Implementation Guide to the Exhaustive Search Code Generator

David W. Krumme

Tufts University Computer Science Department
Technical Report 83-1

April 4, 1983

This is a general guide to the implementation details of the exhaustive search code generator described by Krumme and Ackley: "A practical method for code generation based on exhaustive search," SIGPLAN Notices 17(6): 185-196, 1982. It includes program and table listings with commentary and annotation and includes a reasonably self-sufficient excerpt from the system that was in use as of Spring 1982. It is an excerpt only, not a complete working system. Before reading this, one should read the above-mentioned paper and have on hand a copy of the Sargasso C Compiler User's Manual (available from the author) to explain nonstandard constructions that will be found in the programs. [Probably the most important of these is the form printf("%Uxxxx", target, arg, arg) which directs printf's output to target which is typically a file designator.]

There are 10 files listed here: COSTS.TRN, TABLES.DAT, TRANS.C, INTOPS.HDR, MODES.HDR, FNCTAB.C, FOREST.C, C2.HDR, CODE.C, and FUNCS.C. COSTS.TRN and

TABLES.DAT serve as input to the TRANS program whose purpose it is to convert them to other formats and which produces INTOPS.HDR, MODES.HDR, FNCTAB.C, and FOREST.C as output. The code generator intself is contained in CODE.C; when it runs as a part of the compiler is is loaded with compiled versions of FNCTAB.C, FOREST.C, and FUNCS.C. The header (.HDR) files mostly contain definitions used by various different parts of the compiler; C2.HDR contains definitions used in the back end (code generation pass) of the compiler. Of particular interest in C2.HDR is the layout for "numbers," which represents a standardized representation for run-time operands (constants, variables, registers, etc.). In "code", rval1 and rval2 comprise such a "number", and all "@", "%", and "spval" quantities are likewise. (See "code.")

THE CODE TABLES

The syntax of each code table is approximately as follows. Lower case is used for nonterminals, all-caps for terminals.

```
table -> tblname flags altlist

tblname -> [PLII | ^DI | <STP | . . .

flags -> EMPTY | ( flaglist )

flaglist -> flag | flag , flaglist

flag -> COMMUTE | UNIVERSAL | NOUNIV | . . .

altlist -> alt | alt altlist | lblref

alt -> lbldef alt
       | { key ( arglist ) codelist returnlist }
       | { key ( arglist ) codelist altlist }

lbldef -> : IDENT

lblref -> * tblname : IDENT

key -> recur | func | recur ( arglist ) | func ( arglist )

recur -> L mode | R mode | S mode


mode -> VA | VM | E | . . .

func -> IDENT

arglist -> arg | arg , arglist

arg -> @ NUMBER | % NUMBER | NUMBER

codelist -> EMPTY | code codelist

code -> . opcode . mod . ac . m .

opcode -> MOVE | MOVEM | JFCL | JFFO | . . .

mod -> NUMBER

ac -> % NUMBER | NUMBER

m -> % NUMBER | NUMBER

returnlist -> EMPTY | return returnlist
```

```
    return -> func ( arg )
            | mode ( arg )
```

An extensive set of tables has been included with this document in the file TABLES.DAT. This was excerpted directly from the code tables that were in use as of Spring 1982. All tables that are present are listed in their entirety. However many tables are missing. Tables left out were those that deal with operands of type short, long, character, and pointer to character, although some character-related things were left in. These tables should provide full implementation for all operations on integers, floats, doubles, and pointers to these three. (However, there might be an occasional stray operation that got left out anyway.)

THE TRANS PROGRAM

The TRANS program translates the code tables from human-readable to machine-readable form, making a few significant transformations to them and producing also several auxiliary tables used for various purposes. The TRANS program is in essence a compiler for the code tables that is run only once. The "code" routine is an interpreter that runs at code-generation time interpreting the compiled tables in order to translate parse trees to MACRO-10.

The input to the TRANS program must be a sequence of files whose contents are as follows: first, a list of the machine OP codes with their nominal execution times (taken from an appendix to the DECsystem-10 System Reference Manual) and contained in COSTS.TRN, followed by a list of the names of the modes and then the tables themselves, contained in TABLES.DAT. Each table begins with the operator’s name, possibly followed by some flags describing properties of the operator, followed by the contents of the table.

The most important alteration the TRANS program makes to the tables is that it provides a kind of lookahead for potential returns on the evaluation modes.

Bits telling whether nested alternatives contain any returns on each mode are recorded throughout the tables. These bits are checked by the interpreter so that it can avoid making recursive subevaluations that have no chance of succeeding because there will be no returns.

The output from the TRANS program consists of the file FOREST.C which contains the tables as they will be seen by the interpreter, the file FNCTAB.C which maps functions referenced by an index from within the tables to the actual function entry points, the file INTOPS.HDR which contains the operators (table names) defined as numbers, and the file MODES.HDR which contains miscellaneous definitions used by various odd parts of the compiler. TRANS uses several temporary files: SPVAL.C, ROOTS.C, ATTRIB.C, CODE.C, and CODCST.C are used to build up array declarations for the arrays spval[], roots[], attrib[], codtab[], and codcst[] that appear at the end of FOREST.C. ATRDEF.C collects certain opcode cost definitions and all flag definitions that are then appended to INTOPS.HDR.

A small example has been run through TRANS and the output files INTOPS.HDR, MODES.HDR, FNCTAB.C, and FOREST.C included in this document. COSTS.TRN plus an initial segment of TABLES.DAT (as indicated in TABLES.DAT by the placement of a comment to that effect) were run through the TRANS program to produce the output shown.

Annotation on the TRANS program:

126-129 Main loop.

132-140 Statistics output.

143-164 Initializations.

166-176 Process COSTS.TRN.

178-186 Prepare to turn "[" from table name into flag FBINOP, "<" into FUNOP, and "^" into FLEAF.

199-206 Digest mode declarations.

208-447 The bulk of the program is a recursive descent parser and translator for the code tables themselves.

214-227 Loop over multiple table names digesting names. In 217-223 define once and set flags for leaf (219), unary operator (220), or binary operator (221).

228-239 Get flags, store as bits in attrib.

240 Get table contents.

243-252 Get a sequence of alternatives.

254 Readlist digests the contents of one alternative from the opening "{" to the mathcing "}".

258-263 Handle the definition of a label; see 271.

264-267 Handle a reference to a label.

274-290 This loop handles the table name(s) and flags. The loop is over all the table names. (Multiple names for each table are provided as a convenience.)

292 Go get key.

293 Go get code words (machine instructions).

294-296 Go get more alternatives or get returns.

298 Get final "}".

307-311 Get key indicating recursive call of interpreter on left, right, or same subtree of parse tree.

312-313 Get key indicating call of function (from FUNCS.C via FNCTAB).

315-323 Get arguments for key.

334 Main loop getting code words.

340-351 Get opcode.

357-363 Get 3 instruction fields: mod, ac, m.

367-373 Collect statistics on opcode usage.

378-384 Get function for key.

387-394 Get mode.

396-413 Evaluate a reference to a label.

415-437 Get returns.

585-608 Print out a table into FOREST.C.

615-622 Append temporary files to appropriate output files.

628-684 Maintain symbol table.

686-722 Debugging.

THE "CODE" ROUTINE

The "code" routine embodies the code table interpreter. It searches the tables, calling functions as indicated in the tables, calling itself recursively when indicated in the tables (by L, R, or S) and doing the necessary bookkeeping. It is called with six arguments: (1) argp: pointer to parse tree node to translate; (2) argt: pointer to code table to use; (3) mode: evaluation mode; (4) atp: pointer to top of stack of arguments; (5) maxcost: cost to beat; (6) topflag: nonzero if this is the first call of code for this parse tree node. The code routine returns 0 for failure to find a translation in the

given evaluation mode, and 1 for success. On a successful return it also sets up: (1) rval1 and rval2: two-word returned value (taken from argument to return in table); (2) rtree: translation tree containing translation and related information; (3) rcost: cost of the translation.

Annotation on the code routine:

35 Routine called to translate one expression. Most of the body of this routine is irrelevant and has been omitted.

39 Go read in a parse tree from the intermediate code file. (The "treein" routine is not included.)

41 Call "code" for translation.

44 Print out translation into output file.

49-225 Recursive code table interpreter "code".

60-61 Check for reentry of parse tree node on same evaluation mode and fail if so.

63-65 Statistics collection.

68-69 The gbxxx variables are copies of code’s local variables being made available to routines in FUNCS.C.

71 Begin keeping track of costs, best costs, etc.

75 Loop over alternatives at one level.

77 Skip if no returns on this mode.

80-81 Same as 68 - 69.

83-86 For function calls and recursive calls of code (L, R, S) fetch arguments and push onto global argument stack apstack.

88 Switch statement for 4 cases: L, R, and S recursion and function call.

98 Recursive call of code for L, R, S subtree of parse tree as called for in table.

108 Call function in FUNCS.

117-121 Account for cost of recursive subevaluation

122-135 Build node for return tree containing translation including value and machine code, push it onto the global argument stack.

136-152 If returns are next, process them.

155-167 Otherwise, a nested set of alternatives is next, process them recursively.

161 Similar to 117 - 121.

162-167 Store translation values and trees.

169-180 Select best translation.

182-183 Failure to translate for whatever reason.

187-194 Implement COMMUTE flag, go try again.

195-200 Using universal table, go try again.

202-207 Undo changes made is 187-194.

209-215 If no translation found by this time, return failure.

216-224 Successful return.

228-237 Evaluate a reference to an argument (@, %, or spval).

238-246 Push argument onto global argument stack.

254-257 Calculate code cost.

259-267 Disassemble parse tree and deallocate its nodes.

269-274 Disassemble return tree and deallocate its nodes.

276-294 Copy code in code table to return tree, evaluating operands and calculating code cost.

296-304 Evaluate operands in code words for gencode.

309-343 Recursively print code from a complete return tree to output file.

346-400 Print operands for emitcode.

402-414 Print numbers for lstring.

EXAMPLES

The following three examples illustrate the code generation process. In all three cases, the same results were obtained in student mode as in full-search mode.

Example 1

C source code: extern int i; i += -3;

Parse tree: [APII ^DI:i ^CI:-3
(prefix notation)

