, SYMBOL in AC full word.

4.12 Other Mover Functions

Function	Arguments	Description
ACEQ	R1 (an active register) R2 (an active register)	If both R1 and R2 are the AC then TRUE, else FALSE.
ADDRMODS	I (indirect indicator) R (indexing indicator)	When a load or store class instruction is to be generated then the LOC class modifiers, VIND and VREG or XIND and XREG, are used as arguments to this function to generate the instruction address modifiers. The value is a LAP tag field.
ATTACH1	I (a LAP instruction)	I is edited to remove unnecessary fields and redundant NIL's and 0's. Side-effect: Attaches I to the LISTING.
BACTIV	L (a list of byte names)	Used to compute the octal number for the tag field of a CON instruction which tells what bytes are to be active. The value is the octal.
BEQ	B (a bit indicator)	If VBYTE is the same as B then TRUE, else NIL.
BMODS	C (a class indicator) B (a bit descriptor)	When a move to an active register or to a core location can be done with byte modifiers then the function returns the modifier, else it returns 0 meaning that shifting is necessary.
BEND	B (a bit descriptor)	If B is on byte boundaries (6 bit variety) exactly, then TRUE, else FALSE.
CADRNIL	None	Used as the functional argument for SASSOC when a function is required whose value has a NIL CADR. The value is (() ()).
CANSTZ	B (a bit descriptor) I (a partial LAP instruction)	When VCLASS is DATUM and a move to a core location is requested, then this function receives the bit position information, B, and the address and tag of the store instruction, I. Then if STZ can be used (V-variables are tested) the instruction is made up and attached and TRUE is returned; otherwise NIL is returned.

Function	Arguments	Description
CLVINV	X (an identifier)	Used by ISINV. Finds the first occurence of X on the list VINV, removes it and returns TRUE. If none is present, returns NIL. Side-effect: Resets VINV.
CNVD	T (a type indicator)	When a datum is to be converted this function resets VADDR to the converted value and VTYPE to the new type. CNVDATM is used for the value conversion. Side-effect: Resets the V-variables.
CNVDATM	T1 (a type indicator) V (a value) T2 (a type indicator)	CONVP is used to test if type conversion from T1 to T2 is legal; if not, NIL is returned. If so the datum, V, is converted to type T2 and the new value of V is returned.
CNVL2AC	F (a type descriptor) B (a bit descriptor)	When a move requires type conversion this routine generates the necessary instructions to do the conversion leaving the result in the AC at B. MOVACTIVE is used and the V-variables are reset unless the conversion is illegal or trivial. The value is integer or NIL. Side-effects: Resets all the V-variables.
COMPMSK	B (a bit descriptor)	A mask is manufactured such that an AND with a full word would erase the bits indicated by B. The address of the mask is the CDR of the returned LAP instruction, and the CAR is LDA.
CONVP	T (a type descriptor)	This is the type conversion "table." VTYPE is used free and if the conversion to T is legal the function name or open code indicator to be used is returned, else NIL.
DXREG	R (an active register)	VREG or XREG may be the argument. The function value is used to generate the correct class instruction to load R. The value is an instruction class name (LDA or LDX).
FULLW	B (a bit descriptor)	If B is full word then TRUE, else FALSE.

Function	Arguments	Description
ISINV	X (an identifier)	X is usually MINUS; if X is found an odd number of times on the list VINV then TRUE, else FALSE. (0 occurrences give FALSE). Uses CLVINV to count occurrences of X and reset VINV. The value is Boolean. Side-effects: Removes all occurrences of X from VINV. Resets VINV.
ITSTRU	R (an active register) B (a bit descriptor)	When a type conversion to Boolean is legal and yields TRUE, then this function first sets the V-variables to DATUM, BOOLEAN, TRUE and then uses MOVACTIVE to move to R at B. Side-effect: Resets the V-variables to ACTIVE, BOOLEAN, in R at B.
L2AP.	None	If the V-variables indicate class LOC and it is actually an ACTIVE disguised as a LOC (by using the machine address of the active) then the V-variables are reset to ACTIVE, and TRUE is returned; otherwise nothing is done and NIL is returned. Side-effect: Resets the V-variables.
LDCMP	R (an active register)	Used when VINV has already been found to be MINUS and AC, B, or L is to be loaded. The value is an operation code of LDC, LBC, or LLC.
LOPC	R (an active register)	Used to obtain the operation code to load AC, B, or L. Tests if VINV is MINUS and returns a load complement, a regular load, or an error indication. Side-effect: Resets VINV.
LSYMNS	None	If a SYMBOL is to be moved anyplace and VINV is MINUS then this function attaches instructions to the listing which remove the MINUS condition. Side-effect: The V-variables are reset to ACTIVE, SYMBOL, in AC, full word.
LXN	N (an integer)	If the integer argument is suitable for loading via immediate addressing then the list of the argument and R is returned, otherwise NIL is returned.

