```
CODE F+
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   SHORT_REAL 4 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F– (F1 F2 – F3)

Leave F3, the floating-point difference of F1 minus F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The difference of F1 minus F2.

Example of Use:

```
... Sum F@ PI F– Sum F! ...
```

This code fragment would subtract pi from the value held in Sum.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Subtract the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F–
   AX BP MOV
   BP SP MOV
   SHORT_REAL 4 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F* (F1 F2 – F3)

Leave F3, the floating-point product of F1 times F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The product of F1 times F2.

Example of Use:

```
... Sum F@ 10. F *Sum F! ...
```

This code fragment would multiply the value held in Sum by ten.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Multiply the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F*
   AX BP MOV
   BP SP MOV
   SHORT_REAL 4 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F/ (F1 F2 – F3)

Leave F3, the floating-point quotient of F1 divided by F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The quotient of F1 divided by F2.

Example of Use:

```
... Sum F@ 10. F/ Sum F! ...
```

This code fragment would divide the value held in Sum by ten.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decre-

ment the stack with the pops. Divide the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F/
   AX BP MOV
   BP SP MOV
   SHORT__REAL 4 [BP+#] FLD
   SHORT__REAL 0 [BP+#] FDIV
   BX POP BX POP
   SHORT__REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

FNEGATE (F1 – F2)

Leave F2, F1 with the opposite sign.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 with it's sign reversed.

Example of Use:

```
... Sum F@ PI FNEGATE F* ...
```

This code fragment would multiply Sum by negative pi.

Algorithm: Move the number from the 8088 data stack to the 8087. Negate the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE FNEGATE
   AX BP MOV
   BP SP MOV
   SHORT__REAL 0 [BP+#] FLD
   FCHS
   SHORT__REAL 0 [BP+#] FSTP
```

```
   WAIT BP AX MOV
NEXT
```

FP->INT (F - N)

Leave N, the integer equivalent of F.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (N) The integer equivalent of F.

Example of Use:

```
... PI INT->FP . ...
```

This code fragment would print a three on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Decrement the stack with a pop. Move the result back to the 8088 data stack, using a 8087 integer store.

Suggested Extensions: None.

Definition:

```
CODE FP->INT
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   BX POP
   WORD_INTEGER 2 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

INT->FP (N - F)

Leave F, the floating-point equivalent of N.

Stack on Entry: (N) An integer number.

Stack on Exit: (F) The floating-point equivalent of N.

Example of Use:

```
... 1 INT->FP 3 INT->FP F/ R. ...
```

This code fragment would print a the floating-point representation of one-third on the display.

Algorithm: Move the number from the 8088 data stack to the 8087 with an integer load. Increment the stack with a push. Move the result back to the 8088 data stack, using a 8087 real store.

Suggested Extensions: None.

Definition:

```
CODE INT->FP
   AX BP MOV
   BX PUSH
   BP SP MOV
   WORD_INTEGER 2 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

FABS (F1 – F2)

Leave F2, the absolute value of F1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The absolute value of F1.

Example of Use:

```
... TEMPERATURE F@ FABS R. ...
```

This code fragment would print the absolute value of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the absolute value of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE FABS
   AX BP MOV
```

```
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FABS
   SHORT_REAL 0 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F*10 (F1 – F2)

Leave F2, F1 multiplied by ten. Used in floating-point I/O.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 times ten.

Example of Use:

```
... PI F*10 R. ...
```

This code fragment would print 31.4157 on the display. Algorithm: Use the word F* , place ten on the stack by using the double length equivalent.

Suggested Extensions: None.

Definition:

```
: F*10 16672, F* ;
```

F/10 (F1 – F2)

Leave F2, F1 divided by ten. Used in floating-point I/O.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 divided by ten.

Example of Use:

```
... PI F/10 R. ...
```

This code fragment would print .314157 on the display.

Algorithm: Use the word F/, place ten on the stack by using the double-length equivalent.

Suggested Extensions: None.

Definition:

```
: F/10 16672, F/ ;
```

F= (F1 F2 – B)

Compare two floating-point numbers, checking for equality.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F= ...
```

This code fragment would leave a true flag on the stack if the values in Sum and TOTAL were equal.

Algorithm: Move the arguments to the 8087 registers. Pop them off the 8088 stack. Compare the numbers on the 8087, popping them off its stack. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
0 VARIABLE 18087

CODE F=
   CX BP MOV
   BP SP MOV
   SHORT__REAL 0 [BP+#] FLD
   SHORT__REAL 4 [BP+#] FCOMP
   BX POP BX POP BX POP
   18087 ] FSTSTATUSW DX 0 # MOV
   WAIT AH 18087 1+ ] MOV SAHF ZIF
      DX DEC
```

```
  /ENDIF
  6 [BP+#] DX MOV
  BP CX MOV
NEXT
```

F0= (F – B)

Compare a floating-point number to zero.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is equal to zero.

Example of Use:

```
... Sum F@ F0= ...
```

This code fragment would leave a true flag on the stack if the value in sum was zero.

Algorithm: Move the argument to the 8087 registers. Pop it off the 8088 stack. Compare it to zero on the 8087. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
CODE F0=
  CX BP MOV
  BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  FTST
  BX POP
  [8087 ] FSTSTATUSW DX 0 # MOV
  WAIT AH [8087 1+ ] MOV SAHF ZIF
    DX DEC
  /ENDIF
  2 [BP+#] DX MOV
  BP CX MOV
  0 ST FSTP
NEXT
```

F< (F1 F2 – B)

Compare two floating-point numbers, checking for a less than condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is less than F2.

Example of Use:

```
... Sum F@ TOTAL F@ F< ...
```

This code fragment would leave a true flag on the stack if the value in Sum was less than the value in TOTAL.

Algorithm: Move the arguments to the 8087 registers. Pop them off the 8088 stack. Compare the numbers on the 8087, popping them off its stack. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
CODE F<
   CX BP MOV
   BP SP MOV
   SHORT__REAL 4 [BP+#] FLD
   SHORT__REAL 0 [BP+#] FCOMP
   BX POP BX POP BX POP
   f8087 ] FSTSTATUSW DX 0 # MOV
   WAIT AH f8087 1+ ] MOV SAHF CIF
      DX DEC
   /ENDIF
   6 [BP+#] DX MOV
   BP CX MOV
NEXT
```

F2DUP (F1 F2 – F1 F2 F1 F2)

Duplicate the top two floating-point numbers.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F1) A copy of F1.
(F2) A copy of F2.
(F1) A copy of F1.
(F2) A copy of F2.

Example of Use: See the words defined below.

Algorithm: Loop through the top four words on the stack, rolling them each to the top.

Suggested Extensions: None.

Definition:

```
: F2DUP 4 0 DO 4 PICK LOOP ;
```

F> (F1 F2 – B)

Compare two floating-point numbers, checking for a greater than condition.

Stack on Entry: (F1) A floating-point number.
(F1(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is greater than F2.

Example of Use:

```
... Sum F@ TOTAL F@ F> ...
```

This code fragment would leave a true flag on the stack if the value in Sum was greater than the value in TOTAL.

Algorithm: Swap the arguments and call F<.

Suggested Extensions: None.

Definition:

```
: F> 2SWAP F< ;
```

F<= (F1 F2 – B)

Compare two floating-point numbers, checking for a less than or equal condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is less than or equal F2.

Example of Use:

```
... Sum F@ TOTAL F@ F<= ...
```

This code fragment would leave a true flag on the stack if the value in Sum was less than or equal the value in TOTAL.

Algorithm: Duplicate the arguments and use F< and F=. OR their results.

Suggested Extensions: None.

Definition:

```
: F<= F2DUP F< >R F= R> OR ;
```

F>= (F1 F2 – B)

Compare two floating-point numbers, checking for a greater than or equal condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is greater than or equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F>= ...
```

This code fragment would leave a true flag on the stack if the value in Sum was greater than or equal to the value in TOTAL.

Algorithm: Duplicate the arguments and use F> and F=. OR their results.

Suggested Extensions: None.

Definition:

```
: F>= F2DUP F> >R F= R> OR ;
```

F<> (F1 F2 – B)

Compare two floating-point numbers, checking for equality.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is not equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F<> ...
```

This code fragment would leave a true flag on the stack if the values in Sum and TOTAL were not equal.

Algorithm: Not the result of F=.

Suggested Extensions: None.

Definition:

```
: F<> F= NOT ;
```

FP->DINT (F – D)

Leave D, the double length integer equivalent of F.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (D) The double length integer equivalent of F.

Example of Use:

```
... PI FP->DINT D. ...
```

This code fragment would print a three on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Move the result back to the 8088 data stack, using an 8087 integer store.

Suggested Extensions: None.

Definition:

```
CODE FP->DINT
   AX BP MOV
   BP SP MOV
   SHORT__REAL 0 [BP+#] FLD
   SHORT__INTEGER 0 [BP+#] FSTP
   BP AX MOV WAIT
   CX POP BX POP CX PUSH BX PUSH
NEXT
```

DINT->FP (D - F)

Leave F, the floating-point equivalent of D.

Stack on Entry: (D) A doublelength integer.

Stack on Exit: (F) The floating-point equivalent of D.

Example of Use:

```
... 23, DINT->FP R. ...
```

This code fragment would print a 23 on the display.

Algorithm: Move the number from the 8088 data stack to the 8087 using an integer load. Move the result back to the 8088 data stack, using an 8087 real store.

Suggested Extensions: None.