Evaluation modes used: E WJ VP
(in same order as corresponding parse tree nodes)

```
Returned values:          0   i   R2
(in same order as corresponding parse tree nodes)

Generated code:           HRROI R2, -3
                          ADDM R2, i


Example 2

C source code:            extern in i, b[10];    ++b[i];

Parse tree:               [PIPI  <STP  [PLPI  <ADI  ^DI:b  ^DI:i  ^CI:1

Evaluation modes:         E    WJ    VJ    VJ    WJ    VP    VO

Returned values:          0  b(R2)  b(R2)  b  b  R2  0

Generated code:           MOVE R2, i
                          AOS b(R2)

Example 3

C source code:            extern int *b[10];   ++*b[2];

Parse tree:               [PIII  <STP  <STP  <ADI  ^DI:b+2  ^CI:1

Evaluation modes used:    E    WJ    VJ    VJ    WJ    VO

Returned values:          0  @b+2  b+2  b+2  b+2  0

Generated code:           AOS @b+2
/*******        FILE:  COSTS.TRN        *******/
/*******                                *******/
! 70    ADJSP
!265    LSH
!285    ASH
!300    ASHC
!300    LSHC
!325    ROTC*
!126    MOVE*,MOVS*
!126    HRR*,HRRO*,HRRZ*,HRRE*,HRL*,HRLO*,HRLZ*,HRLE*,HLR*,HLRO*,HLRZ*,HLRE*,HLL*,HLLO*,HLLZ
*,HLLE*
!300    EXCH*
! 45    MOVEI,MOVSI,HRRI,HRRZI,HRROI,HRLI,HRLZI,HRLOI,HLRI,HLRZI,HLROI,HLLI,HLLZI,HLLOI
!106    MOVEM,MOVSM,HRRZM,HRROM,HRLZM,HRLOM,HLRZM,HLROM,HLLZM,HLLOM
!161    MOVES,MOVSS,HRRS,HRRZS,HRROS,HRLS,HRLZS,HRLOS,HLRS,HLRZS,HLROS,HLLS,HLLZS,HLLOS
!172    HRRM,HRLM,HLRM,HLLM
!132    MOVN*,MOVM*
! 51    MOVNI,MOVMI
!112    MOVNM,MOVMM
!167    MOVNS,MOVMS
!190    IBP
!300    ADJBP
!480    LDB
!406    DPB
!547    ILDB
!473    IDPB
!194    PUSH
!216    POP
!112    PUSHJ
!143    POPJ
! 57    AOBJN,AOBJP
```

```
! 62    CAI*,CAM*,AOJ,SOJ
! 56    JUMP
!137    SKIP
!178    AOS,SOS
! 45    JSP
! 34    JRST*,JEN,PORTAL
! 45    JRSTF
! 34    JFCL
!200    JFFO
! 34    XCT
! 45    TLN,TRZ
! 34    TRN
! 56    TLNA,TLZ,TLC,TRZA,TRCA,TRO,TROA
! 62    TLNE,TLNN*,TRZE,TRZN,TRCE,TRCN,TROE,TRON,TRNE,TRNN
! 67    TLZA,TLCA,TLO,TLOA
! 73    TLZE,TLZN,TLCE,TLCN,TLOE,TLON
!115    TDN
!132    TDNE,TDNN,TDO
!126    TDNA,TDZ,TDC,TSN
!137    TDZA,TDCA,TDOA,TSNA,TSZ,TSC
!143    TDZE,TDZN,TDCE,TDOE,TDON,TSNE,TSNN
!148    TSZA,TSCA,TSO,TSOA
!154    TSZE,TSZN,TSCN,TSOE,TSON
! 45    SETZ,SETZI,SETO,SETOI,SETA,SETAI,SETCA,SETCAI
! 95    SETZM,SETZB,SETOM,SETOB,SETAM,SETAB,SETCAM,SETCAB
!126    AND,ANDCA,ANDCM,ANDCB,SETM,SETCM,XOR,EQV
! 45    ANDI,ANDCAI,ANDCMI,ANDCBI,SETMI,SETCMI,XORI,EQVI
!161    ANDM,ANDB,ANDCAM,ANDCAB,ANDCMM,ANDCMB,ANDCBM,ANDCBB,SETMM,SETMB,SETCMM,SETCMB,XORM,X
ORB,EQVM,EQVB
!137    OR,ORCA,ORCM,ORCB,IOR
! 56    ORI,IORI,ORCAI,ORCMI,ORCBI
!172    ORM,ORB,IORM,IORB,ORCAM,ORCAB,ORCMM,ORCMB,ORCBM,ORCBB
!132    ADD,SUB
! 51    ADDI,SUBI
!167    ADDM,ADDB,SUBM,SUBB
!538    MUL
!457    MULI
!551    MULM
!562    MULB
!492    IMUL
!411    IMULI
!527    IMULM,IMULB
!830    DIV
!749    DIVI
!843    DIVM
!854    DIVB
!830    IDIV
!740    IDIVI
!843    IDIVM
!854    IDIVB
!173    DMOVE
!185    DMOVEM
!208    DMOVN
!231    DMOVNM
!110    FSC
!251    FIX,FIXR
!408    FLTR
!150    DFN
!288    UFA
!435    FADR
!365    FADRI
!470    FADRM,FADRB
!452    FSBR
!388    FSBRI
!488    FSBRM,FSBRB
```

```
!427    FMPR
!327    FMPRI
!462    FMPRM,FMPRB
!772    FDVR
!702    FDVRI
!807    FDVRM,FDVRB
!432    FAD
!466    FADL
!467    FADM,FADB
!449    FSB
!483    FSBL
!485    FSBM,FSBB
!424    FMP
!458    FMPL
!459    FMPM,FMPB
!743    FDV
!814    FDVL
!778    FDVM,FDVB
!479    DFAD
!489    DFSB
!874    DFMP
!999    DFDV
/*******        FILE:  TABLES.DAT       *******/
/*******                                *******/

/*******        DEFINED CONSTANTS FOR CODE TABLES       *******/

#define LABEL 01
#define N 02
#define A 03
#define LT 04
#define LE 05
#define EQ 06
#define NE 07
#define GT 010
#define GE 011


#define MA_PRIB 1
#define MA_SPRIB 2
#define MA_RELAT 3

#define IF IFFUNC
#define ELSE ELSEFUNC

#define NOTRANS 0       /* also defined in FUNCS.C */
#define NOJUMPENT 1
#define NOVAENT 2
#define NOVPENT 3
#define KL10 4

#define NOSPLIT 0
#define SPLITADDR 1
#define SPLITPS 2

#define ISSKIP 1
#define ISJUMP 2

#define LHOK 0
#define LHBAD 1


/*******        MODE DECLARATIONS       *******/

E,      /* side Effect, must be listed first */
```

```
MA,     /* Mode is given as Argument */
VJ,     /* Value as a pointer (Jump address) */
VA,     /* Value in a given Accumulator */
VI,     /* Value as an Immediate reference */
VH,     /* Value as a Half word (immediate) on the left */
VP,     /* Value in any ac, will be Preserved */
VO,     /* Value is constant One */
VM,     /* Value required as a Memory reference */
CI,     /* Complement of value as an Immediate */
CM,     /* Complement of value as a Memory reference */
CP,     /* Complement of value in any AC, will be Preserved */
CA,     /* Complement of value in a given Accumulator */
NI,     /* Negative of value an an Immediate */
NM,     /* Negative of value an a Memory reference */
NP,     /* Negative of value in any AC, will be Preserved */
NA,     /* Negative of value in a given Accumulator */
J,      /* Evaluate to Jump on given condition */
K,      /* Evaluate to sKip on given condition */
KA,     /* sKip on condition and load given Ac */
WJ,     /* address (Where) as a pointer (Jump address) */
WI,     /* Where Immediate */
WP,     /* Where Preserved ac */
WM,     /* Where Memory reference */
WA,     /* Where Ac */
AE,     /* Argument Evaluation mode */
AC,     /* Argument Counting mode */
VZ,     /* Value is constant Zero */
VN      /* Value is constant Negative one */


/*******        UNIVERSAL TABLE         *******/

[UNIV   (UNIVERSAL,NOUNIV)      /* universal table */

{ ACSIZE { SVA(%1) VP(%0) } }

{ SVP VM(%0)}

{ SVI { G(@1) .MOVEI..%0.%1. VA(%0) } }
{ SVM { G(@1) { ACSIZE
        { IDENT(%1,-1) .MOVE..%2.%3. F(%3) VA(%2) }
        { IDENT(%1,-2) .DMOVE..%2.%3. F2(%3) VA(%2) }
} } }

{ SVM { AT(%1,@1) VJ(%0) } }

{ ACSIZE { SWA(%1) WP(%0) } }

{ SWP WM(%0) }

{ SWI { G(@1) .MOVEI..%0.%1. WA(%0) } }
{ SWM { G(@1) .MOVE..%0.%1. F(%1) WA(%0) } }

{ SWM { AT(%1,@1) WJ(%0) } }

{ DIFFER(@3,ISSKIP) { INVERT(@1) { SK(%1,@3) { NULL(@2) .JRST...%0. J } } } }

{ DIFFER(@2,ISJUMP) { NEWLABEL { INCR(%1) { SJ(@1,%1,@2) ..LABEL..%2. K } } } }

{ ACSIZE { IDENT(%1,-1)
{ IF(NOTRANS)
        { SVP
                { NULL(@1) { NULL(@2) .JUMP.%1.%2.%0. F(%2) J } }
                /* { DIFFER(@2,ISJUMP) { NULL(@1) { NEWLABEL { INCR(%1) .JUMP.%2.%4.%0. ..LA
BEL..%1. F(%4) K } } } } */
```

```
        }
        { SVM
                { NULL(@1) .SKIP.%0..%1. F(%1) K }
                /* { DIFFER(@2,ISSKIP) { INVERT(@1) { NULL(@2) .SKIP.%1..%3. .JRST...%0. F(%
3) J } } } */
        }
}

{ SVM { NULL(@1) { NULL(@2) .SKIP.%1.%0.%2. F(%2) KA(%0) } } }
{ SVA(@2) { NEWLABEL { INCR(%1) { NULL(@1) .JUMP.%0.%3.%1. ..LABEL..%2. KA(%3) } } } }
} }

{ NOEFFECT(0) E }
{ NOEFFECT(1) { le E } }
{ NOEFFECT(2) { le { RE E } } }


/*******        LEAVES (OPERANDS)       *******/

^CI, ^CP, ^CF   (NOUNIV)        /* constants, one-word */
{ X
        { ISZ(%1)
                { NULL VZ }
                { IDENT(@1,NE) J K }
                { IDENT(@1,EQ)
                        { NULL(@2) .JRST...%0. J }
                        { NULL .SKIP.A... K } } }
        { IF(NOTRANS)   /* equiv to NOTZ(%1) */
                { IDENT(@1,EQ) J K }
                { IDENT(@1,NE)
                        { NULL(@2) .JRST...%0. J }
                        { NULL .SKIP.A... K } } }
        { ISO(%1) VO }
        { ISN(%1) VN }
        { HLZ(%1)       /* "half on left is zero" */
                { NULL VI(%1) VJ(%1) }
                { G(@1) .MOVEI..%0.%1. VA(%0) }
                { G(-1) .MOVEI..%0.%1. VP(%0) }
                { G(@1) .MOVNI..%0.%1. NA(%0) }
                { G(-1) .MOVNI..%0.%1. NP(%0) } }
        { HRZ(%1)
                { NULL VH(%1) }
                { G(@1) .HRLZI..%0.%1. VA(%0) }
                { G(-1) .HRLZI..%0.%1. VP(%0) }}
        { HLO(%1)
                { IDENT(@1,LHBAD) VJ(%1) }
                        /* this was for LSII:  LHBAD implies please discard l.h. if possible
 */
                        /* LHOK really shouldn't reach here */
                { G(@1) .HRROI..%0.%1. VA(%0) }
                { G(-1) .HRROI..%0.%1. VP(%0) }
                { CMPL(%2) CI(%0) }
                { NEGATE(%2) { HLZ(%1) NI(%1) } } }
        { HRO(%1)
                { G(@1) .HRLOI..%0.%1. VA(%0) }
                { G(-1) .HRLOI..%0.%1. VP(%0) } }
        { MAKLIT(%1)
                { NULL VM(%1) }
                { G(@1) .MOVE..%0.%1. VA(%0) }
                { G(-1) .MOVE..%0.%1. VP(%0) } } }

^CD     (NOUNIV,TWOACS)         /* constants, two-word */

{ X { MAKLIT(%1)
        { NULL VM(%1) }
```

```
        { G(@1) .DMOVE..%0.%1. VA(%0) }
        { G(-2) .DMOVE..%0.%1. VP(%0) }
        { G(@1) .DMOVN..%0.%1. NA(%0) }
        { G(-2) .DMOVN..%0.%1. NP(%0) }
} }

/*******        the example run through TRANS           *******/
/*******            stops at this point                 *******/

^DI, ^DP, ^DF, ^LI, ^LP, ^LF    (NOUNIV)        /* data, label */
{ X
        { AT(%1,@1) VJ(%0) }
        { NULL VM(%1) WI(%1) WJ(%1) }
        { G(@1)
                { NULL .MOVE..%1.%2. VA(%1) }
                { NULL .MOVN..%1.%2. NA(%1) }
                { NULL .MOVEI..%1.%2. WA(%1) } }
        { G(-1) .MOVE..%0.%1.
                { NULL VP(%1) }
                { AT(%1,@1) VJ(%0) } }
        { G(-1) .MOVEI..%0.%1. WP(%0) WM(%0) }
        { NULL(@1) .SKIP.%0..%1. K }
        { NULL(@1) { G(@2) .SKIP.%1.%0.%2. KA(%0) } }
        { INVERT(@1) .SKIP.%0..%1.
                { NULL(@2) .JRST...%0. J } }
}

^DC, ^LC, ^RC           (NOUNIV)        /* data, label, register characters */
{ X
        { MAKLIT(%1)
                { G(@1) .LDB..%0.%1. VA(%0) }
                { G(@2) .LDB..%0.%1.
                        { NULL(@1) { NEWLABEL { INCR(%1) .JUMP.%2.%3.%0. ..LABEL..%1. KA(%3)
 } } } } }
                { G(-1) .LDB..%0.%1.
                        { NULL VP(%1) VM(%1) }
                        { AT(%1,LHOK) VJ(%0) }
                        { NULL(@1) { NEWLABEL { INCR(%1) .JUMP.%2.%3.%0. ..LABEL..%1. F(%3)
K } } } }
                        { NULL(@1) { NULL(@2) .JUMP.%1.%2.%0. F(%2) J } } }
        }
        { BPSPLIT(NOSPLIT,%1) { MAKLIT(%2)
                { NULL WM(%1) }
                { G(@1) .MOVE..%0.%1. WA(%0) }
                { G(-1) .MOVE..%0.%1. WP(%0) }
        } }
        { BPSPLIT(SPLITADDR,%1) { BPSPLIT(SPLITPS,%2)
                { G(@1) .MOVEI..%0.%2. .HRLI..%0.%1. WA(%0) }
                { G(-1) .MOVEI..%0.%2. .HRLI..%0.%1. WP(%0) WM(%0) }
        } }
}

^DD, ^LD        (NOUNIV,TWOACS)         /* data, label doubles */

{ X
        { NULL VM(%1) WI(%1) WJ(%1) }
        { G(@1) .DMOVE..%0.%1. VA(%0) }
        { G(-2) .DMOVE..%0.%1. VP(%0) }
        { G(@1) .DMOVN..%0.%1. NA(%0) }
        { G(-2) .DMOVN..%0.%1. NP(%0) }
        { G(@1) .MOVEI..%0.%1. WA(%0) }
        { G(-1) .MOVEI..%0.%1. WP(%0) WM(%0) }
        { NULL(@1) .SKIP.%0..%1. K }
        { INVERT(@1) .SKIP.%0..%1. { NULL(@2) .JRST...%0. J } }
}
```

```
^RI, ^RF, ^RP   (NOUNIV)        /* register one-words */
{ X
        { AT(%1,@1) VJ(%0) }
        { NULL VP(%1) WJ(%1) VM(%1) WI(%1) }
        { NULL(@1)
                { NULL(@2) .JUMP.%1.%2.%0. J }
                { G(@2) .SKIP.%1.%0.%2. KA(%0) }
                { NEWLABEL
                        { INCR(%1) .JUMP.%2.%3.%0. ..LABEL..%1. K }
                }
        }
        { G(@1)
                { NULL .MOVE..%1.%2. VA(%1) }
                { NULL .MOVN..%1.%2. NA(%1) }
                { NULL .MOVEI..%1.%2. WA(%1) } }
        { G(-1)
                { NULL .MOVEI..%1.%2. WP(%1) WM(%1) } } }

^RD             (NOUNIV,TWOACS)         /* register doubles */

{ X
        { NULL VP(%1) WJ(%1) VM(%1) WI(%1) }
        { G(@1) .DMOVE..%0.%1. VA(%0) }
        { G(@1) .DMOVN..%0.%1. NA(%0) }
        { G(-2) .DMOVN..%0.%1. NP(%0) }
        { G(@1) .MOVEI..%0.%1. WA(%0) }
        { G(-1) .MOVEI..%0.%1. WP(%0) }
        { NULL(@1) { NULL(@2) .JUMP.%1.%2.%0. J } }
        { NEWLABEL { INCR(%1) { SJ(@1,%1,@2) K } } }
}

/*      CONVERSIONS     */

<CIC                    /* character-to-integer */

{ IF(NOVPENT) { LVA(@1) VA(%0) } }
{ IF(NOVAENT) { LVP VP(%0) } }
{ le E }        /* l.c. as in CMII */

<IDI    (TWOACS)

{ LVA(@1) { INCR(%1) { G(-2) { INCR(%1)
        { NEWLABEL
        .SETZ..%3.. .MOVM..%2.%4. .JFFO..%2.%0. ..LABEL..%0.
        { AT(%2,LHOK) { MGI(%1,233) { MGI(%2,-11)
                .ASHC..%7.%0. .MOVNI..%4.%0. .FSC..%7.%1. .TLZ..%6.400000.
                F2(%5) VA(%7) } } }
        }
} } } }

<DID

{ GETFOL(@1)
    { DIFFER(%1,-3) { LVA(%2) { INCR(%1) { G(-1) { AT(%1,LHOK) { MGI(%1,-43)
        .MOVM..%2.%4. .TLCN..%2.200000. .TLO..%2.400000.
        .ASH..%2.-33. .ASHC..%4.10. .ASH..%4.%0.
                { FNEW(@1,%4) F(%3) VA(%5) } } } } } } }
    { IDENT(%1,-3) { LVA(%2) { INCR(%1) { G(@1) { AT(%1,LHOK) { MGI(%1,-43)
        .MOVM..%2.%4. .TLCN..%2.200000. .TLO..%2.400000.
        .ASH..%2.-33. .ASHC..%4.10. .ASH..%4.%0.
                .MOVE..%2.%4. F2(%4) VA(%2) } } } } } }
}

<IFI
```

```
{ LVZ
        { NULL VZ }
        { G(@1) .SETZ..%0.. VA(%0) } }
{ LVM
        { G(@1) .FLTR..%0.%1. F(%1) VA(%0) } }

<JFJ

{ LVA(@1) .LSH..%0.-1. .FLTR..%0.%0. .FSC..%0.1. VA(%0) }

<FIF

{ LVM { G(@1) .FIX..%0.%1. F(%1) VA(%0) } }

<FJF

{ LVA(@1) .FSC..%0.-1. .FIX..%0.%0. .LSH..%0.1. VA(%0) }

<FDF    (TWOACS)

{ LVZ
        { NULL VZ }
        { G(@1) { INCR(%1) .SETZB..%0.%1. VA(%1) } }
}
{ LVA(@1) { INCR(%1) .SETZ..%0.. VA(%1) } }

<DFD

{ LVM
        { ISAC(%1) { INCR(%2) F(%0) VP(%2) } }
        { NOTAC(%1) VM(%1) }
}


/*******        FUNCTION CALLING AND COMMA      *******/

[FNII           (USEONE)        /* call of function-returning-integer */

{ LVJ(LHOK) { RAE(-1) .PUSHJ..17.%1. { F(%2)
        { FCINFO(0,%2) { FCINFO(1,%3) .TRN..%1.%0.
                { NULL E }
                { G1(@1,1) VA(%0) }
                { G(@1) .MOVE..%0.1. VA(%0) }
        } }
} } }

[FNFI           (USEONE)        /* call of function-returning-float */

{ LVJ(LHOK) { RAE(-1) .PUSHJ..17.%1. { F(%2)
        { FCINFO(0,%2) { FCINFO(1,%3) .TRN..%1.%0.
                { NULL E }
                { G1(@1,0) VA(%0) }
                { G(@1) .MOVE..%0.0. VA(%0) }
        } }
} } }

[FNLI           (USEONE,TWOACS) /* call of function-returning-double */

{ LVJ(LHOK) { RAE(-1) .PUSHJ..17.%1. { F(%2)
        { FCINFO(0,%2) { FCINFO(1,%3) .TRN..%1.%0.
                { NULL E }
                { G1(@1,0) VA(%0) }
                { G(@1) .DMOVE..%0.0. VA(%0) }
        } }
```

```
} } }

<FNI            (USEONE)        /* call of argumentless function-returning-integer */
{ LVJ(LHOK) .PUSHJ..17.%0. { F(%1)
        { FCINFO(0,0) { FCINFO(1,0) .TRN..%1.%0.
                { NULL E }
                { G1(@1,1) VA(%0) }
                { G(@1) .MOVE..%0.1. VA(%0) }
        } }
} }

<FNF            (USEONE)        /* call of argumentless function-returning-float */