Function	Arguments	Description
LXRM	B (a bit descriptor)	Used when a LDX class instruction is being made up. If LA or RA modifiers should be used (according to value of B) they are returned, else NIL.
MLDX	I (a partial LAP instruction) X (an index register)	When a LDX instruction has been generated by EXHOCKY the operation code is missing and the actual index to be loaded is not included. MLDX completes the instruction and returns it.
MDECR	None	Used by MOVARG when a BXE class has been requested. The V-variables are tested and the appropriate decrement is generated. The value is a LAP decrement field or NIL.
MMSK	B (a bit descriptor)	Used to compute a mask which if AND'ed with the AC would erase all bits except those specified by B. The value is an octal.
NADDR	V (a value) T (a type indicator) B (a bit descriptor)	When a datum is to be loaded into AC, B, or L this function is used to compute the instruction address, tag, and decrement. If B is not on byte boundaries SHFTRA is called, otherwise the instruction or NIL is returned. NIL means that more than one instruction is needed. The value is a partial LAP instruction, or NIL.
REVA2L	R (an active register)	Used to convert an active register name into a LAP address. (Example: if the argument is AC, the value is A. .) Uses TRANSA2L after binding VREG. The value is the LAP address of R.
SETTRU	None	Used by ITSTRU when a conversion to BOOLEAN is legal and TRUE. Side-effect: Resets V-variables to DATUM, BOOLEAN, TRUE.
SHFTRA	V (a value) T (a type indicator) B (a bit descriptor)	When a datum is to be loaded into AC, B, or L not on byte boundaries, this function is called to shift and mask the datum so that the load may be done in one instruction. If successful the instruction is returned, else NIL.

Function	Arguments	Description
SPARAM	B (a bit descriptor)	When either VBYTE or XBYTE is not a full word, and the move can not be made by using byte modifiers, then this function is called with XBYTE as an argument. It uses VBYTE free and computes the shift and determines if masking is required. The value is a pseudo-instruction for shifting only or for shifting and masking.
STOC	R (an active register)	If R is a legal register for the STA class of instruction then the proper operation code is returned. Otherwise COMERR is called and a pseudo-operation is returned.
STXR	B (a bit descriptor)	When an STX class instruction is manufactured this function is used to determine if the STX is for RA, LA, or illegal, according to the value of B. The value is RA, LA, or NIL.
STXREG	R (an index register) P (an address on PDS)	Called by SVXREG to attach instructions for saving R on the push-down stack.
SVACT	R (an active register)	Used to reset the V-variables to describe an ACTIVE, in the register R at bit position B. Side-effect: Resets the V-variables except VTYPE, VINV, VBLOT.
SVLOC	A (a LAP address) X (an index modifier to A) I (the indirect indicator) B (a bit descriptor)	When a move to LOC has been made, this function is called to reset the V-variables to describe the LOC. Side-effect: Resets the V-variables except VTYPE, VINV, VBLOT.
SVXREG	A (a LAP address) R (an index register)	Used to save an index register before doing a move when the register would be destroyed by the move. Called by MOVSAV.
SYMOD	S (a datum symbol)	Used when the V-variables are for a datum of type SYMBOL. The value of the datum is the argument and the correct LAP address field is computed. The value is the LAP address used to load the datum. Examples: (ID A) (NUMBER 3.0) etc.

Function	Arguments	Description
TAGF	T1 (a LAP field) T2 (a LAP field)	When more than one instruction modifier has been determined (as from ADDRMODS and BMODS) then this function combines them into a single LAP field for the instruction. The value is the LAP tag field.
TRANS	B (a bit descriptor)	If B is on byte boundaries, then the list of SCAMP byte names corresponding is returned. Example: B=(12 18) returns ('3 '4 '5). Full word returns ('7), otherwise NIL is returned.
TRANS1	B (a bit descriptor)	Used by TRANS to translate a bit descriptor to a list of byte names. The value is a list of byte names, or NIL.
TRANSA2L	None	Used to obtain the address for an ACTIVE to ACTIVE move. Example: (LDA (Z. 3)). VREG is used free. It is assumed that VCLASS is active. The value is a LAP address of an active register.
WHATBITS	B (a bit descriptor)	Used whenever it is necessary for VBYTE or XBYTE to be in the form (R N) where R is the right-most bit and N is the number of bits. The value is a list of the right-most bit and the number of bits used by the argument.

INDEX OF COMPILER FUNCTIONS

INDEX OF COMPILER FUNCTIONS (Cont'd)

INDEX OF COMPILER FUNCTIONS (Cont'd)

INDEX OF COMPILER FUNCTIONS (Cont'd)

DISTRIBUTION

B. Barancik	2105
J. Barnett	2059
E. Book	2332
R. Bosak	2041
J. Burger	9919
D. Drukey	2105
S. Feingold	9525
D. Firth	2310
E. Jacobs	2344
S. Kameny (100)	2009
E. Myers	2227
M. Perstein	2332
V. Schorre	2330
J. Schwartz	2123
R. Simmons	9439
E. Stefferud	9734
A. Vorhaus	2213
C. Weissman (10)	2214