Definition:

```
CODE DINT->FP
   AX BP MOV
   BP SP MOV
   CX POP BX POP CX PUSH BX PUSH
   SHORT__INTEGER 0 [BP+#] FLD
   SHORT__REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

FTRUNC (F1 - F2)

Leave F2, the integer portion of F1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The whole number portion of F1.

Example of Use:

```
... 22.75 FTRUNC R. ...
```

This code fragment would print a 22 on the display.

Algorithm: Move the number from the 8088 data stack to the 8087.Change the 8087 rounding mode to truncate the number when it is rounded to zero. Restore the 8087 rounding mode and move the result back to the 8088 data stack, using a 8087 real store.

Suggested Extensions: None.

Definition:

```
: 0. 0, ;

CODE FTRUNC
   f8087 ] FSTCW WAIT
   BX f8087 ] MOV AX BX MOV
   AX 3072 # Or f8087 ] AX MOV
   f8087 ] FLDCW
   CX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FRNDINT f8087 ] BX MOV
   SHORT_REAL 0 [BP+#] FSTP
   f8087 ] FLDCW WAIT
   BP CX MOV
NEXT
```

F.R (F N –)

Print F on the display, with N digits after the decimal point.

Stack on Entry: (F) A floating-point number.
(N) The number of digits to print after the decimal point.

Stack on Exit: Empty.

Example of Use:

```
... Sum 8 F.R ...
```

This code fragment would print the value of sum with eight digits after the decimal point.

Algorithm: First, print the integer portion of the number. Divide by ten, building the output string in the pad. Print it when zero is reached. Then loop on the trailing digits, print one at a time until the count provided is exhausted or the number is zero.

Suggested Extensions: None.

Definition:

```
-219857153, FCONSTANT .49->
0 VARIABLE PPOS
: F.R >R 2DUP 0. F< IF
    FNEGATE ." -"
  ENDIF
  PAD 30 + PPOS ! 2DUP BEGIN
    FTRUNC 2DUP F/10 FTRUNC 2DUP >R >R
    F*10 F-
    FP->INT 48 + PPOS @ C!
    -1 PPOS +! R> R> 2DUP FP->INT
  0= UNTIL PPOS @ 1+ DUP PAD 30 + -
  NEGATE 1+ TYPE ." ." 2DROP
  2DUP FTRUNC F- R> 0 DO
    F*10 2DUP FTRUNC FP->INT 48 + EMIT
    2DUP FTRUNC F- 2DUP .49-> F+ FP->INT
    0= IF
      LEAVE
    ENDIF
  LOOP 2DROP SPACE ;
```

F. (F -)

Print F on the display, with six digits after the decimal point.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

... Sum F. ...

This code fragment would print the value of sum with six digits after the decimal point.

Algorithm: Call F.R with a six.

Suggested Extensions: None.

Definition:

```
: F. 6 F.R ;
```

FE.R (F N –)

Print F on the display in scientific notation, with N digits after the decimal point.

Stack on Entry: (F) A floating-point number.
(N) The number of digits to print after the decimal point.

Stack on Exit: Empty.

Example of Use:

```
... Sum 8 FE.R ...
```

This code fragment would print the value of Sum in scientific notation with eight digits after the decimal point.

Algorithm: Normalize the number between zero and one by dividing by 10^6 or 10^−6 until it is within range of 10^6 > X > 10^6. Then, continue the normalization by tens. When it is complete, print out the number, and then the exponent.

Suggested Extensions: None.

Definition:

```
0 VARIABLE TX
16256. FCONSTANT 1.
-858964532. FCONSTANT .1
603998580. FCONSTANT #E6
935146886. FCONSTANT #E-6
: F/#E6 #E6 F/ ;
: F*#E6 #E6 F* ;
: FE.R TX 0SET >R 2DUP 0. F< IF
     ." -" FNEGATE
  ENDIF 2DUP F0= NOT IF
    2DUP #E6 F> = IF BEGIN
```

```
        F/#E6 6 TX +! 2DUP #E6 F<
      UNTIL ELSE
        2DUP #E-6 F< IF BEGIN
            F*#E6 -6 TX +! 2DUP #E-6 F>
      UNTIL ENDIF ENDIF
      2DUP 1. F>= IF BEGIN
        F/10 1 TX +! 2DUP 1. F<
      UNTIL ELSE
      2DUP .1 F<= IF BEGIN
        F*10 -1 TX +! 2DUP .1 F>=
    UNTIL ENDIF ENDIF
    R> F.R ." e" TX ?
  ELSE
    R> DROP 2DROP ." 0.0e0 "
  ENDIF ;
```

FE. (F –)

Print F on the display in scientific notation, with six digits after the decimal point.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

```
... Sum FE. ...
```

This code fragment would print the value of Sum in scientific notation with six digits after the decimal point.

Algorithm: Use FE.R with a value of six.

Suggested Extensions: None.

Definition:

```
: FE. 6 FE.R ;
```

R. (F –)

Print F on the display, in an appropriate form.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

```
... Sum F@ R. ...
```

This code fragment would print the value of Sum on the display.

Algorithm: Use FE. if F is a very large or very small number; otherwise, use F. Check for a possible zero to avoid zero in scientific notation.

Suggested Extensions: None.

Definition:

```
: R. 2DUP 0= SWAP 0= AND IF
     2DROP ." 0.0 " EXIT
   ENDIF
   2DUP FABS
   2DUP #E6 F> >R #E-6 F< R> OR IF
     FE.
   ELSE
     F.
   ENDIF ;
```

FNUM (A – (F) B)

Attempt to convert the text string at A to a floating-point number.

Stack on Entry: (A) A text string as from the word WORD.

Stack on Exit: (B) A Boolean flag, true if a number could be converted.
(F) The converted number if the flag is true.

Example of Use:

```
: GET-A-REAL QUERY >IN 0SET BL WORD FNUM ;
```

GET-A-REAL would input a line of text and them attempt to convert the first part of the line to a real number.

Algorithm: Strip away a negative sign if it is present. Then attempt to convert the integer portion of the number. Convert it to floating-point. Look for a

decimal point. If found, convert the numbers after it to floating-point by dividing by ten. Add this to the whole number portion. Look for an 'E' that would indicate an exponent. If an 'E' is found, strip away a negative sign if present. Convert the exponent to integer and apply it to the number obtained. Leave a true flag if all these steps complete successsfully; false otherwise.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE INS
0 CVARIABLE INXS
0. FVARIABLE DIVAMT
16672, FCONSTANT 10.
: 0-9? DUP 48 >= SWAP 57 <= AND ;
: EXPMOD DROP INXS C@ IF
     0 DO F/10 LOOP
   ELSE
     0 DO F*10 LOOP
   ENDIF ;
0 CVARIABLE .GOT
: FNUM
   10. DIVAMT F! INS C0SET INXS C0SET
   .GOT C0SET DUP 1+ C@ 45 = IF
     1+ INS C1SET
   ENDIF 0, ROT >BINARY >R DINT->FP R>
   DUP C@ 46 = IF
     .GOT C1SET BEGIN
       1+ DUP C@ 0-9?
     WHILE
       DUP C@ 48 - INT->FP DIVAMT F@ F/
       ROT >R F+ R>
       DIVAMT DUP >R F@ F*10 R> F!
     REPEAT
   ENDIF
   DUP C@ 69 = IF
     DUP 1+ C@ 45 = IF
       1+ INXS C1SET
     ENDIF 0, ROT >BINARY >R EXPMOD R>
   ENDIF
   INS C@ IF >R FNEGATE R> ENDIF
   C@ BL <> .GOT C@ 0= OR
   IF 2DROP 0 ELSE -1 ENDIF ; ->

( Make floating-point input possible in Atila)
: FINP BL WORD FNUM ;
```

```
: NNUM DUP FNUM IF ROT DROP
      ELSE [ V.NUM @ ] LITERAL
      EXECUTE ENDIF ;
' NNUM V.NUM !
```

SQRT (F1 – F2)

Leave F2,square root of F1.

Argument Range: 0 <= F1 <= Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The square root of F1.

Example of Use:

```
... TEMPERATURE F @ SQRT R. ...
```

This code fragment would print the square root of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the square root of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE SQRT
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FSQRT
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

LN (F1 – F2)

Leave F2, the natural log of F1.

Argument Range: –Infinity <= Q F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The natural log of F1.