{ LVJ(LHOK) .PUSHJ..17.%0. { F(%1)
        { FCINFO(0,0) { FCINFO(1,0) .TRN..%1.%0.
                { NULL E }
                {G1(@1,0) VA(%0) }
                { G(@1) .MOVE..%0.0. VA(%0) }
        } }
} }

<FNL            (USEONE,TWOACS) /* call of argumentless function-returning-double */
{ LVJ(LHOK) .PUSHJ..17.%0. { F(%1)
        { FCINFO(0,0) { FCINFO(1,0) .TRN..%1.%0.
                { NULL E }
                { G1(@1,0) VA(%0) }
                { G(@1) .DMOVE..%0.0. VA(%0) }
        } }
} }

<RXI            (USEONE)        /* return-from-function, an integer */
{ G1(-1,1) { LVA(%1) F(%1) E } }
{ ELSE(1) { LVM .MOVE..1.%0. F(%0) E } }

<RXF            (USEONE)        /* return-from-function, a float */
{ G1(-1,0) { LVA(%1) F(%1) E } }
{ ELSE(1) { LVM .MOVE..0.%0. F(%0) E } }

<RXL            (USEONE,TWOACS) /* return-from-function, a double */
{ G1(-2,0) { LVA(%1) F2(%1) E } }
{ ELSE(1) { LVM .DMOVE..0.%0. F2(%0) E } }

<SWI            (NOUNIV)        /* switch expression */
{ LVM .MOVE..0.%0. F(%0) E }

[CMII           (NOUNIV)        /* comma operator */
:TOP
{ le    { RE E }  /* 'le' must be lower case -- else it's <= !! */
        { RMA(@1) MA(%0) }
        { RVJ(@1) VJ(%0) }
        { RVA(@1) VA(%0) }
        { RVI VI(%0) }
        { RVH VH(%0) }
        { RVP VP(%0) }
        { RVO VO }
        { RVZ VZ }
        { RVN VN }
        { RVM VM(%0) }
        { RCI CI(%0) }
        { RCM CM(%0) }
        { RCP CP(%0) }
        { RCA(@1) CA(%0) }
        { RJ(@1,@2,@3) J }
        { RK(@1,@2) K }
        { RKA(@1,@2) KA }
        { RWJ(@1) WJ(%0) }
```

```
        { RWI WI(%0) }
        { RWP WP(%0) }
        { RWM WM(%0) }
        { RWA(@1) WA(%0) }
}

[CMLI           (NOUNIV)        /* comma of double with int */
{ NULL AE }     /* this line is believed to be obsolete */
*[CMII:TOP      /* and this line just uses the entire CMII table */

[CMIL           (NOUNIV)        /* comma of int with double */
{ NULL AE }
*[CMII:TOP

[CMLL           (NOUNIV)        /* comma of double with double */
{ NULL AE }
*[CMII:TOP

[FCIS           (NOUNIV)        /* comma between function arguments, "safe" */
                                /* to left is operand, to right rest of list */
{ INCR(@1) { MAKARG(%1)
        { LVZ .SETZM...%1. { RAE(%3) { INCR(%1) AE(%0) } } }
        { LVN .SETOM...%1. { RAE(%3) { INCR(%1) AE(%0) } } }
        { ELSE(2) { LVP .MOVEM..%0.%2. { F(%1) { RAE(%5) { INCR(%1) AE(%0) } } } } }
} }

[FCIU           (NOUNIV)        /* comma between function arguments, "unsafe" */
{ INCR(@1) { MAKARG(%1)
        { LVZ { RAE(%3) .SETZM...%2. { INCR(%1) AE(%0) } } }
        { LVN { RAE(%3) .SETOM...%2. { INCR(%1) AE(%0) } } }
        { ELSE(2) { LVA(-1) { RAE(%4) .MOVEM..%1.%3. { INCR(%1) F(%2) AE(%0) } } } }
} }

[FCLS           (NOUNIV)        /* left operand is double */
{ INCR(@1) { INCR(%1) { MAKARG(%1)
        { LVP .DMOVEM..%0.%1. { F2(%1) { RAE(%4) { INCR(%1) AE(%0) } } } }
} } }

[FCLU           (NOUNIV)        /* left operand is double */
{ INCR(@1) { INCR(%1) { MAKARG(%1)
        { LVA(-2) { RAE(%3) .DMOVEM..%1.%2. { INCR(%1) F2(%2) AE(%0) } } }
} } }


<FCI            (NOUNIV)        /* function argument, no more list */

{ INCR(@1) { INCR(%1) { MAKARG(%2)
        { LVZ .SETZM...%1. AE(1) }
        { LVN .SETOM...%1. AE(1) }
        { ELSE(2)
                { LVP .MOVEM..%0.%2. F(%0) AE(1) } }
} } }

<FCL            (NOUNIV)        /* last function argument, a double */
{ INCR(@1) { INCR(%1) { INCR(%1) { MAKARG(%2)
        { LVP .DMOVEM..%0.%1. F2(%0) AE(1) }
} } } }

/*      ASSIGNMENT, STAR, AND ADDRESS   */

[AGII           (LSTL)          /* assignment of one-word values */

{ LWJ(LHOK)
    { ISAC(%1)
        { RVI .MOVEI..%2.%0. E VP(%2) } /* !! */
```

```
        { RNI .MOVNI..%2.%0. E VP(%2) }
        { RVM .MOVE..%2.%0. F(%0) E VP(%2) }    /* !! */
                /* can't use RVA because of: reg = mem * reg */
        { RNM .MOVN..%2.%0. F(%0) E VP(%2) }
        { RKA(@1,%2) K }

    }
    { NOTAC(%1)
        { RVZ
            { NULL .SETZM...%3. F(%3) VZ E }
            { G(@1) .SETZB..%0.%3. F(%3) VA(%0) } }
        { RVN
            { NULL .SETOM...%3. F(%3) VN E }
            { G(@1) .SETOB..%0.%3. F(%3) VA(%0) } }
        { IF(NOVPENT) { RVA(@1) .MOVEM..%0.%3. F(%3) VA(%0) } }
        { IF(NOVAENT) { RVP .MOVEM..%0.%3. F(%3) VP(%0) F(%0) E } }
        { IF(NOVPENT) { RNA(@1) .MOVNM..%0.%3. F(%3) NA(%0) } }
        { IF(NOVAENT) { RNP .MOVNM..%0.%3. F(%3) NP(%0) F(%0) E } }
    }
}

[AGLL           (TWOACS,LSTL)   /* assignment of two-word values */

{ LWJ(LHOK)
    { ISAC(%1)
        { RVM .DMOVE..%2.%0. F2(%0) E VP(%2) }
        { RNM .DMOVN..%2.%0. F2(%0) E VP(%2) }
    }
    { NOTAC(%1)
        { IF(NOVPENT) { RVA(@1) .DMOVEM..%0.%3. F(%3) VA(%0) } }
        { IF(NOVAENT) { RVP .DMOVEM..%0.%3. F(%3) VP(%0) F2(%0) E } }
        { IF(NOVPENT) { RNA(@1) .DMOVNM..%0.%3. F(%3) NA(%0) } }
        { IF(NOVAENT) { RNP .DMOVNM..%0.%3. F(%3) NP(%0) F2(%0) E } }
} }

<STP            (NOUNIV)        /* star (contents) of pointer */

{ LVJ(LHOK) WJ(%0) }
{ LVI WI(%0) }
{ LVA(@1) WA(%0) }
{ LVP WP(%0) }
{  LVM WM(%0) }

{ LVJ(LHOK) VM(%0) }
{ LVJ(LHOK) { AT(%1,LHOK) VJ(%0) } }
{ LVJ(LHOK)
        { G(@1) .MOVE..%0.%1. F(%1) VA(%0) }
        { G(-1) .MOVE..%0.%1.
                { NULL F(%2) VM(%1) VP(%1) }
                { AT(%1,LHOK) F(%2) VJ(%0) } } }
{ LVA(@1) { AT(%1,LHOK) .MOVE..%1.%0. VA(%1) } }
{ LVA(-1) { AT(%1,LHOK) .MOVE..%1.%0. VM(%1) VP(%1) } }
:AEC    /* convert jump and skip modes for non-users of universal table */
{ SVP { NULL(@1) { NULL(@2) .JUMP.%1.%2.%0. F(%2) J } } }
{ SVM
        { NULL(@1) .SKIP.%0..%1. F(%1) K }
        { NULL(@1) { G(@2) .SKIP.%1.%0.%2. F(%2) KA(%0) } } }

<STT            (NOUNIV)        /* star of pointer to double */

{ LVJ(LHOK) WJ(%0) }
        { LVI WI(%0) }
        { LVA(@1) WA(%0) }
        { LVP WP(%0) }
        { LVM WM(%0) }
```

```
        { LVJ(LHOK) VM(%0) }
        { LVJ(LHOK) { AT(%1,LHOK) VJ(%0) } }
        { LVJ(LHOK)
                { G(@1) .DMOVE..%0.%1. F(%1) VA(%0) }
                { G(-1) .DMOVE..%0.%1. F(%1) VM(%0) VP(%0) }
        { G(@1) { LVA(%1) { AT(%1,LHOK) .DMOVE..%1.%0. VA(%1) } } }
        { G(-2) { LVA(%1) { AT(%1,LHOK) .DMOVE..%1.%0. VM(%1) VP(%1) } } }
}

<ADI,   <ADP,   <ADC            (NOUNIV,LSTL)   /* address operator */

{ LWM VM(%0) }
{ LWJ(LHOK) VJ(%0) }
{ LWI VI(%0) }
{ LWA(@1) VA(%0) }
{ LWP VP(%0) }
*<STP:AEC

[AGCI           (LSTL)          /* assignment  to a character */
{ LMA(MA_SPRIB)
        { RVP .IDPB..%0.%1. F(%1) VP(%0) F(%0) E }
        { RVA(@1) .IDPB..%0.%1. F(%1) VA(%0) } }

{ LWM
        { RVP .DPB..%0.%1. F(%1) VP(%0) F(%0) E }
        { RVA(@1) .DPB..%0.%1. F(%1) VA(%0) } }


/*******        LOGICALS AND COMPARISONS        *******/

[ANII                   /* && operator */

{ IDENT(@1,EQ) { LJ(EQ,@2,ISJUMP) { RJ(EQ,@2,@3) J } } }

{ IDENT(@1,NE) { NEWLABEL { LJ(EQ,%1,ISJUMP) { RJ(NE,@2,@3) ..LABEL..%2. J } } } }

:LOG/* convert skip or jump to a value */
{ IF(NOJUMPENT) { G(@1) { SK(NE,ISSKIP) .TDZA..%1.%1. .MOVEI..%1.1. VA(%1) } } }

[ORII                   /* || operator */

{ IDENT(@1,NE) { LJ(NE,@2,ISJUMP) { RJ(NE,@2,@3) J } } }

{ IDENT(@1,EQ) { NEWLABEL { LJ(NE,%1,ISJUMP) { RJ(EQ,@2,@3) ..LABEL..%2. J } } } }

*[ANII:LOG

<NGI                    /* ! operator */

{ CONDMAP(@1,EQ)
        { LK(%1,@2) K }
        { LJ(%1,@2,@3) J }
}
*[ANII:LOG

[EQII                   /* == operator */

{ CONDMAP(EQ,@1)
        { RVZ
                { DIFFER(@2,ISJUMP) { LK(%3,@2) K } }
                { DIFFER(@3,ISSKIP) { LJ(%3,@2,@3) J } } }
        { LVZ
                { DIFFER(@2,ISJUMP) { RK(%3,@2) K } }
                { DIFFER(@3,ISSKIP) { RJ(%3,@2,@3) J } } }
```

```
        { ELSE(2)
                { LVP
                        { RVI .CAI.%3.%1.%0. F(%1) K }
                        { RVM .CAM.%3.%1.%0. F(%1) F(%0) K } }
                { RVP
                        { LVI .CAI.%3.%1.%0. F(%1) K }
                        { LVM .CAM.%3.%1.%0. F(%1) F(%0) K } }
        }
}
*[ANII:LOG
[EQDD           (COMMUTE)       /* == for doubles */

{ IDENT(@1,NE)
        { LVP { RVM { INCR(%2) { INCR(%2)
                .CAM.NE.%3.%2. .CAM.EQ.%1.%0. F2(%3) F2(%2) K } } } }
}
*[ANII:LOG

[LTII                   /* < operator */

{ RVZ { CONDMAP(LT,@1)
        { DIFFER(@3,ISSKIP) { LJ(%2,@2,@3) J } }
        { DIFFER(@2,ISJUMP) { LK(%2,@2) K } } } }
{ RVO { CONDMAP(LE,@1)
        { DIFFER(@3,ISSKIP) { LJ(%2,@2,@3) J } }
        { DIFFER(@2,ISJUMP) { LK(%2,@2) K } } } }
{ LVZ { CONDMAP(GT,@1)
        { DIFFER(@3,ISSKIP) { RJ(%2,@2,@3) J } }
        { DIFFER(@2,ISJUMP) { RK(%2,@2) K } } } }
{ LVN { CONDMAP(GE,@1)
        { DIFFER(@3,ISSKIP) { RJ(%2,@2,@3) J } }
        { DIFFER(@2,ISJUMP) { RK(%2,@2) K } } } }
{ ELSE(4)
        { CONDMAP(LT,@1) { LVP
                { RVI .CAI.%2.%1.%0. F(%1) K }
                { RVM .CAM.%2.%1.%0. F(%1) F(%0) K } } }
        { CONDMAP(GT,@1) { RVP
                { LVI .CAI.%2.%1.%0. F(%1) K }
                { LVM .CAM.%2.%1.%0. F(%1) F(%0) K } } }
}
*[ANII:LOG

[LTJJ                   /* < for unsigned int's */

{ LVA(-1) { RVM .SUB..%1.%0.
        { IDENT(@1, EQ) { NULL(@2) .JFCL..4.%0. F(%2) F(%3) J } }
        { IDENT(@1, NE) { NEWLABEL { NULL(@2)
                .JFCL..4.%1. .JRST...%0. ..LABEL..%1. F(%3) F(%4) J } } }
} }

[LTDD                   /* < for doubles */

{ RVZ { CONDMAP(LT,@1)
        { LJ(%1,@2,@3) J }
        { LK(%1,@2) K } } }
{ LVZ { CONDMAP(GT,@1)
        { RJ(%1,@2,@3) J }
        { RK(%1,@2) K } } }
{ ELSE(2)
        { CONDMAP(LT,@1) { LVP { RVM { INCR(%2) { INCR(%2)
                { WOEQ(%5) { INVERT(%6) { WEQ(%7)
                        .CAM.%2.%6.%5. .CAM.%1.%4.%3. .CAM.%0.%6.%5.
                F2(%6) F2(%5) K } } }
        } } } } }
        { CONDMAP(GT,@1) { RVP { LVM {  INCR(%2) { INCR(%2)
```

```
                { WOEQ(%5) { INVERT(%6) { WEQ(%7)
                        .CAM.%2.%6.%5. .CAM.%1.%4.%3. .CAM.%0.%6.%5.
                F2(%6) F2(%5) K } } }
        } } } } }
}

[QUII           /* ? of interrogation operator, grouped A ? (B : C) */

{ NEWLABEL { LJ(EQ,%1,ISJUMP)
        { RVA(@1,%2) VA(%0)  }
        { RE(%2) E }
} }

[COII, [COIP, [COPI, [COPP      /* : of interrogation operator */

{ LVA(@1) { NEWLABEL .JRST...%0.
        { NULL(@2) ..LABEL..%0. { RVA(%3) ..LABEL..%2. VA(%0) } } } }
{ le { NEWLABEL .JRST...%0.     /* LE generates <= */
        { NULL(@1) ..LABEL..%0. { RE ..LABEL..%2. E } } } }

<JMI    (LSTL)          /* unconditional jump */
{ LWJ(LHOK) .JRST...%0. E }

[JTII   (LSTL)          /* jump if nonzero (TRUE) */
{ RWJ(LHOK) { LJ(NE,%1,ISJUMP) E } }

[JFII   (LSTL)          /* jump if zero (FALSE) */

{ RWJ(LHOK) { LJ(EQ,%1,ISJUMP) E } }


/*******        INTEGER ARITHMETIC      *******/

[PIII, [PIPI, [APII, [APPI      (LSTL)  /* pre-increment and assign-plus */
:TOP
{ LWJ(LHOK)     /* l-values have 0 left-halves */
        { RVO
                { G(@1) .AOS..%0.%2. F(%2) VA(%0) }
                { NULL .AOS...%2. F(%2) E }
                { NULL(@1) .AOS.%0..%2. F(%2) K }
                { G(@2) { NULL(@1) .AOS.%0.%1.%3. F(%3) KA } } }
        { ISAC(%1)
                { RVO
                        { NULL(@2) { NULL(@1) .AOJ.%0.%4.%1. J } }
                        { NULL .AOJ.N.%3.. VP(%3) E } }
                { RVI .ADDI..%2.%0. VP(%2) E }
                { RVM .ADD..%2.%0. F(%0) VP(%2) E } }
        { NOTAC(%1)
                { RVA(@1) .ADDB..%0.%2. F(%2) VA(%0) }
                { RVP .ADDM..%0.%2. F(%0) F(%2) E } } }


[PDII, [PDPI, [ASII, [ASPI      (LSTL)          /* pre-decrement and assign-minus */
:TOP
{ LWJ(LHOK)
        { RVO
                { G(@1) .SOS..%0.%2. F(%2) VA(%0) }
                { NULL .SOS...%2. VM(%2) F(%2) E }
                { NULL(@1) .SOS.%0..%2. F(%2) K }
                { G(@2) { NULL(@1) .SOS.%0.%1.%3. F(%3) KA } } }
        { ISAC(%1)
                { RVO
                        { NULL(@2) { NULL(@1) .SOJ.%0.%4.%1. J } }
                        { NULL .SOJ.N.%3.. VP(%3) E } }
                { RVI .SUBI..%2.%0. VP(%2) E }
```

```
                { RVM .SUB..%2.%0. F(%0) VP(%2) E } }
        { NOTAC(%1)
                { RVI
                        { G(@1) .MOVNI..%0.%1. .ADDB..%0.%3. F(%3) VA(%0) }
                        { G(-1) .MOVNI..%0.%1. .ADDM..%0.%3. F(%0) F(%3) E } }
                { RVM
                        { G(@1) .MOVN..%0.%1. .ADDB..%0.%3. F(%1) F(%3) VA(%0) }
                        { G(-1) .MOVN..%0.%1. .ADDM..%0.%3. F(%1) F(%0) F(%3) E } }
} }


[SIII, [SIPI    (LSTL)          /* post-increment */

{ EFFECTM *[PIII:TOP }

{ LWJ(LHOK)
        { RVO { G(@1) .MOVE..%0.%2. .AOS...%2. F(%2) VA(%0) } }
        { RVA(@1) .ADD..%0.%2. .EXCH..%0.%2. F(%2) VA(%0) }
}


[SDII, [SDPI    (LSTL)          /* post-decrement */

{ EFFECTM *[PDII:TOP }

{ LWJ(LHOK)
        { RVO { G(@1) .MOVE..%0.%2. .SOS...%2. F(%2) VA(%0) } }
        { RVI { G(@1) .MOVE..%0.%2. .SUBI..%0.%1. .EXCH..%0.%2. F(%2) VA(%0) } }
}


[PLII   (COMMUTE)               /* integer plus integer */

:L0
{ LVA(@1)
        { RVI .ADDI..%1.%0. VA(%1) }
        { RNI .SUBI..%1.%0. VA(%1) }
        { RVM .ADD..%1.%0. F(%0) VA(%1) }
        { RNM .SUB..%1.%0. F(%0) VA(%1) }
}


[PLPI   (COMMUTE)               /* pointer plus integer */

{ LVP { RVJ(LHOK) { AT(%2,LHOK) { MGI(%1,%2) VJ(%0) } } } }
        /* MGI will fail if the VJ has an @, hence LHBAD need not be used */
{ LWI { ISAC(%1) { RVI { AT(%3,LHOK) { MGI(%1,%2) VI(%0) } } } } }
*[PLII:L0

<MII, <MIF                      /* two's complement */

{ LVI NI(%0) }
{ LVM NM(%0) }
{ LVP NP(%0) }
{ LVA(@1) NA(%0) }
{ LNI VI(%0) }
{ LNM VM(%0) }
{ LNP VP(%0) }
{ LNA(@1) VA(%0) }

{ LVM   { G(@1) .MOVN..%0.%1. F(%1) VA(%0) } }

<MID    (TWOACS)                /* negative of double */

{ LVI NI(%0) }
```

```
{ LVM NM(%0) }
{ LVP NP(%0) }
{ LVA(@1) NA(%0) }
{ LNI VI(%0) }
{ LNM VM(%0) }
{ LNP VP(%0) }
{ LNA(@1) VA(%0) }

{ LVM   { G(@1) .DMOVN..%0.%1. F2(%1) VA(%0) } }

[SBII, [SBPI, [SBPP             /* subtract */

{ LVA(@1)
        { RVI .SUBI..%1.%0. VA(%1) }
        { RNI .ADDI..%1.%0. VA(%1) }
        { RVM .SUB..%1.%0. F(%0) VA(%1) }
}

[MLII   (COMMUTE)               /* multiply */
{ LVA(@1)
        { RVI .IMULI..%1.%0. VA(%1) }
        { RVM .IMUL..%1.%0. F(%0) VA(%1) } }

[DVII                   /* divide */

{ GETFOL(@1) { LVA(%1) { INCR(%1)
        { RVI .IDIVI..%2.%0.
                { IDENT(%4,-3) { G(@1) .MOVE..%0.%4. F(%3) F(%4) VA(%0) } }
                { DIFFER(%4,-3) { FNEW(@1,%3) VA(%4) } } }
        { RVM .IDIV..%2.%0.
                { IDENT(%4,-3) { G(@1) .MOVE..%0.%4. F(%3) F(%4) F(%2) VA(%0) } }
                { DIFFER(%4,-3) { FNEW(@1,%3) F(%2) VA(%4) } } }
} } }
{ LVA(-1)
        { RVA(@1) .IDIVM..%1.%0. F(%1) VA(%0) } }

[MOII                   /* modulus */

{ GETPRE(@1) { LVA(%1) { INCR(%1)
        { RVI .IDIVI..%2.%0.
                { IDENT(%4,-3) { G(@1) .MOVE..%0.%3. F(%4) F(%3) VA(%0) } }
                { DIFFER(%4,-3) { FNEW(@1,%4) VA(%3) } } }