Example of Use:

```
... TEMPERATURE F@ LN R. ...
```

This code fragment would print the natural log of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the natural log of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

```
Definition: CODE LN
   AX BP MOV BP SP MOV
   LN2FLD
   SHORT_REAL 0 [BP+#] FLD
   FYL2X
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

LOG (F1 – F2)

Leave F2, the base 10 log of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The base 10 log of F1.

Example of Use:

```
... TEMPERATURE F@ LOG R. ...
```

This code fragment would print the base 10 log of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Deter-

mine the base 10 log of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE LOG
   AX BP MOV BP SP MOV
   LG2FLD
   SHORT_REAL 0 [BP+#] FLD
   FYL2X
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

2∧X

Assembler subroutine to calculate 2 to an arbitray power.

Algorithm: Decompose 2∧X to 2∧X = (2∧I) * (2∧F), where I is the integer portion of X and F is the fractional portion. If f is nonnegative, use the F2XM1 instruction to calculate its value. If it is negative, use the identity 2∧f − 1 = −(2∧−f − 1 / (2∧−F)) to calculate its value. When finished scale, that result by I to obtain the final value. 2∧X operates on the top 8087 stack entry.

Suggested Extensions: None.

Definition:

```
SUBROUTINE 2∧X

   0 ST 0 ST FLD FRNDINT 0 ST 0 ST FLD
   2 ST 0 ST FSUB ANDPOP REVERSE
   1 ST FXCH FTST f8087 ] FSTSTATUSW
   WAIT AH f8087 1+ ] MOV SAHF CIF
      FCHS F2XM1 0 ST 0 ST FLD
      1FLD 1 ST 0 ST FADD ANDPOP
      1 ST 0 ST FDIV ANDPOP REVERSE FCHS
   /ELSE
      F2XM1
   /ENDIF
   1FLD 1 ST 0 ST FADD ANDPOP
   FSCALE
```

```
  1 ST FXCH 0 ST 0 ST FSTP
  WAIT
END-SUB
```

EXP (F1 – F2)

Leave F2, e raised to the F1 power.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) e raised to the F1 power.

Example of Use:

```
... TEMPERATURE F@ EXP R. ...
```

This code fragment would print e raised to the value of the variable TEMPERATURE on the display.

Algorithm: Use the identity Y^X = 2(X*LN(Y)).

Suggested Extensions: None.

Definition:

```
CODE EXP
   DX BP MOV BP SP MOV
   L2EFLD
   SHORT_REAL 0 [BP+#] FMUL
   2^X CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

^(F1 F2 – F3)

Leave F3, F1 raised to the F2 power.

Argument Range: –Infinity <= F1 <= +Infinity.
–Infinity <= F2 <= +Infinity.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) F1 raised to the F2 power.

Example of Use:

```
... TEMPERATURE F@ 3.^R. ...
```

This code fragment would print the value of the variable TEMPERATURE raised to the third power on the display.

Algorithm: Use the identity Y^X = 2(X*LN(Y)).

Suggested Extensions: None.

Definition:

```
CODE ^
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   SHORT_REAL 4 [BP+#] FLD
   FYL2X
   BX POP BX POP 2^X CALL
   SHORT_REAL 4 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

[FPTAN]

Assembler subroutine that will execute the 8087 FPTAN instruction on any number.

Algorithm: The 8087 instruction FPTAN is used in all the trigonometric calculations. FPTAN only operates on values greater than zero and less than pi/4. This routine reduces the number found on the top of the 8087 stack to this range.
First, the identity tan(x+n Pi) = tan(x) is applied using the 8087 remainder instruction FPREM. FPREM is then used again, this time dividing by pi/4, to reduce the argument to the range required by FPTAN. After this second reduction, the 8087 status word will contain the last three bits of the quotient. This number will be used to determine the octant the final value falls in. Octant one or three requires that we call FPTAN using pi/4 minus the value obtained from FPREM. If the octant is one, five, two, or six, we invert the results of FPTAN. If the octant is two, six, three, or seven we reverse the sign of the result. These rules are derived from the identities tan(x+pi/2)=−cot(x) and tan(Pi/2−x)=cot(x). The returned values, X and Y in most 8087 literature, will be used for all of the trigonometric calculations.

Suggested Extensions: None.

Definition:

```
49152, FVARIABLE F-2
SUBROUTINE [FPTAN]
  CH 0 # MOV FTST f8087 ] FSTSTATUSW
  WAIT AH f8087 1+ ] MOV SAHF CIF
    CH DEC FCHS
  /ENDIF
  PIFLD 1 ST FXCH /BEGIN
    FPREM f8087 ] FSTSTATUSW WAIT
    AH f8087 1+ ] MOV SAHF
  JP 1 ST 0 ST FSTP SHORT__REAL F-2 ] FLD
  PIFLD FSCALE 1 ST 0 ST FSTP 1 ST FXCH
  /BEGIN
    FPREM f8087 ] FSTSTATUSW WAIT
    AH f8087 1+ ] MOV SAHF
  JP FTST f8087 ] FSTSTATUSW WAIT
  CL 0 # MOV byte f8087 1+ ] 65 # And
  byte f8087 1+ ] 64 # CMP ZIF CL DEC
  /ENDIF AH 2 # TEST NZIF
     CL 0 # CMP NZIF
       0 ST 0 ST FSTP 0 ST 0 ST FSTP
       1FLD ZFLD
     /ELSE
       1 ST 0 ST FSUB ANDPOP REVERSE
       FPTAN
     /ENDIF
   /ELSE
     1 ST 0 ST FSTP CL 0 # CMP NZIF
       0 ST 0 ST FSTP 1FLD 1FLD
     /ELSE
       FPTAN
     /ENDIF
   /ENDIF
   2AH 66 # And AH 2 # CMP ZIF
     1 ST FXCH
   /ENDIF AH 64 # CMP ZIF
     1 ST FXCH
   /ENDIF AH 64 # TEST NZIF
     FCHS
   /ENDIF CH 0 # CMP NZIF
     FCHS
   /ENDIF
END-SUB
```

TAN (F1 – F2)

Leave F2, The tangent of F1 (expressed in radians).

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The tangent of F1 radians.

Example of Use:

```
... ANGLE F@ TAN R. ...
```

This code fragment would print the tangent of the variable ANANGLE on the display.

Algorithm: Call FPTAN and return Y divided by X.

Suggested Extensions: None.

Definition:

```
CODE TAN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   [FPTAN] CALL
   1 ST 0 ST FDIV ANDPOP REVERSE
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

2FLD

A macro to push a two onto the 8087 stack.

Example of Use: See words defined below.

This macro will load a two onto the 8087 stack. It can be used just like any other 8087 instruction in our assembler code. In effect, we can use macros to extend the 8087 instruction set.

Algorithm: Push a one onto the 8087 stack and add it to itself.

Suggested Extensions: None.

Definition:

```
MACRO 2FLD
   1FLD
   0 ST 0 ST FADD
MEND
```

F/2

A macro to divide the top 8087 stack entry by two.

Example of Use: See words defined below.

This macro will load a one onto the 8087 stack and then divide the number ST(1) by it. It can be used just like any other 8087 instruction in our assembler code. In effect, we can use macros to extend the 8087 instruction set.

Algorithm: Push a one onto the 8087 stack and use FDIC to divide ST(1) by it.

Suggested Extensions: None.

Definition:

```
MACRO F/2
   2FLD
   1 ST 0 ST FDIV ANDPOP REVERSE
MEND
```

SIN (F1 – F2)

Leave F2, the sine of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The sine of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ SIN R. ...
```

This code fragment would print the sine of the variable AN__ANGLE on the display.

Algorithm: Call [FPTAN] and return 2*(Y/X) / 1+(Y/X)**2.

Suggested Extensions: None.

Definition:

```
CODE SIN
   DX BP MOV BP SP MOV
   SHORT__REAL 0 [BP+#] FLD
   FTST f8087 ] FSTSTATUSW WAIT
   AH f8087 1+ ] MOV SAHF ZIF
      0 ST 0 ST FSTP ZFLD
   /ELSE
      F/2 [FPTAN] CALL
      1 ST 0 ST FDIV ANDPOP REVERSE
      0 ST 0 ST FLD
      2FLD 1 ST 0 ST FMUL ANDPOP
      1 ST FXCH 0 ST 0 ST FLD
      1 ST 0 ST FMUL ANDPOP REVERSE
      1FLD 1 ST 0 ST FADD ANDPOP
      1 ST 0 ST FDIV ANDPOP REVERSE
   /ENDIF
   SHORT__REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

COS (F1 – F2)

Leave F2, the cosine of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cosine of F1 (radians).

Example of Use:

```
... AN__ANGLE F@ COS R. ...
```

This code fragment would print the cosine of the variable AN__ANGLE on the display.

Algorithm: Call [FPTAN] and return 1-(Y/X)**2 / 1+(Y/X)**2.

Suggested Extensions: None.

Definition:

```
CODE COS
  DX BP MOV BP SP MOV
  SHORT__REAL 0 [BP+#] FLD
  FTST 18087 ] FSTSTATUSW WAIT
  AH 18087 1+ ] MOV SAHF ZIF
    0 ST 0 ST FSTP 1FLD
  /ELSE
    F/2 [FPTAN] CALL
    1 ST 0 ST FDIV ANDPOP REVERSE
    0 ST 0 ST FMUL 0 ST 0 ST FLD
    1FLD 1 ST 0 ST FSUB ANDPOP
    1 ST FXCH
    1FLD 1 ST 0 ST FADD ANDPOP
    1 ST 0 ST FDIV ANDPOP REVERSE
  /ENDIF
  SHORT__REAL 0 [BP+#] FSTP
  BP DX MOV WAIT
NEXT
```

1/X (F1 – F2)

Leave F2, the multiplicative identity of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The multiplicative identity of F1.

Example of Use:

```
... AN__ANGLE F@ 1/X R. ...
```

This code fragment would print the multiplicative identity of the variable AN__ANGLE on the display.