        { RVM .IDIV..%2.%0.
                { IDENT(%4,-3) { G(@1) .MOVE..%0.%3. F(%4) F(%3) F(%2) VA(%0) } }
                { DIFFER(%4,-3) { FNEW(@1,%4) F(%2) VA(%3) } } }
} } }

[LSII                   /* left shift */
{ LVA(@1)
        { RVJ(LHBAD) .LSH..%1.%0. F(%0) VA(%1) } }

[LSJI                   /* left shift of unsigned int */
{ LVA(@1) { RVJ(LHBAD) .LSH..%1.%0. F(%0) VA(%1) } }

[AMII   (LSTL)          /* assign-multiply */

{ LWJ(LHOK)
        { RVA(@1) .IMULB..%0.%1. F(%1) VA(%0) }
        { RVA(-1) .IMULM..%0.%1. F(%1) F(%0) E }
}

[ADII   (LSTL)          /* assign-divide */

{ SVA(-2) F(%0) E }
```

```
{ GETFOL(@1) { LWJ(LHOK) { G(%2) .MOVE..%0.%1. { INCR(%1)
    { RVI .IDIVI..%2.%0. .MOVEM..%2.%3.
        { DIFFER(%5,-3) { FNEW(@1,%3) F(%5) VA(%4) } }
        { IDENT(%5,-3) { G(@1) .MOVE..%0.%4. F(%3) F(%4) F(%5) VA(%0) } } }
    { RVM .IDIV..%2.%0. .MOVEM..%2.%3.
        { DIFFER(%5,-3) { FNEW(@1,%3) F(%2) F(%5) VA(%4) } }
        { IDENT(%5,-3) { G(@1) .MOVE..%0.%4. F(%1) F(%3) F(%4) F(%5) VA(%0) } } }
} } } }

[AUII   (LSTL)          /* assign-modulus */

{ SVA(-2) F(%0) E }

{ GETPRE(@1) { LWJ(LHOK) { G(%2) .MOVE..%0.%1. { INCR(%1)
    { RVI .IDIVI..%2.%0. .MOVEM..%1.%3.
        { DIFFER(%5,-3) { FNEW(@1,%4) F(%5) VA(%3) } }
        { IDENT(%5,-3) { G(@1) .MOVE..%0.%3. F(%3) F(%4) F(%5) VA(%0) } } }
    { RVM .IDIV..%2.%0. .MOVEM..%1.%3.
        { DIFFER(%5,-3) { FNEW(@1,%4) F(%2) F(%5) VA(%3) } }
        { IDENT(%5,-3) { G(@1) .MOVE..%0.%3. F(%2) F(%3) F(%4) F(%5) VA(%0) } } }
} } } }

[ALII   (LSTL)          /* assign-left-shift */

{ SVP F(%0) E }

{ LWJ(LHOK)
        { ISAC(%1)
                { RVJ(LHBAD) .LSH..%2.%0. F(%0) VP(%2) } }
        { NOTAC(%1)
                { G(@1) .MOVE..%0.%2.
                        { RVJ(LHBAD) .LSH..%1.%0. .MOVEM..%1.%3. F(%0) F(%3) VA(%1) } } }
}

<CMI                    /* one's complement */

{ LVI CI(%0) }
{ LVM CM(%0) }
{ LVP CP(%0) }
{ LVA(@1) CA(%0) }

{ LCI VI(%0) }
{ LCM VM(%0) }
{ LCP CP(%0) }
{ LCA(@1) VA(%0) }

{ LVI { G(@1) .SETCMI..%0.%1. VA(%0) } }

{ LVM { G(@1) .SETCM..%0.%1. F(%1) VA(%0) } }

{ LVA(@1) .SETCA..%0.%0. VA(%0) }

[BAII   (COMMUTE)       /* bitwise and */

{ IDENT(@1,EQ) { LVP
        { RVI .TRNE..%1.%0. F(%1) K }
        { RVH .TLNE..%1.%0. F(%1) K }
        { RVM .TDNE..%1.%0. F(%1) F(%0) K }
} }
{ IDENT(@1,NE) { LVP
        { RVI .TRNN..%1.%0. F(%1) K }
        { RVH .TLNN..%1.%0. F(%1) K }
        { RVM .TDNN..%1.%0. F(%1) F(%0) K }
} }
```

```
{ LVA(@1)
        { RVI .ANDI..%1.%0. VA(%1) }
        { RVM .AND..%1.%0. F(%0) VA(%1) }
        { RCI .ANDCMI..%1.%0. VA(%1) }
        { RCM .ANDCM..%1.%0. F(%0) VA(%1) }
}
{ LCA(@1)
        { RVI .ANDCAI..%1.%0. VA(%1) }
        { RVM .ANDCA..%1.%0. F(%0) VA(%1) }
        { RCI .ANDCBI..%1.%0. VA(%1) }
        { RCM .ANDCB..%1.%0. F(%0) VA(%1) }
}

[BOII   (COMMUTE)       /* bitwise or */

{ LVA(@1)
        { RVI .ORI..%1.%0. VA(%1) }
        { RVM .OR..%1.%0. F(%0) VA(%1) }
        { RCI .ORCMI..%1.%0. VA(%1) }
        { RCM .ORCM..%1.%0. F(%0) VA(%1) }
}
{ LCA(@1)
        { RVI .ORCAI..%1.%0. VA(%1) }
        { RVM .ORCA..%1.%0. F(%0) VA(%1) }
        { RCI .ORCBI..%1.%0. VA(%1) }
        { RCM .ORCB..%1.%0. F(%0) VA(%1) }
}

[BXII   (COMMUTE)       /* bitwise exclusive or */

{ LVA(@1)
        { RVI .XORI..%1.%0. VA(%1) }
        { RVM .XOR..%1.%0. F(%0) VA(%1) }
        { RCI .EQVI..%1.%0. VA(%1) }
        { RCM .EQV..%1.%0. F(%0) VA(%1) }
}
{ LCA(@1)
        { RVI .EQVI..%1.%0. VA(%1) }
        { RVM .EQV..%1.%0. F(%0) VA(%1) }
        { RCI .XORI..%1.%0. VA(%1) }
        { RCM .XOR..%1.%0. F(%0) VA(%1) }
}

[AAII   (LSTL)          /* assign-and */

{ LWJ(LHOK)
    { ISAC(%1)
        { RVI .ANDI..%2.%0. VP(%2) E }
        { RVM .AND..%2.%0. F(%0) VP(%2) E }
        { RCI .ANDCMI..%2.%0. VP(%2) E }
        { RCM .ANDCM..%2.%0. F(%0) VP(%2) E }
    }
    { NOTAC(%1)
        { RVA(@1) .ANDB..%0.%2. F(%2) VA(%0) }
        { RVA(-1) .ANDM..%0.%2. F(%2) F(%0) E }
        { RCA(@1) .ANDCAB..%0.%2. F(%2) VA(%0) }
        { RCA(-1) .ANDCAM..%0.%2. F(%2) F(%0) E }
    }
}

[AOII   (LSTL)          /* assign-or */

{ LWJ(LHOK)
    { ISAC(%1)
        { RVI .ORI..%2.%0. VP(%2) E }
```

```
        { RVM .OR..%2.%0. F(%0) VP(%2) E }
        { RCI .ORCMI..%2.%0. VP(%2) E }
        { RCM .ORCM..%2.%0. F(%0) VP(%2) E }
    }
    { NOTAC(%1)
        { RVA(@1) .ORB..%0.%2. F(%2) VA(%0) }
        { RVA(-1) .ORM..%0.%2. F(%2) F(%0) E }
        { RCA(@1) .ORCAB..%0.%2. F(%2) VA(%0) }
        { RCA(-1) .ORCAM..%0.%2. F(%2) F(%0) E }
    }
}

[AXII   (LSTL)          /* assign-exclusive-or */

{ LWJ(LHOK)
    { ISAC(%1)
        { RVI .XORI..%2.%0. VP(%2) E }
        { RVM .XOR..%2.%0. F(%0) VP(%2) E }
        { RCI .EQVI..%2.%0. VP(%2) E }
        { RCM .EQV..%2.%0. F(%0) VP(%2) E }
    }
    { NOTAC(%1)
        { RVA(@1) .XORB..%0.%2. F(%2) VA(%0) }
        { RVA(-1) .XORM..%0.%2. F(%2) F(%0) E }
        { RCA(@1) .EQVB..%0.%2. F(%2) VA(%0) }
        { RCA(-1) .EQVM..%0.%2. F(%2) F(%0) E }
    }
}


/*******        FLOATING POINT ARITHMETIC       *******/

[PLFF   (COMMUTE)       /* float plus float */

{ LVA (@1)
         {RVI .FADRI..%1.%0. VA(%1)}
         {RNI .FSBRI..%1.%0. VA(%1) }
         {RVM .FADR..%1.%0. F(%0) VA(%1)}}


[PLDD   (COMMUTE,TWOACS) /* double plus double */

{LVA(@1)
        {RVM .DFAD..%1.%0. F2(%0) VA(%1)}}


[SBFF                   /* float minus float */

{LVA(@1)
        {RVI .FSBRI..%1.%0. VA(%1)}
        {RNI .FADRI..%1.%0. VA(%1) }
        {RVM .FSBR..%1.%0. F(%0) VA(%1)}}


[SBDD   (TWOACS)        /* double minus double */

{LVA(@1)
        {RVM .DFSB..%1.%0. F2(%0) VA(%1)}}


[MLFF   (COMMUTE)       /* float times float */

{LVA(@1)
        {RVI .FMPRI..%1.%0. VA(%1)}
        {RVM .FMPR..%1.%0. F(%0) VA(%1)}}
```

```
[MLDD   (COMMUTE,TWOACS) /* double time double */

{LVA(@1)
        {RVM .DFMP..%1.%0. F2(%0) VA(%1)}}


[DVFF                   /* float divided by float */

{LVA(@1)
        {RVI .FDVRI..%1.%0. VA(%1) }
        {RVM .FDVR..%1.%0. F(%0) VA(%1)}}


[DVDD   (TWOACS)        /* double divided by double */

{ LVA(@1)
        {RVM .DFDV..%1.%0. F2(%0) VA(%1) }}


[APFF   (LSTL)          /* float assign-plus float */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVH .FADRI..%2.%0. VP(%2) E }
                { RVM .FADR..%2.%0. F(%0) VP(%2) E }}
        { NOTAC(%1)
                { RVA(@1) .FADRB..%0.%2. F(%2) VA(%0) }
                { RVP .FADRM..%0.%2. F(%0) F(%2) E }}}




[ASFF   (LSTL)          /* float assign-minus float */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVH .FSBRI..%2.%0. VP(%2) E }
                { RVM .FSBR..%2.%0. F(%0) VP(%2) E }}
        { NOTAC(%1)
                { RVH
                        { NEGATE(%1)
                                { G(@1) .HRLOI..%0.%1. .FADRB..%0.%4. F(%4) VA(%0) }
                                { G(-1) .HRLOI..%0.%1. .FADRM..%0.%4. F(%0) F(%4) E }}}
                { RNA(@1) .FADRB..%0.%2. F(%2) VA(%0) }
                { RNP .FADRM..%0.%2. F(%0) F(%2) E }}}



[AMFF   (LSTL)          /* float assign-multiply float */

{ LWJ(LHOK)
        { RVA(@1) .FMPRB..%0.%1. F(%1) VA(%0) }
        { RVA(-1) .FMPRM..%0.%1. F(%1) F(%0) E }}


[ADFF   (LSTL)          /* float assign-divide float */

{ SVA(-2) F(%0) E }
{ LWJ(LHOK)
        { ISAC(%1)
                { RVH .FDVRI..%2.%0. VP(%2) E }
                { RVM .FDVRM..%2.%0. F(%2) VP(%0) E }}
        { NOTAC(%1)
```

```
                { G(-1) .MOVE..%0.%2.
                        { RVH .FDVRI..%1.%0. .MOVEM..%1.%2. F(%1) E }
                        { RVM .FDVR..%1.%0. .MOVEM..%1.%2. F(%0) F(%1) E }}}}
[APDD   (LSTL,TWOACS)   /* double assign-plus double */

/* doesn't return on VA, but wait until AGG is redone to fix */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVM .DFAD..%2.%0. F2(%0) VP(%2) E }}
        { NOTAC(%1)
                { G(@1) .DMOVE..%0.%2.
                        { RVM .DFAD..%1.%0. .DMOVEM..%1.%3. F2(%0) VA(%1) }}}}
{ IF(NOTRANS) { SVA(-2) F2(%0) E } }

[ASDD   (LSTL,TWOACS)   /* double assign-minus double */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVM .DFSB..%2.%0. F2(%0) VP(%2) E }}
        { NOTAC(%1)
                { G(@1) .DMOVE..%0.%2.
                        { RVM .DFSB..%1.%0. .DMOVEM..%1.%3. F2(%0) VA(%1) }}}}
{ IF(NOTRANS) { SVA(-2) F2(%0) E } }


[AMDD   (LSTL,TWOACS)   /* double assign-multiply double */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVM .DFMP..%2.%0. F2(%0) VP(%2) E }}
        { NOTAC(%1)
                { G(@1) .DMOVE..%0.%2.
                        { RVM .DFMP..%1.%0. .DMOVEM..%1.%3. F2(%0) VA(%1) }}}}
{ IF(NOTRANS) { SVA(-2) F2(%0) E } }

[ADDD   (LSTL,TWOACS)   /* double assign-divide double */

{ LWJ(LHOK)
        { ISAC(%1)
                { RVM .DFDV..%2.%0. F2(%0) VP(%2) E }}
        { NOTAC(%1)
                { G(@1) .DMOVE..%0.%2.
                        { RVM .DFDV..%1.%0. .DMOVEM..%1.%3. F2(%0) VA(%1) }}}}
{ IF(NOTRANS) { SVA(-2) F2(%0) E } }
     2  /*******        FILE:TRANS      *******/
     3  /*******                        *******/
     4  /* TRANS - Build internal format translation tables
     5   *      written by David H. Ackley
     6   *              copyright (c) 1981 D. H. Ackley
     7   *              Permission to copy without fee all or part of
     8   *              this program is granted provided that the
     9   *              copies are not made or distributed for direct
    10   *              commercial advantage and that this copyright
    11   *              notice is included in its entirety.
    12   *
    13   * Reads: list of files.
    14   * Writes:
    15   *      FOREST.C - the table definitions.
    16   *      FNCTAB.C - table of function dispatches.
    17   *      INTOPS.HDR - map of parse tree operators onto indices into ROOTS
    18   *      MODES.HDR - map of mode names onto their bit positions
    19   *
    20   * Input specification:
    21   *      codelist - listing of opcodes with execution times
```

```
 *      modelist - listing of all modes used separated by commas.
 *      tablelist - a sequence of tables terminated by end-of-files.
 */
#old on
#include <IODEFS.HDR>
#define EVER (;;)

struct { unsigned short lhalf, rhalf; };

#define STRIP(p) ((intgr=(p))&~0400000)

#define NEW 0           /* symbol table flags */
#define OLD 1
#define EITHER 2

#define NAME0 0         /* symbol table items */
#define NAME1 1
#define TYPE 2
#define VALUE 3
#define LEFT 4
#define RIGHT 5
#define HASHSIZE 6

#define MODE 0          /* symbol table type values */
#define FCTN 1
#define SPVAL 2
#define TABLE 3
#define LABEL 4
#define FLAG 5
#define MCODE 6

int tvals[]={"mode","function","literal value","table","label","flag","code"};

#define BROTHER 0       /* translation table items */
#define MODES 1
#define FUNC 2
#define RETNXT 3
#define COUNT 4
#define WRITTEN 5
#define CODE 6
#define TABLESIZE 15
#define MAXCODES (TABLESIZE-CODE)

#define KEY 0           /* byte offsets into FUNC */
#define MODEA 4

#define RLEFT 0         /* KEY values */
#define RRIGHT 1
#define RSELF 2

char cmdlin[150],*cmdp;

#define FILES 10
int inpc,treec,spvc,fncc,ftrc,tric,atrc,atdc,codc,cdcc,modc;
int intgr;

int spvidx,fncidx,tblidx,modenum,flgnum,codnum,opnum;
int modcodnum, appear, modappear;


int name[2];

int rmodes,*link,tblname;
int attrib;
int uflag;
```

```
    87  int price;
    88
    89  #define HASHTABSIZE 401
    90  int hashtab[HASHTABSIZE];
    91
    92  int wcount,lincount,scount;
    93
    94  int *chn[]={&treec,&spvc,&fncc,&ftrc,&tric,&atrc,&atdc,&codc,&cdcc,&modc};
    95  int spc[]={"FOREST.C","SPVAL.C","FNCTAB.C","ROOTS.C","INTOPS.HDR","ATTRIB.C","ATRDEF
.C","CODE.C","CODCST.C","MODES.HDR"};
    96  int apnd[]={0,1,0,1,0,1,5,1,1,0};
    97  int hdr[]={
    98          "#data high\r\n",
    99          "\r\n#define LIT 0\r\nint spval[]={LIT,0",
   100          "\r\nint (*fnctab[])()={0,0,0",
   101          "\r\nint roots[]={0",
   102          "",
   103          "\r\nint attrib[]={0",
   104          "",
   105          "#data high\r\nint codtab[]={\"\"",
   106          "#data high\r\nint codcst[]={0",
   107          ""};
   108
   109  int dne[]={
   110          "",
   111          "};\r\n",
   112          "};\r\n",
   113          "};\r\n",
   114          "",
   115          "};\r\n",
   116          "",
   117          "};\r\n",
   118          "};\r\n",
   119          ""};
   120
   121  main(){
   122          init();
   123          codedecs();
   124          flagdecs();
   125          modedecs();
   126          for EVER {
   127                  if (try(0)) break;
   128                  writetable(readtable());
   129          }
   130          closeo();
   131          use(0);
   132          printf("\r\nTables: %d Functions: %d Spvals: %d\r\n",tblidx,fncidx,spvidx/2)
;
   133          printf("Used %d of 35 modes (%d%%)\r\n",modenum,modenum*100/35);
   134          scount =/ 5;
   135          printf("Storage: %d words (tables) + %d words (strings) = %d (%dP)\r\n",
   136                  wcount,scount,wcount+scount,((wcount+scount)>>8)+1);
   137          printf("Defined %d of 35 attributes (%d%%)\r\n",flgnum,flgnum*100/35);
   138          printf("Used %d of %d known opcodes (%d%%)\r\n",codnum,opnum,codnum*100/opnu
m);
   139          printf("%d opcodes (%d modified) appeared %d times (%d with modifiers)\r\n",
   140                  codnum, modcodnum, appear, modappear);
   141  }
   142
   143  init(){
   144          int i;
   145          for (i = 0;i<FILES;++i){
   146                  if ((*chn[i] = open(spc[i],IO_AWRITE))<=0)
   147                          error("can't write: %s",spc[i]);
   148                  printf("%U%r",*chn[i],hdr[i]);
```

```
   149          }
   150          printf("%UFiles: ",0);
   151          getlin(cmdlin);
   152          cmdp = cmdlin;
   153          inpc = open("NUL:",IO_AREAD);
   154          use(treec);
   155          wcount = 0;
   156          scount = 0;
   157          tblidx = 0;
   158          modenum = 0;
   159          spvidx = 0;
   160          fncidx = 2;
   161          flgnum = 0;
   162          codnum = 0;
   163          opnum = 0;
   164  }
   165
   166  codedecs(){
   167          char *b;
   168          while (try('!')) {
   169                  price = readdec();
   170                  do {
   171                          readname();
   172                          hash(NEW,MCODE);
   173                          if (try('*')) printf("%U#define %s %d\r\n",atdc,b= name,pric
e);
   174                  } while (try(','));
   175          }
   176  }
   177
   178  flagdecs(){
   179          char *cp;
   180          strcopy("LEAF",cp = name);
   181          hash(NEW,FLAG);
   182          strcopy("UNOP",cp = name);
   183          hash(NEW,FLAG);
   184          strcopy("BINOP",cp = name);
   185          hash(NEW,FLAG);
   186  }
   187
   188  strcopy(fp,tp) char *fp,*tp;{
   189          --fp; --tp;