Algorithm: Move F1 onto the 8087 stack and them push a 1 on the 8087 stack. Divide the two and return the result to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE 1/X
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   1FLD 1 ST 0 ST FDIV ANDPOP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

COT (F1 – F2)

Leave F2, the cotangent of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cotangent of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ COT R. ...
```

This code fragment would print the cotangent of the variable AN_ANGLE on the display.

Algorithm: Call tangent and inverse the result. (COT(X) = 1/TAN(X).

Suggested Extensions: None.

Definition:

```
: COT TAN 1/X ;
```

CSC (F1 – F2)

Leave F2, the cosecant of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cosecant of F1 (radians).

Example of Use:

```
... ANANGLE F@ CSC R. ...
```

This code fragment would print the cosecant of the variable AN__ANGLE on the display.

Algorithm: Call sine and inverse the result. (CSC(X) = 1/SIN(X).

Suggested Extensions: None.

Definition:

```
: CSC SIN 1/X ;
```

SEC (F1 – F2)

Leave F2, the secant of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The secant of F1 (radians).

Example of Use:

```
... AN__ANGLE F@ SEC R. ...
```

This code fragment would print the secant of the variable AN__ANGLE on the display.

Algorithm: Call cosine and inverse the result. (SEC(X) = 1/COS(X)

Suggested Extensions: None.

Definition:

```
: SEC COS 1/X ;
```

Assembler subroutine that will execute the 8087 FPATAN instruction on any number.

Algorithm: The 8087 instruction FPATAN is used in all the inverse trigonometric calculations. FAPTAN requires two values, known as X and Y, on the 8087 stack. 0 < Y < X must hold true. The identities ArcTan(Z) = −ArcTan(−Z) and ArcTAn(Z) = Pi/2 − ArcTan(1/Z) are used to bring the two arguments into the proper range.

Definition:

```
SUBROUTINE [FPATAN]
   1 ST 0 ST FLD FTST
   CH 0 # MOV f8087 ] FSTSTATUSW
   0 ST 0 ST FSTP
   WAIT AH f8087 1+ ] MOV SAHF CIF
      CH DEC 1 ST FXCH FABS 1 ST FXCH
   /ENDIF
   1 ST FCOM f8087 ] FSTSTATUSW CL 0 #
   MOV WAIT AH f8087 1+ ] MOV SAHF CIF
      CL DEC 1 ST FXCH
   /ENDIF FPATAN CL 0 # CMP NZIF
      FCHS
      1FLD FCHS PIFLD FSCALE
      1 ST FSTP
      1 ST 0 ST FADD ANDPOP
   /ENDIF CH 0 # CMP NZIF
      FCHS
   /ENDIF
END-SUB
```

ATAN (F1 − F2)

Leave F2, the angle (in radians) whose tangent is F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arctangent of F1.

Example of Use:

```
... SOME_RADIANS F@ ATAN R. ...
```

This code fragment would print the angle whose tangent was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as Y and set X to one, then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ATAN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD 1FLD
   [FPATAN] CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ACOTN (F1 – F2)

Leave F2, the angle (in radians) whose cotangent is F1.

Argument Range: −Infinity <= F1 <= −1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccotangent of F1.

Example of Use:

```
... SOME_RADIANS F@ ACOTN R. ...
```

This code fragment would print the angle whose cotangent was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as X and set Y to one, then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ACOTN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD FTST
```

```
    18087 ] FSTSTATUSW WAIT
    AH 18087 1+ ] MOV SAHF CIF
      FCHS 1FLD FCHS
    /ELSE
      1FLD
    /ENDIF 1 ST FXCH FPATAN CALL
    SHORT_REAL 0 [BP+#] FSTP
    BP DX MOV WAIT
NEXT
```

ASIN (F1 – F2)

Leave F2, the angle (in radians) whose sine is F1.

Argument Range: −1 <= F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arcsine of F1.

Example of Use:

```
... SOME_RADIANS F@ ASIN R. ...
```

This code fragment would print the angle whose sine was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as Y and set X to SQR((1−F1)*(1+F1)), then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ASIN
  DX BP MOV BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  0 ST FLD 0 ST FLD 1FLD
  1 ST 0 ST FADD ANDPOP
  1 ST FXCH 1FLD
  1 ST 0 ST FSUB ANDPOP
  1 ST 0 ST FMUL ANDPOP
  FABS FSQRT
```

```
   [FPATAN] CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ACOS (F1 – F2)

Leave F2, the angle (in radians) whose cosine is F1.

Argument Range: −1 <= F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccosine of F1.

Example of Use:

```
... SOME_RADIANS F@ ACOS R. ...
```

This code fragment would print the angle whose cosine was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to SQR(1−F1), set X equal to SQR(1+F1), then call [FPATAN]. Multiply the result [FPATAN] returned by 2.

Suggested Extensions: None.

Definition:

```
CODE ACOS
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   0 ST FLD 1FLD
   1 ST 0 ST FSUB ANDPOP FSQRT
   1 ST FXCH 1FLD
   1 ST 0 ST FADD ANDPOP FSQRT
   [FPATAN] CALL
   2FLD 1 ST 0 ST FMUL ANDPOP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ASEC (F1 – F2)

Leave F2, the angle (in radians) whose secant is F1.

Argument Range: −Infinity <= F1 <= −1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arcsecant of F1.

Example of Use:

```
... SOME_RADIANS F@ ASEC R. ...
```

This code fragment would print the angle whose secant was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to SQR(F1 − 1), set X equal to SQR(F1 + 1), then call [FPATAN]. Multiply the result [FPATAN] returned by 2.

Suggested Extensions: None.

Definition:

```
CODE ASEC
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   0 ST FLD 1FLD
   1 ST 0 ST FSUB ANDPOP REVERSE
   FABS FSQRT 1 ST FXCH 1FLD
   1 ST 0 ST FADD ANDPOP
   FABS FSQRT [FPATAN] CALL
   2FLD 1 ST 0 ST FMUL ANDPOP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ACSC (F1 – F2)

Leave F2, the angle (in radians) whose cosecant is F1.

Argument Range: −Infinity <= F1 <= −1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccosecant of F1.

Example of Use:

```
... SOME_RADIANS F@ ACSC R. ...
```

This code fragment would print the angle whose cosecant was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to the sign of F1 (that is, 1 or −1), set X equal to SQR((F1+1)*(F1−1)), then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ACSC
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FTST 18087 ] FSTSTATUSW
   1FLD
   WAIT AH 18087 1+ ] MOV SAHF CIF
      FCHS
   /ENDIF 1 ST FXCH
   0 ST FLD 1FLD
   1 ST 0 ST FADD ANDPOP
   1 ST FXCH 1FLD
   1 ST 0 ST FSUB ANDPOP REVERSE
   1 ST 0 ST FMUL ANDPOP
   FABS FSQRT
   [FPATAN] CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

FSIGN (F1 – F2)

Leave F2, −1 if F1 is negative, 0 if F1 is zero, 1 if F1 is positive.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The sign of F1 as above.

Example of Use:

```
... SOME__RADIANS F@ ACSC R. ...
```

This code fragment would print the angle whose cosecant was the value held in the variable SOME__RADIANS.

Algorithm: Use the 8087 instruction FTST to test F1, then move the proper value to the 8087 stack.

Suggested Extensions: None.

Definition:

```
CODE FSIGN
   DX BP MOV BP SP MOV
   SHORT__REAL 0 [BP+#] FLD
   FTST f8087 ] FSTSTATUSW
   1FLD
   WAIT AH f8087 1+ ] MOV SAHF CIF
      FCHS
   /ENDIF
   ZIF
      0 ST 0 ST FSTP ZFLD
   /ENDIF 1 ST FXCH 0 ST 0 ST FSTP
   SHORT__REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

SINH (F1 – F2)

Leave F2, the hyperbolic sine of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic sine of F1.

Example of Use:

```
... CABLE F@ SINH R. ...
```

This code fragment would print the hyperbolic sine of the variable CABLE on the display.

Algorithm: Calculate the identity SINH(X) = (e∧X − e∧X)/2

Suggested Extensions: None.

Definition:

```
16384, FCONSTANT 2.0

: SINH C( F1 – F2)
   2DUP EXP 2SWAP FNEGATE EXP F–
   2.0 F/ ;
```

COSH (F1 – F2)

Leave F2, the hyperbolic cosine of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cosine of F1.

Example of Use:

```
... CABLE F@ COSH R. ...
```

This code fragment would print the hyperbolic cosine of the variable CABLE on the display.

Algorithm: Calculate the identity COSH(X) = (e∧X + e∧−X)/2

Suggested Extensions: None.

Definition:

```
: COSH C( F1 – F2)
   2DUP EXP 2SWAP FNEGATE EXP F+
   2.0 F/ ;
```

TANH (F1 – F2)

Leave F2, the hyperbolic tangent of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic tangent of F1.

Example of Use:

```
... CABLE F@ TANH R. ...
```

This code fragment would print the hyperbolic tangent of the variable CABLE on the display.

Algorithm: Calculate the identity TANH(X) = (−e^−X/(e^X + e−X)) * 2 + 1

Suggested Extensions: None.

Definition:

```
16256, FCONSTANT 1.0

: TANH 2.0 F* EXP 1.0 F+ 2.0 2SWAP
   F/ 1.0 2SWAP F− ;
```

SECH (F1 – F2)

Leave F2, the hyperbolic secant of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic secant of F1.

Example of Use:

```
... CABLE F@ SECH R. ...
```

This code fragment would print the hyperbolic secant of the variable CABLE on the display.

Algorithm: Inverse the result of COSH, SECH(X) = 1 / COSH(X).

Suggested Extensions: None.

Definition:

```
: SECH COSH 1/X ;
```

CSCH (F1 – F2)

Leave F2, the hyperbolic cosecant of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cosecant of F1.

Example of Use:

```
... CABLE F@ CSCH R. ...
```

This code fragment would print the hyperbolic cosecant of the variable CABLE on the display.

Algorithm: Inverse the result of SINH, CSCH(X) = 1 / SINH(X).

Suggested Extensions: None.

Definition:

```
: CSCH SINH 1/X ;
```

COTNH (F1 – F2)

Leave F2, the hyperbolic cotangent of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cotangent of F1.

Example of Use:

```
... CABLE F@ COTNH R. ...
```

This code fragment would print the hyperbolic cotangent of the variable CABLE on the display.

Algorithm: Inverse the result of TANH, COTNH(X) = 1 / TANH(X).

Suggested Extensions: None.

Definition:

```
: COTNH TANH 1/X ;
```

ASINH (F1 – F2)

Leave F2, the value whose hyperbolic sine is F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic sine of F1.

Example of Use:

```
... H-VALUE F@ ASINH R. ...
```

This code fragment would print the inverse hyperbolic sine of the variable H–VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X^2 + 1)).

Suggested Extensions: None.

Definition:

```
: ASINH
  2DUP 2DUP F* 1.0 F+ SQRT F+ LN ;
```

ACOSH (F1 – F2)

Leave F2, the value whose hyperbolic cosine is F1.

Argument Range: 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cosine of F1.

Example of Use:

```
... H-VALUE F@ ACOSH R. ...
```

This code fragment would print the inverse hyperbolic cosine of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X^2 - 1)).

Suggested Extensions: None.

Definition:

```
: ACOSH
  2DUP 2DUP F* 1.0 F- SQRT F+ LN ;
```

ATANH (F1 - F2)

Leave F2, the value whose hyperbolic tangent is F1.

Argument Range: -Infinity < F1 < -1 or 1 < F1 < Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic tangent of F1.

Example of Use:

```
... H-VALUE F@ ATANH R. ...
```

This code fragment would print the inverse hyperbolic cosine of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X^2 - 1)).

Suggested Extensions: None.

Definition:

```
: ATANH
   2DUP 1.0 F+ 2SWAP 1.0 2SWAP F- F/ LN
   2.0 F/ ;
```

ASECH (F1 – F2)

Leave F2, the value whose hyperbolic secant is F1.

Argument Range: 0 < F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic secant of F1.

Example of Use:

```
... H-VALUE F@ ASECH R. ...
```

This code fragment would print the inverse hyperbolic secant of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASECH(X) = LN((1 + SQR(1 – X^2))/ X).

Suggested Extensions: None.

Definition:

```
: ASECH
   2DUP 2DUP FNEGATE F* 1.0 F+ SQRT
   1.0 F+ LN 2.0 F/ ;
```

ACSCH (F1 – F2)

Leave F2, the value whose hyperbolic cosecant is F1.

Argument Range: –Infinity < F1 < 0 or 0 < F1 < +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cosecant of F1.

Example of Use:

```
... H-VALUE F@ ASECH R. ...
```

This code fragment would print the inverse hyperbolic cosecant of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ACSCH(X) = LN(1/X + (SQR(1+ X^2)/ABS(X))).

Suggested Extensions: None.

Definition:

```
: ACSCH
   2DUP 2DUP 2DUP F* 1.0 F+ SQRT 1.0 F+
   2SWAP FSIGN F* LN 2SWAP F/ ;
```

ACOTNH (F1 – F2)

Leave F2, the value whose hyperbolic cotangent is F1.

Argument Range: −Infinity < F1 < −1 or 1 < F1 < Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cotangent of F1.

Example of Use:

```
... H-VALUE F@ ACOTNH R. ...
```

This code fragment would print the inverse hyperbolic cotangent of the variable H-VALUE on the display.

Algorithm: Calculate the identity, COTNH(X)=LN((X+1)/(X−1))/2

Suggested Extensions: None.

Definition:

```
: ACOTNH
   2DUP 2DUP 1.0 F+ 2SWAP 1.0 F- F/ LN
   2.0 F/ ;
```

7

Strings

Words Defined in This Chapter:

$VARIABLE	Define a string variable.
$CONSTANT	Define a string constant.
$ARRAY	Define a one-dimensional string array.
$@	Fetch a string.
$!	Store a string.
$.	Print a string.
$?	Fetch and print a string.
LEN	Determine the length of a string.
LEFT$	Return the left-hand portion of a string.
RIGHT$	Return the right-hand portion of a string.
MID$	Return a section of a string.
ASC	Return the ASCII value of a string.
CHR$	Form a string from a specified ASCII character.
$+!	Concatenate two strings.
+ $=	Compare two strings for equality.
$<	String compare, less than.
$>	String compare, greater than.
$<=	String compare, less than or equal.
$>=	String compare, greater than or equal.
$<>	String compare, not equal.

FIND$	Search for a substring in a string.
NUM$	Convert a single-length number to string.
DNUM$	Convert a double-length number to a string.
DVAL	Convert a string to a double-length number.
VAL	Convert a string to a single-length number.
SORT	Sort a string array.

This chapter contains a complete string-handling package for Forth. The ability to manipulate and process text is, to some degree, called upon in almost every computer application. The need may range from the simple prompts of an engineering program to the complex string manipulation of a text editor. Despite the fact that basic Forth does not come with a predefined string package, Forth easily adapts itself to the manipulation of strings. We'll borrow some terminology from BASIC and use the $ as shorthand for the word string.

Each string will have a length byte, and each string variable will use a byte to hold the maximum size string the variable can hold. Since bytes are being used the maximum size of any single string will be 255 characters.

The first three words, $VARIABLE, $CONSTANT, and $ARRAY will enable us to allocate space for strings. These three words are defining words that have both a compile-time and run-time behavior. The words $! (String Store), $@ (String Fetch), and $+! (String Plus Store) will allow us to store values into string variables and fetch them for further use.
They are analogous to the words @, !, and +!.

LEFT$, RIGHT$, and MID$ enable us to break up strings into smaller strings. $. (String Dot) allows us to print out strings. The words $= (String Equal), $< (String Less Than), $> (String Greater Than), $<= (String Less Than or Equal), $>= (String Greater Than or Equal), and $<> (String Not Equal) can be used to compare string lexically. The ASCII code used on the IBM-PC determines the lexical ordering.

VAL, DVAL, NUM$, and DNUM$ are used to convert between numeric and string format. ASC returns the ASCII value of the first character of a string, and CHR$ will make a one-character string consisting of the ASCII character passed to it. FIND$ can be used for substring searches.

The final word, SORT, uses a quicksort algorithm to sort a string array. This is an example of a generic word; we provide SORT with compare and exchange routines (using vectors) and it handles the sort from then on.

Possible Enhancements

The strings in this package are limited to 255 bytes in length. If you

encounter an application that requires larger strings, this package could easily be modified to use larger strings. Because they use the core Forth string words that return byte length strings, NUM$ and VAL$ would be the most difficult to redefine.

A more subtle limitation of this string package is the way in which the string-handling words affect the string variables directly. This is unlike number manipulation in Forth in which numbers on the stack are manipulated. One possible solution would be to have a string stack. A disadvantage would be the large amount of memory required by a string stack; however, if your Forth has the ability to access memory outside the normal 64K Forth limit, a string stack could be an extremely useful enhancement.

$VARIABLE (N –) (– A)

Define a string and allocate space for it.

Stack on Entry: (Compile Time) (N) – Maximum Size of String.
(Run Time) Empty

Stack on Exit: (Compile Time) Empty
(Run Time) (A) – Address of the string variable.

Example of Use:

```
64 $VARIABLE DISK-NAME
```

This will allot a string that can hold up to 64 characters, with the name DISK-NAME.

Algorithm: Enclose the maximim length, then an initial length of zero in the dictionary. Allot space for the string.

Suggested Extensions: Allow the created variable to have an initial value.

Definition:

```
: $VARIABLE <BUILDS DUP C,
   0 C, ALLOT DOES> ;
```

$CONSTANT (–) (– A)

Define and set a string constant.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of string constant.

Example of Use:

```
$CONSTANT MY__NAME JAMES"
```

This defines a string constant, MY__NAME, with a value "JAMES". Note that only the trailing quotation mark is used in the definition.

Algorithm: The word WORD returns a string in memory; simply enclose what it returns in the dictionary.

Suggested Extensions: Allow the delimiter to be any character, not just the quote used now.

Definition:

```
34 CCONSTANT ASCII"

: $CONSTANT <BUILDS ASCII" WORD
   C@ 1+ ALLOT DOES> ;
```

$ARRAY (N1 N2 –) (N – A)

Create and allocate space for a string array.

Stack on Entry: (Compile Time) (N1) – Number of strings the array holds.
(N2) – The length of each string.
(Run Time) (N) – Index number of string.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of string variable.

Example of Use:

```
6 40 $ARRAY PHYLUM
```

This would allocate a 16-string array with the name PHYLUM. Each string in the array could hold a maximum of 40 characters. Individual strings in the array could be accessed like this:

This would leave the address of the fourth string in the array. The first string would be referenced as zero.

Algorithm: (Compile Time) For each string, store the maximum size and an initial length of zero, then allocate space for the string.

(Run Time): Calculate the position of the requested string by multiplying the index on the stack by the space for each string and adding the starting address of the array. The space each string occupies is the maximum length plus two (the maximum length byte and the length byte).

Suggested Extensions: Allow the definition of multidimensional arrays.

Definition:

```
: $ARRAY <BUILDS
   SWAP 0 DO
      DUP C, 0 C, DUP ALLOT
   LOOP DROP DOES>
   DUP C@ 2+ ROT * + ;
```

$@ (A1 – A2)

Fetch the address of a string from a string variable.

Stack on Entry: (A1) – Address of string variable.

Stack on Exit: (A2) – Address of the string the variable holds.

Example of Use:

```
DISK-NAME $@
```

This would leave the address of the string held by DISK-NAME on the stack.

Algorithm: Simply add one to the address on the stack to skip the maximum length byte.

Suggested Extensions: None.

Definition:

```
: $@ 1+ ;
```

$! (A1 A2 –)

Store a string in a string variable.

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string variable.

Stack on Exit: Empty.

Example of Use:

```
DISK-NAME $@ 2 PHYLUM $!
```

This would store the string held in DISK-NAME into the third string of the array PHYLUM.

Algorithm: First, check to make sure there is sufficient room in the string variable to hold the string; if there is not, abort with an error message. If sufficient room is available, use CMOVE to move the string and length (since they are contiguous) into the string variable.

Suggested Extensions: None.

Definition:

```
: $LEN-CHECK
   <
   ABORT" String Past Storage Allocated."
   ;

: $! 2DUP C@ SWAP C@ $LEN-CHECK
   1+ OVER C@ 1+ CMOVE ;
```

$. (A –)

Print a string on the display.

Stack on Entry: (A) – Address of a string.

Stack on Exit: Empty.

Example of Use:

```
MY-NAME $.
```

Assuming MY__NAME was defined as under $CONSTANT, this code would print "JAMES" on the display.

Algorithm: If the length of the string is nonzero, convert the string address to an address and count suitable for TYPE.

Suggested Extensions: None.

Definition:

```
: $. DUP C@ ?DUP IF
     SWAP 1+ SWAP TYPE
  ENDIF ;
```

$? (A –)

Fetch and print a string.

Stack on Entry: (A) – Address of a String Variable.

Stack on Exit: Empty.

Example of Use:

```
DISK-NAME $?
```

This code would print the string held in DISK-NAME on the display.

Algorithm: Fetch the string, the print it.

Suggested Extensions: None.

Definition:

```
: S? S@ $. ;
```

LEN (A – N)

Return the length of a string.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (N) – The length of that string.

Example of Use:

```
MY-NAME LEN .
```

Again, assuming MY-NAME is defined as under $CONSTANT, this code would print a 5 on the display.

Algorithm: Since the current length is kept as part of the string, simply fetch the length.

Suggested Extension: None.

Definition:

```
: LEN C@ ;
```

LEFT$ (A N -)

Take the leftmost characters of a string.

Stack on Entry: (A) – Address of a string.
(N) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY__NAME DISK-NAME $! DISK-NAME 3 LEFT$ DISK-NAME $?
```

This would print "JAM" on the display. If the number of characters the left string is passed is greater than the length of the string, the string will not be changed.

Algorithm: First, check the current length of the string against the desired new length, choose the smaller of the two. Store this value in the length byte of the string.

Suggested extensions: None.

Definition:

```
: LEFT$ SWAP DUP C@ ROT MIN SWAP C! ;
```

RIGHT$ (A N -)

Take the rightmost characters of a string.

Stack on Entry: (A) – Address of a string.
(N) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! DISK-NAME 2 RIGHT$ DISK-NAME $?
```

This would print "ES" on the display. If the number of characters that right string is passed is greater than the length of the string, the string will not be changed.

Algorithm: If the length of the string is less than the number of desired characters, exit the word. Otherwise, determine the start address that must be moved from by calculating START ADDRESS OF STRING + LENGTH OF STRING – NUMBER OF CHARACTERS DESIRED. Move from this address to the start of the string. Finally, store the new length of the string in the length byte.

Suggested Extensions: None.

Definition:

```
: 3PICK 3 PICK ;

: RIGHT$ 2DUP SWAP C@ >= IF
     2DROP EXIT
  ENDIF
  2DUP
  SWAP DUP DUP C@ + 1+ 3PICK -
  SWAP 1+ ROT CMOVE
  SWAP C! ;
```

MID$ (A N1 N2 –)

Return the middle portion of a string.

Stack on Entry: (A) – Address of the string.
(N1) – Starting position in the string.
(N2) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! DISK-NAME 2 2 RIGHT$ DISK-NAME
$?
```

This would print "AM" on the display. If the starting position that MID$ is passed is greater than the length of the string, the string will become an empty string. If there are insufficient characters after the start position, only those available would be returned.

Algorithm: First, determine if the string contains the start position; if not convert it to the empty string and exit the word. Next, determine the number of characters left in the string after the start position, and use this as the length if it is smaller than the number of characters requested. Move the characters from the middle of the string to the start, and, finally, store the new length.

Suggested Extensions: None.

Definition:

```
: MID$ >R 2DUP SWAP C@ > R> SWAP IF
     2DROP DROP EXIT
  ENDIF
  3PICK C@ 3PICK - MIN
  DUP 4 PICK C!
  SWAP 3PICK + 1+
  LROT SWAP 1+ SWAP CMOVE ;
```

ASC (A - C)

Return the ASCII value of the first character of a string.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (C) – ASCII value of the first character.

Example of Use:

```
MY_NAME ASC .
```

This code would print a 99 on the display, since the ASCII value for the character "J" is 99. If ASC is passed the empty string, it will return a zero.

Algorithm: If the string has a length of zero, return a zero. Otherwise, fetch the value of the first character of the string.

Suggested Extensions: Implement a flag for the string package that would enable a user of the package to specify how to handle the ASC of an empty string, either returning a zero (as the code does now), or aborting with an error message.

Definition:

```
: ASC DUP C@ IF
     1+ C@
  ELSE
     DROP 0
  ENDIF ;
```

CHR$ (C – A)

Return a string that consists of a specific ASCII character.

Stack on Entry: (C) – ASCII value.

Stack on Exit: (A) – Address of a string that consists of the ASCII value.

Example of Use:

```
77 CHR$ $.
```

This code would print an asterisk on the display.

Algorithm: /CHR$/ is a string variable that will be used to hold the string CHR$ will create. When CHR$ is invoked, store the ASCII value on the stack into /CHR$/ itself, then fetch its address.

Suggested Extensions: None.

Definition:

```
1 $VARIABLE /CHR$/
1 /CHR$/ 1+ C!
: CHR$ /CHR$/ 2+ C! /CHR$/ $@ ;
```

$+! (A1 A2 –)

Concatenate a string into a string variable.

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string variable.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! 102 CHR$ DISK-NAME $+! DISK-
NAME $?
```

This code would print "JAMES!" on the display.

Algorithm: First, determine if there is room for the new string that will be created; if not, exit with an error message. If there is room, move the string onto the end of the string variable. Add the length of the string to the current length of the string variable.

Suggested Extensions: Define a word to concatenate onto the left side of a string.

Definition:

```
: $+! 2DUP DUP C@ SWAP 1+ C@ ROT
   C@ + $LEN-CHECK
   2DUP
   1+ DUP C@ + 1+
   SWAP DUP C@ >R 1+ SWAP R> CMOVE
   SWAP C@ SWAP 1+ C+! ;
```

$= (A1 A2 – F)

Compare two strings. (Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Example of Use:

```
MY_NAME DISK-NAME $@ $= .
```

This code would print a Boolean flag on the display, 0 (or false) if the two strings were not equal, –1 (or true) if they were.

Algorithm: First, determine if the lengths of the two strings are the same; if not, exit the word with a false flag on the stack. If they are of the same length, use –TEXT, which will compare the strings.

Suggested Extensions: None.

Definition:

```
: $= 2DUP C@ SWAP C@ = IF
     1+ SWAP DUP C@ SWAP 1+ -TEXT NOT
   ELSE
     2DROP 0
   ENDIF ;
```

$< (A1 A2 – F)

Compare two strings. (Greater Than).

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Example of Use:

```
MY__NAME DISK-NAME $@ $< .
```

This code would print a Boolean flag on the display, –1 (or true) if the string DISK–NAME holds is greater lexically than MY__NAME, 0 (or false) if it is less than or equal to MY__NAME.

Algorithm: Compare the strings using the length of the smaller string. If a less than condition is found, exit the word with a true flag. If a greater than condition is found, exit with a true flag. If the two strings were found to be equal, compare the lengths and return that flag.

Suggested Extensions: None.

Definition:

```
: $< 2DUP 2DUP C@ SWAP C@ MIN >R
   1+ SWAP 1+ R> SWAP -TEXT DUP 0< IF
     2DROP DROP 0
   ELSE
     0> IF
       2DROP -1
     ELSE
       C@ SWAP C@ >
     ENDIF
   ENDIF ;
```

$> (A1 A2 – F)

Compare two strings (Less Than)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $> SWAP $< ;
```

$<=

Compare two strings (Greater Than or Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $>= $< NOT ;
```

$>= (A1 A2 – F)

Compare two strings (Less Than or Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $<= $> NOT ;
```

$<>

Compare two strings (Not Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $<> $= NOT ;
```

FIND$ (A1 A2 – N)

Determine if one string is a substring of another.

Stack on Entry: (A1) – Address of a string. (String to look for.)
(A2) – Address of a string. (String to look in.)

Stack on Exit: (N) – Position substring found at. (Zero, if not found.)

Example of Use:

```
$CONSTANT ME ME"
ME MY__NAME FIND$ .
```

This code would print 3 on the display, the position of "ME" in "JAMES".

Algorithm: Loop through the string being searched, calling –TEXT once for each possible starting position. The length –TEXT uses will be the length of the string being searched for. If –TEXT returns a match, exit the loop and return the loop index.

Suggested Extensions: None.

Definition:

```
: FIND$ 0 LROT DUP C@ 0 DO
     DUP C@ I - 3PICK C@ < IF
       LEAVE
   ELSE
     OVER DUP C@ SWAP 1+ SWAP
     ?DUP IF
       3PICK I 1+ + -TEXT NOT
     ELSE
       DROP 0
     ENDIF
     IF
```