   190          while (*++tp = *++fp);
   191  }
   192
   193  strcmp(fp,tp) char *fp,*tp; {
   194          --fp; --tp;
   195          while (*++fp==*++tp) if(*fp==0) return(1);
   196          return (0);
   197  }
   198
   199  modedecs(){
   200          for EVER {
   201                  readname();
   202                  hash(NEW,MODE);
   203                  if (try(',')) continue;
   204                  break;
   205          }
   206  }
   207
   208  readtable(){
   209          int *p;
   210          char *b;
   211          link = 0;
   212          tblname = 0;
```

```
213          attrib = 0;
214          for EVER {
215                  readtname();
216                  if (tblname==0) {
217                          tblname = name[0];
218                          switch (*(b = name)){
219                     case '^':    attrib =| 1; break;
220                     case '<':    attrib =| 2; break;
221                     case '[':    attrib =| 4; break;
222                          }
223                  }
224                  link = hash(NEW,TABLE,link);
225                  if (try(',')) continue;
226                  break;
227          }
228          if (try('('){
229                  for EVER {
230                          readname();
231                          if (strcmp("UNIVERSAL",b = name)) uflag = 1;
232                           else {
233                                  p = hash(EITHER,FLAG);
234                                  attrib =| p[VALUE];
235                          }
236                          if (try(')')) break;
237                          getit(',');
238                  }
239          }
240          return(readalts());
241  }
242
243  readalts(){
244          int *p,modes;
245          p = readlist();
246          modes = rmodes;
247          if (peek('{')||peek(':')||peek('*')) {
248                  p[BROTHER] = readalts();
249                  rmodes =| modes;
250          }
251          return(p);
252  }
253
254  readlist(){
255          int *p,*pp,*lbl;
256          char *b;
257          lbl = 0;
258          if (try(':')) {
259                  readname();
260                  name[1] = name[0];
261                  name[0] = tblname;
262                  lbl = hash(NEW,LABEL);
263          }
264          if (peek('*')) {
265                  p = tblref();
266                  if (lbl) lbl[VALUE] = p;
267                  return (p);
268          } else {
269                  getit('{');
270                  p = alcore(TABLESIZE);
271                  if (lbl) lbl[VALUE] = p;
272                  for (pp=p;pp<&p[TABLESIZE];++pp)*pp=0;
273                  if (link) {
274                          printf("\r\n/* ");
275                          while (link) {
276                                  printf(" %s(",b= &link[NAME0]);
277                                  printf("%U,\r\nt%+5d",ftrc,STRIP(p));
```

```
   278                                  printf("%U\r\n,0%o",atrc,attrib);
   279                                  pp = link[VALUE];
   280                                  link[VALUE] = p;
   281                                  printf("%U#define %s %d\r\n"
   282                                 ,tric,stringer(b = &link[NAME0]),++tblidx);
   283                                  printf("%d) ",tblidx);
   284                                  if (uflag) {
   285                                          printf("%U#define UNIVERSAL %d\r\n",atdc,tbl
idx);
   286                                          uflag = 0;
   287                                  }
   288                                  link = pp;
   289                          }
   290                          printf("*/\r\n");
   291                  }
   292                  readkey(p);
   293                  readcode(p);
   294                  if (peek(':') || peek('{')|| peek('*'))
   295                           p[RETNXT] = readalts();
   296                   else p[RETNXT] = readreturns();
   297                  p[MODES] = rmodes;
   298                  getit('}');
   299                  return(p);
   300          }
   301  }
   302
   303  readkey(p) int *p;{
   304          char *b;
   305          b = &p[FUNC];
   306          switch(getcn()){
   307     case 'L':    b[KEY] = RLEFT; goto getmode;
   308     case 'R':    b[KEY] = RRIGHT; goto getmode;
   309     case 'S':    b[KEY] = RSELF;
   310     getmode:     b[MODEA] = readmode();
   311                  break;
   312     default:     reeat(inpc);
   313                  b[KEY] = readfunc();
   314          }
   315          if (try('(')) {
   316                  for EVER {
   317                          *++b = readarg();
   318                          if (try(',')) continue;
   319                          break;
   320                  }
   321                  getit(')');
   322                  b = &p[FUNC];
   323          }
   324          return;
   325  }
   326
   327  readcode(p) int *p;{
   328          int *ip,*count,i;
   329          int cod, mod;
   330          char *b;
   331          count = &p[COUNT];
   332          p = &p[CODE];
   333          *count = 0;
   334          while (try('.')) {
   335                  if(*count >= MAXCODES) {
   336                          error("more than %d instructions per alternative", MAXCODES)
;
   337                  }
   338                  cod = mod = 0;
   339                  *p = 0;
   340                  if(!peek('.')) {
```

```
341                         ++cod;
342                         readname();
343                         ip = hash(OLD,MCODE);
344                         if (ip[VALUE]<0){
345                                 printf("%U,\"%s\"",codc,b=name);
346                                 printf("%U,%d",cdcc,-ip[VALUE]);
347                                 ip[VALUE] = ++codnum;
348                                 ++cod;
349                                 if (codnum%10==0) printf("%U\r\n%U\r\n",codc,cdcc);
350                         }
351                         *p =| ip[VALUE]<<22;
352                 }
353                 b = p;
354                 ++b;
355                 getit('.');
356                 for (i = 0;i<3;++i) {
357                         if (try('.')) *++b = 0;
358                          else {
359                                 if(i==0) ++mod;
360                                 *++b = readarg();
361                                 if (*b>>5==2) error("@ reference in code string");
362                                 getit('.');
363                         }
364                 }
365                 ++*count;
366                 ++p;
367                 if(cod > 0) {   /* statistics collection */
368                         ++appear;
369                         if(mod > 0) {
370                                 ++modappear;
371                                 if(cod > 1) ++modcodnum;
372                         }
373                 }
374         }
375         return;
376 }
377
378 readfunc(){
379         int *p;
380         readname();
381         p = hash(EITHER,FCTN);
382         if (p[TYPE]!=FCTN)
383                  error("function needed, found a %s",tvals[p[TYPE]]);
384         return(p[VALUE]);
385 }
386
387 readmode(){
388         int *p;
389         readname();
390         p = hash(OLD,MODE);
391         if (p[TYPE]!=MODE)
392                 error("mode needed, found a %s",tvals[p[TYPE]]);
393         return(p[VALUE]);
394 }
395
396 tblref(){
397         int **p,n;
398         char *b;
399         getit('*');
400         readtname();
401         n = name[0];
402         p = hash(OLD,TABLE);
403         if (p[TYPE]!=TABLE) error("not table name in *ref: %s",b=name);
404         getit(':');
405         readname();
```

```
406          name[1] = name[0];
407          name[0] = n;
408          p = hash(OLD,LABEL);
409          if (p[TYPE]!=LABEL) error("not label in *ref: %s:%s",
410                          b = &name[0],b = &name[1]);
411          rmodes = (p[VALUE])[MODES];
412          return (p[VALUE]);
413  }
414
415  readreturns(){
416          char *rp,*cp;
417          int *p;
418          extern int *alcore();
419          rp = cp = alcore(8);
420          --cp;
421          rmodes = 0;
422          while (!peek('}')) {
423                  readname();
424                  p = hash(EITHER,FCTN);
425                  if (p[TYPE]==MODE) {
426                          rmodes =| (unsigned)1<<(p[VALUE]-1);
427                          *++cp = p[VALUE];
428                  } else  *++cp = p[VALUE]|0100;
429                  if (try('(')) {
430                          *++cp = readarg();
431                          if(peek(',')) error("only one argument allowed in returns");
432                          getit(')');
433                  } else  *++cp = 0;
434          }
435          *++cp = 0;
436          return(rp);
437  }
438
439  readarg(){
440          int *p;
441          if (try('%')) return(readdec()|0140);
442          if (try('@')) return(readdec()|0100);
443          name[0] = *(p = "lit");
444          name[1] = readnum();
445          p = hash(EITHER,SPVAL);
446          return (p[VALUE]);
447  }
448
449  readdec(){int n,c;
450          n = 0;
451          c = getcn();
452          while (c>='0' && c<='9') {n = n*10+c-'0';
453                                    c = tgetc();}
454          reeat(inpc);
455          return(n);
456  }
457
458  readnum(){
459          unsigned n;
460          int sign,c;
461          sign = 1;
462          n = 0;
463          if (try('-')) sign = -1;
464          while ((c = tgetc())>='0' && c<='7') n = (n<<3)+c-'0';
465          if(c=='8' || c=='9') error("non-octal digit");
466          reeat(inpc);
467          return(sign*n);
468  }
469
470  readtname(){
```

```
471         char *b,c;
472         name[0] = name[1] = 0;
473         b = name;
474         while ((c = getcn())>='A' && c<='Z' ||
475                 c>='0' && c<='9' ||
476                 c=='<' || c=='[' || c=='^') *b++ = c;
477         reeat(inpc);
478 }
479
480 readname(){
481         char *b,c;
482         name[0] = name[1] = 0;
483         b = name;
484         c = getcn();
485         while (c>='A' && c<='Z' || c>='0' && c<='9') {
486                 *b++ = c;
487                 c = tgetc();
488         }
489         reeat(inpc);
490 }
491
492 char strnam[15];
493 stringer(b) char *b;{
494         char *p;
495         p = &strnam[-1];
496         while (*b) switch (*b++){
497            case '[':
498            case '<':
499            case '^':    continue;
500           default:      *++p = b[-1]; continue;
501         }
502         *++p = 0;
503         return (&strnam[0]);
504 }
505
506 tgetc(){
507         char c,*cp;
508 again:  c = getu(inpc);
509         if (c=='\t' || c=='\r') c = ' ';
510          else if (c=='\n') {++lincount; c = ' ';}
511         if (c<' ' && c!=0)goto again;
512         if (c!=0 || *cmdp==0) return(c);
513         c = 0;
514         for (cp = cmdp;(*cmdp!=',' || c) && *cmdp;++cmdp) {
515                 if (*cmdp=='[') c = 1;
516                  else if (*cmdp==']') c = 0;
517         }
518         *cmdp++ = 0;
519         close(inpc);
520         if ((inpc = open(cp,IO_AREAD))<=0) error("can't read: %s",cp);
521         printf("%U[Processing %s]\r\n",0,cp);
522         lincount = 1;
523         goto again;
524 }
525
526 getcn(){
527         int c;
528         while ((c = tgetc())==' ');
529         return(c);
530 }
531
532 getit(c){
533         if (c==getcn()) return;
534         reeat(inpc);
535         if (getcn()=='#')
```

```
536          error("expected %c, found # (file not preprocessed?)",c);
537         reeat(inpc);
538         error("expected %c, found %c",c,tgetc());
539 }
540
541 try(c){
542         if (c==getcn()) return(1);
543         reeat(inpc);
544         return(0);
545 }
546
547 peek(c){
548         int cc;
549         cc = getcn();
550         reeat(inpc);
551         if (c==cc) return(1);
552         return(0);
553 }
554
555 error(s,args){
556         printf("%U\r\nError on line %d: %r\r\nTrace? ",0,lincount,s,&args);
557         getlin(cmdlin);
558         if (cmdlin[0]=='y')strace();
559         printf("%USymbol table dump? ",0);
560         getlin(cmdlin);
561         if (cmdlin[0]=='y')dump();
562         printf("%UUpdate files? ",0);
563         getlin(cmdlin);
564         if (cmdlin[0]=='y') closeo();
565         exit();
566 }
567
568 getlin(b) char *b;{
569         --b;
570         while ((*++b = getl(0))!='\n');
571         *b-- = 0; *b = 0;
572 }
573
574 getl(ch) {
575         int c;
576         if((c=getc(ch))>='A' && c<='Z') c+='a'-'A';
577         return(c);
578 }
579 getu(ch) {
580         int c;
581         if((c=getc(ch))>='a' && c<='z') c-='a'-'A';
582         return(c);
583 }
584
585 writetable(argp){
586         register *p,*pp;
587         char *b;
588         if ((p = argp)==0 || p[WRITTEN]) return;
589         writetable(p[BROTHER]);
590         if (p[RETNXT].lhalf==0) writetable(p[RETNXT]);
591         p[WRITTEN] = 1;
592         wcount =+ CODE+p[COUNT];
593         printf("\tint t%+5d[]={",STRIP(p));
594         if (p[BROTHER]) printf("t%+5d,",STRIP(p[BROTHER]));
595          else printf("0,");
596         printf("0%o,0%o,",p[MODES],p[FUNC]);
597         if (p[RETNXT].lhalf) {
598                 printf("\"");
599                 for (b = p[RETNXT];*b!=0;b =+ 2,scount =+ 2)
600                         printf("\\%+3o\\%+3o",b[0],b[1]);
```

```
601                 printf("\",");
602                 ++scount;
603         } else  printf("t%+5d,",STRIP(p[RETNXT]));
604         printf("%d",p[COUNT]);
605         for (pp = &p[CODE];pp<&p[CODE+p[COUNT]];++pp)
606                 printf(",0%o",*pp);
607         printf("};\r\n");
608 }
609
610 closeo(){
611         int i,appc,chnl;
612         printf("%U[Closing files...]\r\n",0);
613         for (i = 0;i<FILES;++i){
614                 printf("%U%r",*chn[i],dne[i]);
615                 if (apnd[i]) {
616                         close(*chn[i]);
617                         appc = open(spc[i],IO_AREAD);
618                         chnl = *chn[apnd[i]-1];
619                         while(put(chnl,getc(appc)));
620                         close(appc);
621                         delete(spc[i]);
622                 }
623         }
624         for (i = 0;i<FILES;++i) if (apnd[i]==0) close(*chn[i]);
625         return;
626 }
627
628 hash(flag,type,value){
629         register *np,*ip;
630         int n;
631         char *b;
632         np = &hashtab[((name[0].lhalf)+(name[0].rhalf)+
633                         (name[1].lhalf)+(name[1].rhalf))%HASHTABSIZE];
634         while (ip = *np) {
635                 if (name[0]==ip[0] && name[1]==ip[1]) {
636                         if (flag==NEW)
637                                 error("redeclaration of %s",b=name);
638                         return(ip);
639                 }
640                 if (name[0]<ip[0] || name[0]==ip[0] && name[1]<ip[1])
641                         np = &ip[LEFT];
642                 else
643                         np = &ip[RIGHT];
644         }
645         if (flag==OLD) if (type==LABEL) {
646                 n = name[1];
647                 name[1] = 0;
648                 error("undefined label: %s:%s",b = name,b = &n);
649         } else  error("undefined %s: %s",tvals[type],b = name);
650
651         ip = alcore(HASHSIZE);
652         ip[0] = name[0];
653         ip[1] = name[1];
654         ip[TYPE] = type;
655         ip[LEFT] = ip[RIGHT] = 0;
656         *np = ip;
657         switch (type) {
658     case MODE:   ip[VALUE] = ++modenum;
659                 if (modenum>35) error("too many modes");
660                 printf("%U#define MODE_%s %d\r\n",modc,b=name,modenum);
661                 return(ip);
662
663     case FCTN:   ip[VALUE] = ++fncidx;
664                 printf("%U,\r\n%s",fncc,b=name);
665                 if (fncidx>077) error("too many functions");
```

```
   666                  return(ip);
   667
   668      case SPVAL:  ip[VALUE] = (spvidx =+ 2);
   669                   printf("%U,\r\n\tLIT,0%o",spvc,name[1]);
   670                   if (spvidx>077) error("too many spvals");
   671                   return(ip);
   672
   673      case FLAG:   ip[VALUE] = (unsigned)1<<flgnum++;
   674                   if (flgnum>35) error("too many flags");
   675                   printf("%U#define F%s 0%o\r\n",atdc,b=name,ip[VALUE]);
   676                   return(ip);
   677
   678      case MCODE:  ip[VALUE] = -price; ++opnum;
   679                   return(ip);
   680
   681      case TABLE:  ip[VALUE] = value;
   682      case LABEL:  return(ip);
   683           }
   684  }
   685
   686  dump(){
   687           int i;
   688           use(0);
   689           for (i=0;i<HASHSIZE;++i) if (hashtab[i]) {
   690                   printf("[%d]>",i);
   691                   dumpr(hashtab[i]);
   692           }
   693  }
   694
   695  dumpr(p) int **p;{
   696           char *b;
   697           if (p==0) return;
   698           dumpr(p[LEFT]);
   699           switch (p[TYPE]){
   700      case MODE:   printf("\tMode:\t%s,%d\r\n",b= &p[NAME0],p[VALUE]);
   701                   break;
   702
   703      case FCTN:   printf("\tFctn:\t%s,%d\r\n",b= &p[NAME0],p[VALUE]);
   704                   break;
   705
   706      case SPVAL:  printf("\tSpval:\t%d,%d\r\n",p[NAME1],p[VALUE]);
   707                   break;
   708
   709      case TABLE:  printf("\tTable:\t%s,t%+5d\r\n",b= &p[NAME0],STRIP(p[VALUE]));
   710                   break;
   711
   712      case LABEL:  printf("\tLabel:\t%s:%s,t%+5d\r\n",
   713                           b = &p[NAME1],b = &p[NAME0],STRIP(p[VALUE]));
   714                   break;
   715      case FLAG:   printf("\tFlag:\t%s,0%o\r\n",b= &p[NAME0],p[VALUE]);
   716                   break;
   717
   718      case MCODE:  printf("\tOpcode:\t%s,%d\r\n",b= &p[NAME0],p[VALUE]);
   719                   break;
   720           }
   721           dumpr(p[RIGHT]);
   722  }
/*******        FILE:  INTOPS.HDR       *******/
/*******                                *******/

#define UNIV 1
#define CF 2
#define CP 3
#define CI 4
#define CD 5
```

```
#define ROTC 325
#define MOVE 126
#define MOVS 126
#define HRR 126
#define HRRO 126
#define HRRZ 126
#define HRRE 126
#define HRL 126
#define HRLO 126
#define HRLZ 126
#define HRLE 126
#define HLR 126
#define HLRO 126
#define HLRZ 126
#define HLRE 126
#define HLL 126
#define HLLO 126
#define HLLZ 126
#define HLLE 126
#define EXCH 300
#define MOVN 132
#define MOVM 132
#define CAI 62
#define CAM 62
#define JRST 34
#define TLNN 62
#define FLEAF 01
#define FUNOP 02
#define FBINOP 04
#define FNOUNIV 010
#define UNIVERSAL 1
#define FTWOACS 020
/*******                                  *******/
/*******        FILE:  MODES.HDR          *******/
/*******                                  *******/


#define MODE_E 1
#define MODE_MA 2
#define MODE_VJ 3
#define MODE_VA 4
#define MODE_VI 5
#define MODE_VH 6
#define MODE_VP 7
#define MODE_VO 8
#define MODE_VM 9
#define MODE_CI 10
#define MODE_CM 11
#define MODE_CP 12
#define MODE_CA 13
#define MODE_NI 14
#define MODE_NM 15
#define MODE_NP 16
#define MODE_NA 17
#define MODE_J 18
#define MODE_K 19
#define MODE_KA 20
#define MODE_WJ 21
#define MODE_WI 22
#define MODE_WP 23
#define MODE_WM 24
#define MODE_WA 25
#define MODE_AE 26
#define MODE_AC 27
#define MODE_VZ 28
```

```
#define MODE_VN 29
/*******                                  *******/
/*******        FILE:  FNCTAB.C           *******/
/*******                                  *******/


int (*fnctab[])()={0,0,0,
ACSIZE,
G,
IDENT,
F,
F2,
AT,
DIFFER,
INVERT,
NULL,
NEWLABEL,
INCR,
IFFUNC,
NOEFFECT,
X,
ISZ,
ISO,
ISN,
HLZ,
HRZ,
HLO,
CMPL,
NEGATE,
HRO,
MAKLIT};
/*******                                  *******/
/*******        FILE:  FOREST.C           *******/
/*******                                  *******/


#data high

/*  [UNIV(1) */
        int t19931[]={0,01,04000000002,"\001\000",0};
        int t19915[]={0,01,02,t19931,0};
        int t19899[]={0,01,074200000000,t19915,0};
        int t19874[]={0,01,02,"\001\000",0};
        int t19858[]={t19899,01,074140000000,t19874,0};
        int t19826[]={t19858,01,074240000000,"\001\000",0};
        int t19801[]={0,02000000,056020000000,"\024\143",2,0134061702,0600304};
        int t19785[]={0,02000000,067020000000,t19801,0};
        int t19769[]={0,02000000,060000000000,t19785,0};
        int t19753[]={0,02000000,012040000010,t19769,0};
        int t19728[]={0,02000000,056040000000,"\106\142\024\140",1,0154160304};
        int t19712[]={0,02000000,056020000000,t19728,0};
        int t19696[]={t19753,02000000,010000000022,t19712,0};
        int t19671[]={0,01000000,056020000000,"\106\141\023\000",1,0154000302};
        int t19655[]={0,01000000,010000000022,t19671,0};
        int t19630[]={0,0400000,056040000000,"\106\142\022\000",1,0134161300};
        int t19614[]={0,0400000,056020000000,t19630,0};
        int t19598[]={t19655,0400000,010000000016,t19614,0};
        int t19568[]={t19696,01400000,070240000000,t19598,0};
        int t19552[]={0,03400000,027020200000,t19568,0};
        int t19536[]={t19826,03400000,014000000000,t19552,0};
        int t19511[]={0,01000000,012034141044,"\023\000",1,0600304};
        int t19488[]={0,01000000,067020000000,t19511,0};
        int t19465[]={0,01000000,060000000000,t19488,0};
        int t19442[]={t19536,01000000,046041000000,t19465,0};
        int t19410[]={0,0400000,056040000000,"\022\000",1,0100000300};
```

```
        int t19394[]={0,0400000,013030300046,t19410,0};
        int t19371[]={0,0400000,052020000000,t19394,0};
        int t19341[]={t19442,0400000,046060600000,t19371,0};
        int t19316[]={0,04000000,043030100000,"\025\140",0};
        int t19300[]={t19341,04000000,010000000060,t19316,0};
        int t19275[]={0,0100000000,022020000000,"\106\141\031\140",1,040060302};
        int t19259[]={t19300,0100000000,010000000060,t19275,0};
        int t19234[]={0,0100000000,022020000000,"\031\140",1,020060302};
        int t19218[]={t19259,0100000000,010000000054,t19234,0};
        int t19193[]={t19218,040000000,010000000056,"\030\140",0};
        int t19168[]={0,020000000,013020000062,"\027\140",0};
        int t19152[]={t19193,020000000,014000000000,t19168,0};
        int t19120[]={0,04,043030100000,"\003\140",0};
        int t19104[]={t19152,04,010000000022,t19120,0};
        int t19065[]={0,010,027020400000,"\107\143\004\142",1,060061306};
        int t19019[]={t19065,010,027020200000,"\106\143\004\142",1,040061306};
        int t19003[]={0,010,014000000000,t19019,0};
        int t18987[]={0,010,022020000000,t19003,0};
        int t18971[]={t19104,010,010000000022,t18987,0};
        int t18939[]={0,010,022020000000,"\004\140",1,020060302};
        int t18923[]={t18971,010,010000000012,t18939,0};
        int t18898[]={t18923,0400,010000000016,"\011\140",0};
        int t18873[]={0,0100,013020000010,"\007\140",0};
        int t18850[]={t18898,0100,014000000000,t18873,0};

/*  ^CF(2)  ^CP(3)  ^CI(4) */
        int t20901[]={0,0100,020040000000,"\007\140",1,040060302};
        int t20876[]={t20901,010,022020000000,"\004\140",1,040060302};
        int t20851[]={t20876,0400,054000000000,"\011\141",0};
        int t20828[]={0,0510,0153020000000,t20851,0};
        int t20803[]={0,0100,020040000000,"\007\140",1,0240060302};
        int t20778[]={t20803,010,022020000000,"\004\140",1,0240060302};
        int t20755[]={t20828,0110,0147020000000,t20778,0};
        int t20730[]={0,020000,0123020000000,"\016\141",0};
        int t20707[]={0,020000,0143040000000,t20730,0};
        int t20675[]={t20707,01000,0137040000000,"\012\140",0};
        int t20650[]={t20675,0100,020040000000,"\007\140",1,0220060302};
        int t20625[]={t20650,010,022020000000,"\004\140",1,0220060302};
        int t20600[]={t20625,04,026020600000,"\003\141",0};
        int t20577[]={t20755,021114,0133020000000,t20600,0};
        int t20552[]={0,0100,020040000000,"\007\140",1,0200060302};
        int t20527[]={t20552,010,022020000000,"\004\140",1,0200060302};
        int t20502[]={t20527,040,054000000000,"\006\141",0};
        int t20479[]={t20577,0150,0127020000000,t20502,0};
        int t20454[]={0,0100000,020040000000,"\020\140",1,0160060302};
        int t20429[]={t20454,0200000,022020000000,"\021\140",1,0160060302};
        int t20404[]={t20429,0100,020040000000,"\007\140",1,020060302};
        int t20379[]={t20404,010,022020000000,"\004\140",1,020060302};
        int t20354[]={t20379,024,054000000000,"\005\141\003\141",0};
        int t20331[]={t20479,0300134,0123020000000,t20354,0};
        int t20299[]={t20331,02000000000,0117020000000,"\035\000",0};
        int t20267[]={t20299,0200,0113020000000,"\010\000",0};
        int t20242[]={0,01000000,054000000000,"\023\000",1,0142000000};
        int t20217[]={t20242,0400000,056040000000,"\022\000",1,0100000300};
        int t20201[]={0,01400000,026021400000,t20217,0};
        int t20176[]={t20201,01400000,026021600000,"\022\000\023\000",0};
        int t20160[]={t20267,01400000,070240000000,t20176,0};
        int t20128[]={0,01000000,054000000000,"\023\000",1,0142000000};
        int t20103[]={t20128,0400000,056040000000,"\022\000",1,0100000300};
        int t20080[]={0,01400000,026021600000,t20103,0};
        int t20048[]={t20080,01400000,026021400000,"\022\000\023\000",0};
        int t20023[]={t20048,01000000000,054000000000,"\034\000",0};
        int t20000[]={t20160,01001400000,0107020000000,t20023,0};
        int t19977[]={0,03001721774,0100000000000,t20000,0};
```

```
/*  ^CD(5) */
        int t21072[]={0,0100000,020100000000,"\020\140",1,0260060302};
        int t21047[]={t21072,0200000,022020000000,"\021\140",1,0260060302};
        int t21022[]={t21047,0100,020100000000,"\007\140",1,060060302};
        int t20997[]={t21022,010,022020000000,"\004\140",1,060060302};
        int t20972[]={t20997,0400,054000000000,"\011\141",0};
        int t20956[]={0,0300510,0153020000000,t20972,0};
        int t20940[]={0,0300510,0100000000000,t20956,0};

#define LIT 0
int spval[]={LIT,0,
        LIT,0777777777777,
        LIT,0777777777776,
        LIT,01,
        LIT,02,
        LIT,00,
        LIT,07,
        LIT,06,
        LIT,03};

int roots[]={0,
t18850,
t19977,
t19977,
t19977,
t20940};

int attrib[]={0
,014
,011
,011
,011
,031};
#data high
int codtab[]={"","MOVEI","MOVE","DMOVE","JRST","JUMP","SKIP","MOVNI","HRLZI","HRROI","HRLOI"
,"DMOVN"};
#data high
int codcst[]={0,45,126,173,34,56,137,51,45,45,45
,208};
/*******        FILE:  C2.HDR           *******/
/*******                                *******/

/* Data declarations */

extern int macc, intc, strc, namc, swbc, datc;
extern int timpct, spcpct, stujob;
extern int trcflg, tracelev;
/* Miscellaneous constants
 */
#define PROINIT 1100    /* profiler autocall level, set high so after possible freeze */

#define DTIMPCT 1       /* Default 'time percentage' in cost */
#define SPCMLT 60               /* 'space percentage' multiplier, for compatibility */
#define DSPCPCT 3       /* 'space percentage' */
#define EVER (;;)

/* Number format
 *
 * |-----------------|----------------|
 * | TYPE |  |L|@|NDX|  PTR TO NAME   |
 * |---------------------------------|
 * | POS | SIZE |    |   OFFSET       |  or fullword VALUE if type CON
 * |-----------------|----------------|
 */
```

```
struct {
        char:9 ntype;
         char:3 ;
          char:1 nlit,nat;
           char:4 nndx;
            unsigned short nnam;
} ;

struct { char:6 npos,nsize, ; unsigned short noff; };

struct { int nvalue; };

#define NTYPE 0777000000000     /* Bit masks */
#define NLIT  0000040000000
#define NAT   0000020000000
#define NNDX  0000017000000
#define NNAM  0000000777777
#define NPOS  0770000000000
#define NSIZE 0007700000000
#define NOFF  0000000777777
#define NOFH  0000000400000  /* High order of offset */

/*
#define GTYPE(n) ((n)>>27)
#define GLIT(n) (((n)>>23)&1)
#define GAT(n) (((n)>>22)&1)
#define GNDX(n) (((n)>>18)&017)
#define GNAM(n) ((n)&0777777)
#define GPOS(n) ((n)>>30)
#define GSIZE(n) (((n)>>24)&077)
#define GOFF(n) ((n)&0777777)
#define GVALUE(n) ((n))

#define GTYPE_(n,v) ((n) = (n)&~NTYPE|((v)<<27))
#define GLIT_(n,v) ((n) = (n)&~NLIT|(((v)&1)<<23))
#define GAT_(n,v) ((n) = (n)&~NAT|(((v)&1)<<22))
#define GNDX_(n,v) ((n) = (n)&~NNDX|(((v)&017)<<18))
#define GNAM_(n,v) ((n) = (n)&~NNAM|((v)&0777777))
#define GPOS_(n,v) ((n) = (n)&~NPOS|((v)<<30))
#define GSIZE_(n,v) ((n) = (n)&~NSIZE|(((v)&077)<<24))
#define GOFF_(n,v) ((n) = ((n)&~NOFF)|((v)&0777777))
#define GVALUE_(n,v) ((n) = (v))
*/

#define GTYPE(n) ((n).ntype)
#define GLIT(n) ((n).nlit)
#define GAT(n) ((n).nat)
#define GNDX(n) ((n).nndx)
#define GNAM(n) ((n).nnam)
#define GPOS(n) ((n).npos)
#define GSIZE(n) ((n).nsize)
#define GOFF(n) ((n).noff)
#define GVALUE(n) ((n).nvalue)

#define GTYPE_(n, v) ((n).ntype = (v))
#define GLIT_(n, v) ((n).nlit = (v))
#define GAT_(n, v) ((n).nat = (v))
#define GNDX_(n, v) ((n).nndx = (v))
#define GNAM_(n, v) ((n).nnam = (v))
#define GPOS_(n, v) ((n).npos = (v))
#define GSIZE_(n, v) ((n).nsize = (v))
#define GOFF_(n, v) ((n).noff = (v))
#define GVALUE_(n, v) ((n).nvalue = (v))

#define CON 0           /* Number types */
```

```
#define DAT 1
#define REG 2
#define IMM 3
#define LBL 4
#define PAIR 5

/* Parse tree nodes
 *
 * |-----------------|
 * |      OP         |
 * |-----------------|
 * !     FLAGS       |
 * |-----------------|
 * | LEFTSON or VAL1 |
 * |-----------------|
 * |RIGHTSON or VAL2 |
 * |-----------------|
 *
 */

#define OP 0
#define FLAGS 1
#define PLEFT 2
#define PVAL1 2
#define PRIGHT 3
#define PVAL2 3
#define PARSESIZE 4

/* Return tree nodes
 *
 * |-----------------|
 * |     RVAL1       |
 * |-----------------|
 * |     RVAL2       |
 * |-----------------|
 * |     RTREE       |
 * |-----------------|
 * |     RNEXT       |
 * |-----------------|
 * |     RCOUNT      |
 * |-----------------|