```
      2DROP DROP I 1+ 0, LEAVE
    ENDIF
  ENDIF
LOOP 2DROP ;
```

NUM$ (N – A)

Convert a number to a string.

Stack on Entry: (N) – Integer

Stack on Exit: (A) – Address of a string.

Example of Use:

```
456 NUM$ 1 LEFT$ $.
```

This code would print a 4, the leftmost character of the string "456".

Algorithm: Use the built–in Forth conversions words.

Suggested Extensions: Define a word that produces a string corresponding to the currency form of a number, such as "$456.00" for the above 456.

Definition:

```
: INUM$ <# #S SIGN #> OVER 1– C! 1– ;

: NUM$ DUP ABS 0 INUM$ ;
```

DNUM$ (D – A)

Convert a double-length number to a string.

Stack on Entry: (D) – A double–length number.

Stack on Exit: (A) – Address of a string.

Definition:

```
: DNUM$ SWAP OVER DABS INUM$ ;
```

DVAL (A – D)

Convert a string to a double-length number.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (D) – Double-length number.

Example of Use:

```
452 NUM$ DVAL D.
```

This code would print the number "452" on the display.

Algorithm: Check the string for a leading negative sign, then use the Forth word >BINARY to do the conversion. If a negative sign was found, negate the result that >BINARY leaves on the stack.

Suggested Extensions: None.

Definition:

```
$CONSTANT "-" -"

: DVAL DUP >R "-" SWAP FIND$ 1 = IF
     R> 1+ -1 >R
  ELSE
     R> 0 >R
  ENDIF
  0, ROT >BINARY DROP R> IF
     DNEGATE
  ENDIF ;
```

VAL (A – N)

Convert a string to a single-length number.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (N) – A single-length number.

Definition:

```
: VAL DVAL DROP ;
```

SORT (N1 N2 –)

Sort a string array.

Stack on Entry: (N1) – Index of first array position.
(N2) – Index of second array position.

Stack on Exit: Empty.

Example of Use:

```
0 9 SORT
```

This code would sort the array NAMES.

Algorithm: The quicksort Algorithm can be found in Wirth[76]. We must provide the SORT word with compare and exchange operations for strings. The words .EXCHANGE, .<COMPARE, .>COMPARE do this.

Suggested Extensions: Implement another vector to allow SORT to sort any string array. Moving each string in the exchange word is a time-consuming process, SORT could be extended to use an index array.

Definition:

```
0 VARIABLE [<COMPARE]
0 VARIABLE [>COMPARE]
0 VARIABLE [EXCHANGE]
0 VARIABLE [X!]

: <X-COMPARE [<COMPARE] @ EXECUTE ;
: >X-COMPARE [>COMPARE] @ EXECUTE ;
: EXCHANGE [EXCHANGE] @ EXECUTE ;
: X! [X!] @ EXECUTE ;

: LEFTSWEEP
   SWAP BEGIN
      DUP <X-COMPARE
   WHILE
      1+
   REPEAT SWAP ;
: RIGHTSWEEP
   BEGIN
      DUP >X-COMPARE
   WHILE
      1-
   REPEAT ;

: SORT
   2DUP 2DUP + 2/ X! BEGIN
```

```
    LEFTSWEEP RIGHTSWEEP
    2DUP <= IF
      2DUP EXCHANGE
      1- SWAP 1+ SWAP
    ENDIF
    2DUP >
  UNTIL
  >R ROT R>
  2DUP < IF
    SORT
  ELSE
    2DROP
  ENDIF
  SWAP 2DUP < IF
    SORT
  ELSE
    2DROP
  ENDIF ;


64 10 $ARRAY NAMES
64 $VARIABLE TEMP
64 $VARIABLE EXCH

: .X! NAMES $@ TEMP $! ;

: .<COMPARE
NAMES $@ TEMP $@ $< ;

: .>COMPARE
   NAMES $@ TEMP $@ > ;

: .EXCHANGE
   DUP NAMES $@ EXCH $!
   OVER NAMES $@ SWAP NAMES $!
   EXCH $@ SWAP NAMES $! ;

' .<COMPARE [<COMPARE] !
' .>COMPARE [>COMPARE] !
' .EXCHANGE [EXCHANGE] !
' .X! X! !
```

Input Formatting

Words Defined in This Chapter:

MININ	A byte variable used to specify the minimum number of characters allowed by GET-INPUT.
MAXIN	A byte variable used to specify the maximum number of characters allowed by GET-INPUT.
LEGAL-CHARS	A string variable used to specify the characters GET-INPUT will allow to be entered.
INPUT	A string variable that holds the string gotten by GET-INPUT.
OK-TO-BEEP	A Boolean variable, true if GET-INPUT should beep on an error condition, false otherwise.
KILL-CHAR	A byte constant for the keyboard character that will erase a line in GET-INPUT.
RETURN-CHAR	A byte constant for the keyboard character that will cause GET-INPUT to exit.
BACKSPACE	A byte constant for the keyword character that will cause GET-INPUT to back up one space.
ASCII-CHAR?	Determine if a character is a printable ASCII character.
LEGAL?	Determine if a character will be valid for GET-INPUT.
NULL$SET	Set a string variable to the empty string.
CHOP1	Remove the rightmost character from a string.

DESTRUCT	Erase the character at the current cursor position and place the cursor one to the left of the current cursor position.
RETURN-OK?	Leave a true flag if a return is legal in GET-INPUT.
BACKSPACE-OK?	Leave a true flag if a backspace is legal in GET-INPUT.
ANY?	Leave a true flag if any character is legal in GET-INPUT.
/KILL/	/Erase the current input field and move the cursor to the beginning of the input field.
/OK/	Store a valid character in the string INPUT.
/BACKSPACE/	Handle a backspace for GET-INPUT.
/INIT/	Initialize values for GET-INPUT.
?BEEP	Beep if the variable OK-TO-BEEP holds a true flag.
GET-INPUT	Get input from the keyboard.
INT-INP	Input an integer.
DINT-INP	Input a double-length integer.
FP-INP	Input a floating-point number.
MINVAL	A variable used to specify the minimum allowed integer for INT-BOUNDED-INP.
MAXVAL	A variable used to specify the maximum allowed integer for INT-BOUNDED-INP.
INT-BOUNDED-INP	Input an integer that falls within a specified range.
MONTH	Prompt the user for, and input, a month.
DAY	Prompt the user for, and input, a day.
YEAR	Prompt the user for, and input, a year.
MDY-INPUT	Input a date.
AM/PM	Ask for the strings "AM" or "PM".
HOUR	Prompt the user for, and input, an hour.
MINUTE	Prompt the user for, and input, a minute.
TIME-INP	Input a time in military format.
(Y/N)	Ask for a "Y" or "N" key.
SINGLE-KEY	Ask for a specific set of single keys.
$->UPPER	Convert a string to uppercase.

This chapter presents a set of words to deal with input from the keyboard. There is a infamous acronym in computer science, GIGO, that stands for "Garbage In, Garbage Out." The words in this chapter will stop garbage from ever getting into your programs.

The words in this chapter will function by restricting the user. When we want a number to be input, we will only allow the number keys to be active. If we want a month to be input, we'll only allow the integers between 1 and 12. The word GET-INPUT will be our workhorse. It will be able to place a minimum and maximum on the number of characters to be entered. It will be able to restrict the keys active on the keyboard to any set we specify. By using it as a base we will be able to make sure we get exactly what we want from the user.

The words in this chapter make use of the string words presented in the previous chapter.

Suggested Extensions: This chapter's set of words is quite complete. One possible extension would be to implement a keyboard macro facility into the input routine. This could be accomplished by having GET-INPUT look up characters in an array of strings, and replacing the character with the string.

MININ (- A)

A byte variable used to specify the minimum number of characters allowed by GET-INPUT. Valid range: 0 to 255.

Stack on Entry: Empty.

Stack on Exit: (A) - The address of MININ.

Example of Use:

```
2 MININ C!
```

Storing a two in MININ will cause GET-INPUT to require that at least two characters are entered.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE MININ
```

MAXIN (- A)

A byte variable used to specify the maximum number of characters allowed by GET-INPUT. Valid range: 1 to 255.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MAXIN.

Example of Use:

```
12 MAXIN C!
```

Storing a twelve in MAXIN will cause GET–INPUT to limit the number of characters entered to twelve.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
255 CVARIABLE MAXIN
```

LEGAL–CHARS (– A)

A string variable that will specify the characters GET–INPUT will allow to be entered.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of LEGAL–CHARS.

Example of Use:

```
DIGITS LEGAL–CHARS $!
```

The string DIGITS is defined as "0123456789–". Storing the string DIGITS in LEGAL–CHARS will cause GET–INPUT to allow only numeric input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
128 $VARIABLE LEGALCHARS
```

INPUT (– A)

A string variable that will hold the string gotten by GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of INPUT.

Example of Use:

```
GET–INPUT INPUT $?
```

This code would print the value obtained by GET–INPUT on the display.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
255 $VARIABLE INPUT
```

OK–TO–BEEP (– A)

A Boolean variable, true if GET–INPUT should beep on an error condition, false if it should not.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of OK–TO–BEEP.

Example of Use:

```
OK–TO–BEEP C1SET
```

Setting OK–TO–BEEP to true will cause GET–INPUT to beep if an illegal character is entered, or if too many or too few characters are entered.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE OK-TO-BEEP
```

KILL-CHAR (- N)

A byte constant for the keyboard character that will erase a line in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of KILL-CHAR.

Example of Use: See GET-INPUT.