 * |     RCODE       |
 *         .
 *         .
 *         .
 * |                 |
 * |-----------------|
 *
 */

#define RVAL1 0
#define RVAL2 1
#define RTREE 2
#define RNEXT 3
#define RCOUNT 4
#define RCODE 5
#define RSIZE 5

/* Translation tree format
 *
 * |----------------|
 * |    BROTHER     |
 * |----------------|
 * |     MODES      |
 * |----------------|   |-----|-----|-----|-----|-----||
```

```
 * |     FUNC       |    | KEY | ARG1| ARG2| ARG3|MODEA||
 * |----------------|    |-----|-----|-----|-----|-----||
 * |    RETNXT      |
 * |----------------|
 * |    COUNT       |
 * |----------------|
 * |     CODE       |
 *         .
 *         .
 *         .
 * |                |
 * |----------------|
 *
 */

#define BROTHER 0
#define MODES 1
#define FUNC 2
#define RETNXT 3
#define COUNT 4
#define CODE 5
#define TSIZE 5

#define KEY 0
#define ARG1 1
#define ARG2 2
#define ARG3 3
#define MODEA 4

/* Miscellaneous constants
 */

#define INFINITY 0377777777777
#define ATTIME 102              /* indirect reference time */
#define INDEX 6         /* indexing time */
#define MEMREF 20       /* Arbitraryish memory reference time */
#define EFFECT 1        /* Note 'E' must be first mode to TRANS */
#define LHALF 0777777000000
#define RHALF 0777777
#define FRSTL 0 /* as long as no one in the tables mentions it */


/* Ac usage map
 *
 * 2 bits per ac indexed from the right
 *
 * High order 4 bits are the ac# of highest ac used
 */

#define TEMPFREE 0
#define BOUND 1
#define TEMPINUSE 2
#define SPECIAL 3

#define GETAC(n) ((map>>((n)<<1))&3)
#define GETAC_(n,v) (map = ((map&~(3<<((n)<<1)))|((v)<<((n)<<1))))
#define GHIAC (map>>32)
#define GHIAC_(v) (map = (map&~0740000000000)|((v)<<32))

/* Routine structures
 */
struct { unsigned short lhalf, rhalf; };
struct { int word; };
     2  /*******        FILE:  CODE.C           *******/
     3  /*******                                *******/
```

```
 4  /* C Code Generator
 5   * Written by David H. Ackley and David W. Krumme
 6   *      Copyright (C) 1981 D. H. Ackley and D. W. Krumme
 7   *      Permission to copy without fee all or part of this
 8   *      program is granted provided that the copies are
 9   *      not made or distributed for direct commercial
10   *      advantage and that this copyright notice is included
11   *      in its entirety.
12   */
13
14  #old on
15  #space 100
16  #include "intops.hdr"    /* -- from TRANS */
17  #include "CC.HDR"
18  #include "C2.HDR"
19
20  int **ap,*rtree,rcost,rop,rval1,rval2,map;
21  int onecount=0, onebound=0;
22  int misseffect;
23  int *gbp,*gbt,gbmode,*gbatp,gbbcost, gbsucceed;
24
25
26  int apstack[800];
27
28  int constant;
29
30  #define MAXDEPTH 50
31  int depth,tvisits[MAXDEPTH],tnodes[MAXDEPTH];
32  int treeno,visits[MAXDEPTH],nodes[MAXDEPTH],worvisits[MAXDEPTH],worno[MAXDEPTH];
33  int totcost,totnodes,totvisits;
34
35  doexp(){
36          extern roots[];
37          int *p;
38
39          p = treein();
40
41          if (code(p,roots[p[OP]],EFFECT,apstack,INFINITY,1)) {
42                  totcost =+ rcost;
43                  deptree(p);
44                  emitcode(rtree);
45          }
46
47  }
48
49  code(argp,argt,mode,atp,maxcost,topflg) int *atp; {
50          extern spval[],roots[],(*fnctab[])(),attrib[];
51          register *p,*t,*atree;
52          int acost,aval1,aval2,amap,amaxc;
53          int bcost,bval1,bval2,bmap,bmaxc,btree;
54          int **sp,*np,i,*ip;
55          char *cp,*pc;
56          int lastsucceed;
57          int uflag;
58          gbp = p = argp;
59
60          if (topflg&&(p[FLAGS]&(1<<(mode-1)))) return(0);
61          p[FLAGS] =| 1<<(mode-1);
62
63          if (topflg==1) {
64                  if(depth<MAXDEPTH-1) ++visits[depth];
65          }
66          ++tracelev;
67          if(trcflg) tracef("Entering code(0%o,0%o,%d,0%o,%d,%d) op: %d",argp,argt,mod
e,atp,maxcost,topflg,(p = argp)[OP]);
```

```
    68          gbatp = atp;
    69          gbmode = mode;
    70          sp = ++ap;
    71          bcost = -1; btree = 0; bmaxc = maxcost;
    72          amap = map;
    73          lastsucceed = 10000;
    74          uflag = 0;
    75  again:  for (t = argt;gbt = t;t = t[BROTHER]) {
    76                  /* if(trcflg) tracef("Checking table 0%o",t); */
    77                  if (((1<<(mode-1))&t[MODES])==0) continue;
    78                  map = amap;
    79                  /* if(trcflg) tracef("Mode ok"); */
    80                  gbp = p; gbmode = mode; gbatp = atp; gbt = t;   gbbcost = bcost;
    81                  gbsucceed = ++lastsucceed;
    82                  acost = 0; amaxc = bmaxc;
    83                  for (cp = &t[FUNC],++cp;*cp&&(ip = cp)==&t[FUNC];++cp) {
    84                          /* if(trcflg) tracef("Pushing arg 0%o at 0%o",*cp,ap); */
    85                          *ap++ = adarg(*cp,sp,atp);
    86                  }
    87                  i = 0;
    88                  switch (*(cp = &t[FUNC])) {
    89              case 0:     np = p[PLEFT]; ++i; ++depth;
    90                          if(trcflg) tracef("Recurse Left");
    91                          goto recur;
    92              case 1:     np = p[PRIGHT]; ++i; ++depth;
    93                          if(trcflg) tracef("Recurse Right");
    94                          goto recur;
    95              case 2:     np = p;
    96                          if(trcflg) tracef("Recurse Self");
    97              recur:      if(trcflg) tracef("on Mode %d",(pc = &t[FUNC])[MODEA]);
    98                          if (!code(np,roots[np[OP]],(pc = &t[FUNC])[MODEA],sp,amaxc,2
-i)) {
    99                                  if(trcflg) tracef("Failed in recursion on op: %d in
mode %d",np[OP],(pc = &t[FUNC])[MODEA]);
   100                                  if (i) --depth;
   101                                  goto fail1;
   102                          }
   103                          if (i) --depth;
   104                          if(trcflg) tracef("Succeeds");
   105                          gbp = p; gbmode = mode; gbatp = atp; gbt = t;
   106                          break;
   107              default:    if(trcflg) tracef("Function %d", *(cp = &t[FUNC]));
   108                          if(!(*fnctab[*(cp= &t[FUNC])])()) {
   109                                  if(trcflg) tracef("Function failed");
   110                                  goto fail1;
   111                          }
   112                          if(trcflg) tracef("Function succeeds");
   113                  }
   114                  if(trcflg) tracef("Rval1: 0%o Rval2: 0%o Rtree: 0%o Rcost: %d",rval1
,rval2,rtree,rcost);
   115                  lastsucceed = 0;
   116                  ap = sp;
   117                  amaxc =- rcost; acost =+ rcost;
   118                  if (amaxc <= 0) {
   119                          if(trcflg) tracef("Failing on cost");
   120                          goto fail1;
   121                  }
   122                  atree = node(RSIZE+t[COUNT]);
   123                  for (i = 0;i<RSIZE;++i) atree[i] = 0;
   124                  atree[RVAL1] = rval1; atree[RVAL2] = rval2;
   125                  atree[RTREE] = rtree; atree[RCOUNT] = t[COUNT];
   126                  *ap = &atree[RVAL1];
   127                  if (t[COUNT]) {
   128                          if(trcflg) traces("generating %d code words",t[COUNT]);
   129                          gencode(t,atree);
```

```
   130                          amaxc =- rcost; acost =+ rcost;
   131                          if (amaxc<=0) {
   132                                  if(trcflg) traces("failing on code cost");
   133                                  goto fail2;
   134                          }
   135                  }
   136                  if (t[RETNXT]&LHALF) {
   137                          if(trcflg) tracef("Processing returns");
   138                          for (cp = t[RETNXT];*cp;cp =+2 ){
   139                                  if (*cp==mode) {
   140                                          aval1 = args(cp[1],1,sp,atp);
   141                                          aval2 = args(cp[1],2,sp,atp);
   142                                          if(trcflg) tracef("Mode match-- aval1: 0%o a
val2: 0%o",aval1,aval2);
   143                                          break;
   144                                  }
   145                                  if (*cp&~077) {
   146                                          if(trcflg) tracef("Calling function %d",*cp&
077);
   147                                          *++ap = adarg(cp[1],sp,atp);
   148                                          ++ap;
   149                                          (*fnctab[*cp&077])();
   150                                          ap =- 2;
   151                                  }
   152                          }
   153                  } else {
   154                          if(trcflg) tracef("Recursing on next: code(0%o,0%o,%d,0%o,%d
)",p,t[RETNXT],mode,atp,amaxc);
   155                          if (!code(p,t[RETNXT],mode,atp,amaxc,0)) {
   156                                  if(trcflg) tracef("Failing on next");
   157                                  goto fail2;
   158                          }
   159                          if(trcflg) tracef("Succeeds");
   160                          gbp = p; gbmode = mode; gbatp = atp; gbt = t;
   161                          amaxc =- rcost; acost =+ rcost;
   162                          atree[RNEXT] = rtree;
   163                          aval1 = rval1; aval2 = rval2;
   164                          if (amaxc<=0) {
   165                                  if(trcflg) tracef("Failing on cost");
   166                                  goto fail2;
   167                          }
   168                  }
   169                  if (bcost<0||acost<bcost) {
   170                          if(trcflg) traces("Bcost %d Acost %d Aval1 0%o Aval2 0%o Atr
ee 0%o",bcost,acost,aval1,aval2,atree);
   171                          if (btree) dectree(btree);
   172                          bcost = bmaxc = acost;
   173                          btree = atree;
   174                          bval1 = aval1; bval2 = aval2;
   175                          bmap = map;
   176                          if (stujob) {
   177                                  if(trcflg) tracef("Cancelling further search -- stud
ent job");
   178                                  goto win;
   179                          }
   180                  }
   181                  continue;
   182      fail2:      dectree(atree);
   183      fail1:      ap = sp;
   184           }
   185           if (topflg) {
   186                   switch (uflag) {
   187              case 0:      if (attrib[p[OP]]&FCOMMUTE) {
   188                                  if(trcflg) tracef("Commuting 0%o and 0%o on op %d",p
[PLEFT],p[PRIGHT],p[OP]);
```

```
   189                                  ++uflag;
   190                                  i = p[PLEFT];
   191                                  p[PLEFT] = p[PRIGHT];
   192                                  p[PRIGHT] = i;
   193                                  goto again;
   194                          }
   195             univ:        if (!(attrib[p[OP]]&FNOUNIV)) {
   196                                  if(trcflg) tracef("Searching Universal table on op %
d",p[OP]);
   197                                  uflag =+ 2;
   198                                  argt = roots[UNIVERSAL];
   199                                  goto again;
   200                          }
   201                          break;
   202             case 1:      if(trcflg) tracef("UnCommuting on op %d",p[OP]);
   203                          i = p[PLEFT];
   204                          p[PLEFT] = p[PRIGHT];
   205                          p[PRIGHT] = i;
   206                          goto univ;
   207                      }
   208          }
   209          if (bcost<0) {
   210                  if(trcflg) tracef("Failing to translate op %d in mode %d",p[OP],mode
);
   211                  map = amap;
   212                  --tracelev;
   213                  if (topflg) p[FLAGS] =& ~(1<<(mode-1));
   214                  return (0);
   215          }
   216  win:    rval1 = bval1; rval2 = bval2;
   217          rcost = bcost;
   218          rtree = btree;
   219          map = bmap;
   220          --ap;
   221          if(trcflg) tracef("Code returning: Rval1: 0%o Rval2: 0%o Rcost: %d Rtree: 0%
o",rval1,rval2,rcost,rtree);
   222          --tracelev;
   223          if (topflg) p[FLAGS] =& ~(1<<(mode-1));
   224          return(1);
   225  }
   226
   227
   228  args(c,n,sp,atp)  int **sp,**atp;{
   229          extern spval[];
   230          --n;
   231          switch (c>>5) {
   232     case 3:      return(sp[-(c&037)][n]);
   233     case 2:      return(atp[(c&037)-1][n]);
   234     case 1:
   235     case 0:      return(spval[c+n]);
   236          }
   237  }
   238  adarg(c,sp,atp) int **sp,*atp;  {
   239          extern spval[];
   240          switch (c>>5) {
   241     case 3:      return(sp[-(c&037)]);
   242     case 2:      return(atp[(c&037)-1]);
   243     case 1:
   244     case 0:      return(&spval[c]);
   245          }
   246  }
   247
   248  #optimize off
   249  #space
   250
```

```
   251
   252  #define MODLBL 1
   253
   254  cost(time,space){
   255          if(trcflg) traces("Charging %d for time=%d and space=%d",time*timpct+space*s
pcpct,time,space);
   256          return (time*timpct+space*spcpct);
   257  }
   258
   259  deptree(p) int *p; {
   260          extern attrib[];
   261          if (attrib[p[OP]]&FLEAF) goto leaf;
   262          if (attrib[p[OP]]&FUNOP) goto unop;
   263  binop:  deptree(p[PRIGHT]);
   264  unop:   deptree(p[PLEFT]);
   265  leaf:   denode(p,PARSESIZE);
   266          return;
   267  }
   268
   269  dectree(p) int *p; {
   270          if (p[RTREE]) dectree(p[RTREE]);
   271          if (p[RNEXT]) dectree(p[RNEXT]);
   272          denode(p,RSIZE+p[RCOUNT]);
   273          return;
   274  }
   275
   276  gencode(ttree,rtree) int *ttree,*rtree; {
   277          extern codcst[],spval[],codtab[];
   278          int t,*n;
   279          char *b;
   280          rcost = rval1 = rval2 = 0;
   281          for (t = 0;t<ttree[COUNT];++t) {
   282                  rtree[RCODE+t] = ttree[CODE+t];
   283                  b = &rtree[RCODE+t];
   284                  n = addr(b[2]);
   285                  if (n[1]==MODLBL) continue;
   286                  if(trcflg) traces("Opcode: %s",codtab[(*b<<7)+b[1]]);
   287                  rcost =+ cost(codcst[(*b<<7)+b[1]],1);
   288                  n = addr(b[4]);
   289                  if (n[0]&NLIT) rcost =+ cost(0,1);
   290                  if (n[0]&NAT) rcost =+ cost(ATTIME,0);
   291                  if (n[0]&NNDX) rcost =+ cost(INDEX,0);
   292                  if(trcflg) traces("Incremental cost of code: %d",rcost);
   293          }
   294  }
   295
   296  addr(c){
   297          extern spval[];
   298          switch(c>>5){
   299     case 3:      return(ap[-(c&037)]);
   300     case 2:      return(gbatp[(c&037)-1]);
   301     case 1:
   302     case 0:      return(&spval[c]);
   303          }
   304  }
   305
   306  int modstr[] = {"","","","A","L","LE","E","N","G","GE"};
   307
   308  #space 10
   309  emitcode(p) int *p; {
   310          extern codtab[];
   311          int i,m;
   312          register int *n;
   313          register char *b;
   314          *++ap = &p[RVAL1];
```

```
315          if (p[RTREE]) emitcode(p[RTREE]);
316          for (i = 0;i<p[RCOUNT];++i) {
317                  b = &p[RCODE+i];
318                  printf("%s",codtab[(*b<<7)+*++b]);
319                  if(trcflg) tracen("\t%s",codtab[(b[-1]<<7)+b[0]]);
320                  n = addr(*++b);
321                  if (GTYPE(n[0])!=CON) error("not CON in mod field");
322                  if(n[1]==MODLBL) {
323                          n = addr(b[2]);
324                          printf("%%%d:\r\n",GNAM(n[0]));
325                          if(trcflg) tracen("%%%d:\r\n",GNAM(n[0]));
326                          continue;
327                  }
328                  printf("%s ",modstr[n[1]]);
329                  if(trcflg) tracen("%s ",modstr[n[1]]);
330                  n = addr(*++b);
331                  if (GTYPE(n[0])!=REG&&GTYPE(n[0])!=CON)
332                          error("not CON or REG in ac field");
333                  printf("%o,",n[1]);
334                  if(trcflg) tracen("%o,",n[1]);
335                  n = addr(*++b);
336                  lstring(n[0],n[1]);
337                  printf("\r\n");
338                  if(trcflg) tracen("\r\n");
339          }
340          if (p[RNEXT]) emitcode(p[RNEXT]);
341          --ap;
342          denode(p,RSIZE+p[RCOUNT]);
343  }
344  #space
345
346  lstring(aval1,aval2){
347          register val1,val2;
348          extern int *names;
349
350          val1 = aval1; val2 = aval2;
351          if (val1&NLIT){
352                  printf("[");
353                  if(trcflg) tracen("[");
354          }
355          switch (GTYPE(val1)){
356      case CON:    emitoffset(val2,1);
357                   break;
358      case IMM:
359      case REG:    emitoffset(val2,0);
360                   break;
361      case PAIR:   printf("EXP ");
362                   emitoffset(((int*)GNAM(val1))[0], 1);
363                   putchar(',');
364                   emitoffset(((int*)GNAM(val1))[1], 1);
365                   break;
366      case LBL:
367      case DAT:    if (val2&NSIZE) {
368                           printf("POINT %d,",GSIZE(val2));
369                           if(trcflg) tracen("POINT %d,",GSIZE(val2));
370                   }
371                   if (val1&NAT) {
372                           printf("@");
373                           if(trcflg) tracen("@");
374                   }
375                   if (GTYPE(val1)==LBL) {
376                           printf("%%%d",GNAM(val1));
377                           if(trcflg) tracen("%%%d",GNAM(val1));
378                   }
379                   else    if (GNAM(val1)) {
```

```
   380                                  printf("%s", (char*)&names[GNAM(val1)<<1]);
   381                                  if(trcflg) tracen("%s", (char*)&names[GNAM(val1)<<1]
);
   382                  }
   383                  if (val2&NOFF) {
   384                          emitoffset(val2,0);
   385                  }
   386                  if (val1&NNDX) {
   387                          printf("(%o)",GNDX(val1));
   388                          if(trcflg) tracen("(%o)",GNDX(val1));
   389                  }
   390                  if (val2&NSIZE) {
   391                          printf(",%d",35-GPOS(val2));
   392                          if(trcflg) tracen(",%d",35-GPOS(val2));
   393                  }
   394                  break;
   395          }
   396          if (val1&NLIT) {
   397                  printf("]");
   398                  if(trcflg) tracen("]");
   399          }
   400  }
   401
   402  emitoffset(n,full) {
   403          if (!full) {
   404                  n =& NOFF;
   405                  if (n&NOFH) n =| LHALF;
   406          }
   407          if (n<0) {
   408                  printf("-%o",-n);
   409                  if(trcflg) tracen("-%o",-n);
   410          } else  {
   411                  printf("+%o",n);
   412                  if(trcflg) tracen("+%o",n);
   413          }
   414  }
/*******        FILE:  FUNCS.C          *******/
/*******                                *******/
#old on
#include "C2.HDR"

#include "modes.hdr"    /* from TRANS */
#include "intops.hdr"   /* from TRANS */

extern int **ap,*rtree,rcost,rval1,rval2,map,lineno;
extern int *gbp,*gbt,gbmode,*gbatp,gbbcost;
extern int label,retlbl;

#define JUMPMODES ((1<<(MODE_J-1))|(1<<(MODE_K-1))|(1<<(MODE_KA-1)))

#include "TAB.HDR"


zero() {
        rcost = rtree = rval1 = rval2 = 0;
}

IFFUNC() {
        extern processor;
        NULL();
        switch(rval2) {
        case NOTRANS:
                if(gbbcost>=0) return(0);
                return(1);
        case NOJUMPENT:
```

```
                if(gbp[FLAGS]&JUMPMODES) return(0);
                return(1);
        case NOVAENT:
                if(gbp[FLAGS]&(1<<(MODE_VA-1))) return(0);
                return(1);
        case NOVPENT:
                if(gbp[FLAGS]&(1<<(MODE_VP-1))) return(0);
                return(1);
        case KL10:
                if(processor == 'L') return 1;
                return 0;
        }
        return(1);
}

ELSEFUNC() {
        extern gbsucceed;

        NULL();
        if(gbsucceed <= rval2) return(0);
        return(1);
}

EFFECTM() {
        zero();
        if (gbmode==1) return(1);
        return(0);
}

NOEFFECT() {
        register int i = 0;
        extern attrib[];
        extern misseffect;

        if(gbbcost >= 0) return(0);
        NULL();
        switch(rval2) {
        case 0:
                i = FLEAF;
                break;
        case 1:
                i = FUNOP;
                break;
        case 2:
                i = FBINOP;
                break;
        }
        if( !(attrib[gbp[OP]] & i)) return 0;
        misseffect = 1;
        return 1;
}

CMPL() {
        NULL();
        rval2 = ~rval2;
        return(1);
}
ISZ() {
        NULL();
        if (rval2==0) return(1);
        return(0);
}

ISN() {
        NULL();
```

```
        if (rval2==-1) return(1);
        return(0);
}

NEGATE() {
        NULL();
        rval2 = -rval2;
        return(1);
}

ISO() {
        NULL();
        if (rval2==1) return(1);
        return(0);
}


MAKLIT(){
        NULL();
        rval1 =| NLIT;
        return(1);
}


INCR(){
        NULL();
        ++rval2;
        return (1);
}

INVERT(){
        NULL();
        rval2 = 13-rval2;
        return(1);
}

CONDMAP() {
        zero();
        rval1 = ap[-2][0];
        rval2 = ap[-2][1];
        switch(ap[-1][1]) {
        case NE:
        case GT:
                break;
        case EQ:
        case LE:
                rval2 = 13 - rval2;
                break;
        case GE:
                rval2 = A;
                break;
        case LT:
                rval2 = N;
                break;
        }
        return 1;
}

WOEQ() {
        NULL();
        switch(rval2) {
        case 05: case 011:      /* convert LE and GE to LT and GT */
                --rval2;
        }
        return(1);
```

```
}

WEQ() {
        NULL();
        switch(rval2) {
        case 04: case 010:      /* convert LT and GT to LE and GE */
                ++rval2;
        }
        return(1);
}

NULL(){
        rcost = rtree = 0;
        rval1 = ap[-1][0];
        rval2 = ap[-1][1];
        return (1);
}

ISAC(){
        NULL();
        if (GTYPE(rval1)==REG) return(1);
        if (GTYPE(rval1)==CON && rval2>=0 && rval2<=017) {
                GTYPE_(rval1,REG);
                return (1);
        }
        return (0);
}

NEWLABEL(){
        zero();
        GTYPE_(rval1,LBL);
        GNAM_(rval1,++label);
        return (1);
}


NOTAC(){return (!ISAC());}

MGI(){
        int lv1,lv2;
        NULL();
        lv1 = ap[-2][0]; lv2 = ap[-2][1];
        stdize(&lv1);
        stdize(&rval1);
        if ((GTYPE(lv1)!=DAT)||(GTYPE(rval1)!=DAT)) return(0);
        if (GAT(lv1)||GAT(rval1)) return(0);
        if (GNDX(lv1))
                if (GNDX(rval1)) return(0);
                else GNDX_(rval1,GNDX(lv1));
        if (GNAM(lv1))
                if (GNAM(rval1)||GOFF(rval2)) return(0);
                else GNAM_(rval1,GNAM(lv1));
        GOFF_(rval2,GOFF(lv2)+GOFF(rval2));
        return(1);
}

stdize(argp) {
        register *p;
        p = argp;
        if (GTYPE(p[0])==IMM || (GTYPE(p[0])==CON &&
                        ((p[1]&0777777000000)==0 || (-p[1]&0777777000000)==0))) {
                p[1] &= 0777777;
                p[0] = 0;
                GTYPE_(p[0],DAT);
        }
```

```
}

AT(){
        rcost = rtree = 0;
        rval1 = ap[-2][0];
        rval2 = ap[-2][1];
        if (GTYPE(rval1)==REG||GTYPE(rval1)==CON&&GVALUE(rval2)>=1&&GVALUE(rval2)<=017) {
                rval1 = 0;
                GTYPE_(rval1,DAT);
                GNDX_(rval1,rval2);
                rval2 = 0;
                return (1);
        }
        if (GTYPE(rval1)==DAT && !GAT(rval1) && ap[-1][1]!=LHBAD) {
                GAT_(rval1,1);
                return(1);
        }
        if (GTYPE(rval1)==CON) warn("AT called with type CON");
        return(0);
}

FCINFO() {
        extern infolbl;

        zero();
        if(ap[-2][1]==0) {
                GTYPE_(rval1,CON);
                GVALUE_(rval2,ap[-1][1]>>4);
        } else {
                GTYPE_(rval1,LBL);
                GNDX_(rval1,ap[-1][1]&017);
                GNAM_(rval1,infolbl);
        }
        return(1);
}


MAKARG(){
        zero();
        GTYPE_(rval1,DAT);
        GNDX_(rval1,017);
        GOFF_(rval2,-ap[-1][1]);
        return (1);
}

DIFFER(){ return(!IDENT()); }

IDENT(){
        NULL();
        if(ap[-2][0]==rval1&&ap[-2][1]==rval2) return(1);
        return(0);
}

ACSIZE() {
        extern attrib[];

        zero();
        GTYPE_(rval1, CON);
        if(attrib[gbp[OP]]&FTWOACS) GVALUE_(rval2, -2);
        else GVALUE_(rval2, -1);
        return(1);
}

X(){
        rcost = rtree = 0;
```

```
        rval1 = gbp[PVAL1];
        rval2 = gbp[PVAL2];
        return (1);
}

HLZ(){
        NULL();
        if ((rval2>>18)==0) {
                return(1);
        }
        return(0);
}

HRZ(){
        NULL();
        if ((rval2&0777777)==0) {
                rval2 =>> 18;
                return (1);
        }
        return (0);
}

HLO(){
        NULL();
        if ((rval2>>18)==0777777) {
                rval2 =& 0777777;
                return (1);
        }
        return (0);
}

HRO(){
        NULL();
        if ((rval2&0777777)==0777777) {
                rval2 =>> 18;
                return (1);
        }
        return (0);
}

BPSPLIT() {     /* avoid making a literal with an index reg. in it */

        NULL();
        switch(ap[-2][1]) {
        case NOSPLIT:
                if(GNDX(rval1)==0) return(1);
                break;
        case SPLITADDR:
                if(GNDX(rval1) != 0) {
                        rval2 &= RHALF;
                        return(1);
                }
                break;
        case SPLITPS:
                if(GNDX(rval1) != 0) {
                        rval1 = 0;
                        GTYPE_(rval1,CON);
                        GVALUE_(rval2,(rval2>>18)&RHALF);
                        return(1);
                }
                break;
        }
        return(0);
}
```

```
G1() {  /* reserve 0 and 1 */
        int i;
        extern onecount;

        if(onecount > 1) return(0);
        NULL();
        GTYPE_(rval1,REG);
        i = ap[-2][1];
        if(i >= 0 && i == rval2) return(1);
        if(GETAC(1) == BOUND) error("AC-1 bound twice");
        if(i==-1 || i<=-2 && rval2==0) {
                GETAC_(0, BOUND);
                GETAC_(1, BOUND);
                return(1);
        }
        return(0);
}

G(){
        int n,m,am,block;
        rcost = rtree = 0;
        n = ap[-1][1];
        block = 0;
        switch(n) {
        case -4:
                ++block;
        case -3:
        case -2:
                ++block;
        case -1:
                for (m = 2;m<=017;++m) {
                        for(am=m; am<=m+block; ++am)
                                if(am>017 || GETAC(am)!=TEMPFREE) continue 2;
                        for(am=m; am<=m+block; ++am) {
                                GETAC_(am,TEMPINUSE);
                                if (am>GHIAC) GHIAC_(am);
                        }
                        rval1 = 0;
                        GTYPE_(rval1,REG);
                        rval2 = m;
                        return (1);
                }
                warn("running out of registers");
                return(0);
        }
        switch (GETAC(n)) {
   case TEMPFREE: GETAC_(n,TEMPINUSE); break;
   case BOUND:  break;
   case TEMPINUSE:      break;
   case SPECIAL:
                if(n==0||n==1) break; /* used in function calls */
                warn("attempt to G(et) special purpose register: %o",n);
                return(0);
        }
        rval1 = 0;
        GTYPE_(rval1,REG);
        rval2 = n;
        return (1);
}

GETFOL(){
        NULL();
        if (rval2<0) {
                rval2 = -2;
                return(1);
```

```
        }
        if (GETAC(rval2+1)==TEMPFREE) {
                GETAC_(rval2+1,TEMPINUSE);
                return(1);
        }
        if(rval2 == 0) return(1);
        GTYPE_(rval1,CON);
        rval2 = -3;
        return(1);
}

GETPRE(){
        NULL();
        if (rval2<0) {
                rval2 = -2;
                return(1);
        }
        if (GETAC(rval2-1)==TEMPFREE){
                GETAC_(--rval2,TEMPINUSE);
                return(1);
        }
        if(rval2 == 1) {
                rval2 = 0;
                return(1);
        }
        GTYPE_(rval1,CON);
        rval2 = -3;
        return (1);
}

F(){
        int n;
        NULL();
        if (GTYPE(rval1)==REG) {
                n = rval2;
        } else {
                if ((GTYPE(rval1)==DAT||GTYPE(rval1)==LBL)&&GNDX(rval1))

                        n = GNDX(rval1);
                else    return;
        }
        if (GETAC(n)==TEMPINUSE) GETAC_(n,TEMPFREE);
        if(n==0 || n==1) {
                if(GETAC(1) == BOUND) {
                        GETAC_(0, SPECIAL);
                        GETAC_(1, SPECIAL);
                }
        }
        return;
}

F2() {
        int n;

        F();
        if(GTYPE(rval1)==REG) {
                n = rval2 + 1;
                if(GETAC(n)==TEMPINUSE) GETAC_(n,TEMPFREE);
        }
}

FNEW() {

        if(ap[-2][1] > -2) F();
}
```

```
#include "fnctab.c"
```