Algorithm: The KILL-CHAR specified by this constant will cause GET-INPUT to erase all the characters typed in so far, and place the cursor back to the start of the input field. KILL-CHAR has been set for the escape key on the IBM-PC keyboard.

Suggested Extensions: KILL-CHAR could be changed to use another key if desired.

Definition:

```
27 CCONSTANT KILL-CHAR
```

RETURN-CHAR (- N)

A byte constant for the keyboard character that will cause GET-INPUT to accept the characters typed in.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of RETURN-CHAR.

Example of Use: See GET-INPUT.

Algorithm: GET-INPUT accepts the RETURN-CHAR to end input.

Suggested Extensions: RETURN-CHAR could be changed to another value for special situations. For example, the plus (+) key could be used on the IBM-PC for input using the numeric keypad.

Definition:

```
13 CCONSTANT RETURN-CHAR
```

BACKSPACE (– B)

A byte constant for the keyboard character that will cause GET-INPUT to back up one space.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of BACKSPACE.

Example of Use: See GET-INPUT.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
8 CCONSTANT BACKSPACE
```

ASCII-CHAR? (C – F)

Determine if a character is a printable ASCII character.

Stack on Entry: (C) The character to check.

Stack on Exit: (F) – A flag, true if the character was a printable ASCII character, false otherwise.

Example of Use:

```
KEY ASCII-CHAR?
```

This code would print a –1 if the value returned by KEY was a legal ASCII value, otherwise it would print a zero.

Algorithm: See if the value of the character lies in the range 32–128.

Suggested Extensions: None.

Definition:

```
: ASCII-CHAR? C(C-F)
   DUP 31 > SWAP 129 < AND;
```

LEGAL? (C - F)

Determine if a character will be valid for GET-INPUT.

Stack on Entry: (C) The character to check.

Stack on Exit: (F) A flag, true if the character is valid for GET-INPUT, false otherwise.

Example of Use:

```
KEY LEGAL?
```

This code would print a −1 if the value returned by KEY was valid for GET-INPUT; otherwise, it would print a zero.

Algorithm: If the string LEGAL-CHARS is empty, all characters are valid. If the string is not empty, check to see if the character is contained in the string.

Suggested Extensions: None.

Definition:

```
: LEGAL? C( C - F)
LEGAL-CHARS $@ LEN 0= IF
   DROP -1
ELSE
   CHR$ LEGAL-CHARS FIND$ NOT NOT
ENDIF ;
```

NULL$SET (A -)

Set a string variable to the empty string.

Stack on Entry: (A) The address of the string variable.

Stack on Exit: Empty.

Example of Use:

```
LEGAL-CHARS NULL$SET
```

This would set the string LEGAL-CHARS to the empty string. This would cause GET-INPUT to allow any ASCII character to be input.

Algorithm: Store a zero in the string's length byte.

Suggested Extensions: None.

Definition:

```
: NULL$SET 1+ C0SET ;
```

CHOP1 (A -)

Remove the rightmost character from a string.

Stack on Entry: (A) The address of the string.

Stack on Exit: Empty.

Example of Use:

```
MY-NAME $@ CHOP1 MY-NAME $?
```

If the string MY-NAME contained the value "JAMES" before the above code was executed, "JAME" would be printed on the display by the above code.

Algorithm: Take the left string using the length minus one.

Suggested Extensions: None.

Definition:

```
: CHOP1 $@ DUP LEN 1- LEFTS ;
```

DESTRUCT (-)

Erase a character at the current cursor position and place the cursor one space to the left of the current position.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Emit a blank to wipe out the current character. Then emit two backspaces to place the cursor one to the left of the former current character.

Suggested Extensions: None.

Definition:

```
: DESTRUCT BL EMIT BACKSPACE EMIT
   BACKSPACE EMIT ;
```

RETURN-OK? (- F)

Leave a true flag if a return is legal in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if return is legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If the length of the input string INPUT is greater than or equal to the minimum number of characters allowed, as specified by MININ, then a return is valid.

Suggested Extensions: None.

Definition:

```
: RETURN-OK?
   INPUT $@ LEN MININ C@ >= ;
```

BACKSPACE-OK? (- F)

Leave a true flag if a backspace is legal in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if a backspace is legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If we are not at the left hand side of the input field, then a backspace is legal. INPUT will be an empty string, with a length of zero, if we are at the left–hand side of the input field.

Suggested Extensions: None.

Definition:

```
: BACKSPACE-OK?
   INPUT $@ LEN 0 <> ;
```

ANY? (– F)

Leave a true flag if any characters are legal in GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if any characters are legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If we are at the right-hand side of the input field, no more characters can be allowed. This condition exists when the length of INPUT is equal to the value in MAXIN.

Suggested Extensions: None.

Definition:

```
: ANY?
   INPUT $@ LEN MAXIN C@ <> ;
```

/KILL/ (–)

Erase the current input line and move the cursor to the start of the input field.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If INPUT is not a null string, loop through its length, wiping out each character on the display. If INPUT is a null string, the line is already empty, so don't do anything.

Suggested Extensions: None.

Definition:

```
: /KILL/
   INPUT $@ LEN ?DUP IF
      0 DO DESTRUCT LOOP
      BL EMIT BACKSPACE EMIT
   ENDIF
   INPUT NULL$SET ;
```

/OK/ (C –)

Store a valid character in the string INPUT.

Stack on Entry: (C) The character to store in INPUT.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the string concatenation word to append the character onto INPUT.

Suggested Extensions: None.

Definition:

```
: /OK/ C( C – )
   DUP EMIT CHR$ INPUT $+! ;
```

/BACKPSACE/ (–)

Handle a backspace for GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Wipe out a character on the screen, then remove it from INPUT.

Suggested Extensions: None.

Definition:

```
: /BACKSPACE/
   DESTRUCT INPUT CHOP1 ;
```

/INIT/ (–)

Initialize values for GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: The way GET–INPUT is currently defined, the only initialization necessary is to set the string INPUT to the empty string.

Suggested Extensions: None.

Definition:

```
: /INIT/
   INPUT NULL$SET ;
```

?BEEP (–)

Beep if the variable OK–TO–BEEP holds a true flag.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Check the variable, and emit an ASCII bell character if it is true.

Suggested Extensions: None.

Definition:

```
: ?BEEP OK-TO-BEEP C@ IF 7 EMIT ENDIF ;
```

GET-INPUT (-)

Get input from the keyboard.

Minimum length, maximum length, and restricted characters are supported.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
1 MININ C! 1 MAXIN C! DIGITS LEGAL-CHARS $! GET-INPUT
```

This code would get input from the keyboard. It would require one character, and only allow digits to be input. The string input, in this case only a single character in length, would be returned in INPUT.

Algorithm: Begin by initializing the necessary values. Then, start an endless loop of inputting a character from the keyboard. First, check if the key hit was the KILL-CHAR; if it was, call /KILL/ to wipe out the current line. Next, check for the RETURN-CHAR. If the RETURN-CHAR was hit and a return is valid, exit the word. This is the only exit from the endless loop. Check for the BACKSPACE character and execute /BACKSPACE/ if it was found and a backspace is valid. Next, see if any character can be entered. If not continue the loop. If characters can be input, make sure we have a valid ASCII character. If the character is valid, store it in the string INPUT and continue the loop.

Suggested Extensions: The keyboard macros mentioned in the Introduction are one possible extension to this word. Another extension might be to allow the user to escape from the input without hitting return. A function key could

be used. This would give programs that use GET-INPUT the ability to allow users to exit from input screens easily.

Definition:

```
: GET-INPUT
  /INIT/ BEGIN
    KEY DUP KILL-CHAR = IF
      DROP /KILL/
    ELSE
      DUP RETURN-CHAR = IF
        DROP RETURN-OK? IF
          EXIT
        ELSE
          ?BEEP
        ENDIF
      ELSE
        DUP BACKSPACE = IF
          DROP BACKSPACE-OK? IF
            /BACKSPACE/
          ELSE
            ?BEEP
          ENDIF
        ELSE
          ANY? NOT IF
            DROP ?BEEP
          ELSE
            DUP ASCII-CHAR? NOT IF
              DROP ?BEEP
            ELSE
              DUP LEGAL? NOT IF
              01DROP ?BEEP
              ELSE
              1/OK/
              ENDIF
            ENDIF
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  0 UNTIL ;
```

0$ (A -)

Zero the unused portions of a string variable.

Stack on Entry: (A) The address of the string variable.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the length of the string and how much space is allocated to it to determine how many bytes to zero out. This is done so that the word >BINARY will terminate properly when passed a string.

Suggested Extensions: None.

Definition:

```
: 0$ DUP C@ OVER 1+ C@ - SWAP
   DUP 1+ C@ + 2+ SWAP ERASE ;
```

INT-INP (- N)

Input an integer.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input.

Example of Use:

```
1 MININ C! 20 MAXIN C! INT-INP
```

This would get an integer from the keyboard; at least one, and a maximum of twenty characters would be input.

Algorithm: Set the legal characters to the digits and the negative sign. Start a loop calling GET-INPUT. Take the string returned and see if it has a leading negative sign. If it does, place a negative one on the return stack. If it does not place a one there. Call >BINARY to convert the string to an integer. If >BINARY has stopped on a zero byte (not an ASCII zero), then the conversion was successful. If the conversion was successful, multiply the result by the one or negative one placed on the return stack, and leave a true on the stack to terminate the loop. If the conversion was unsuccessful, clear the stacks and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET-INPUT will be executed.

Suggested Extensions: None.

Definition:

```
$CONSTANT DIGITS -0123456789"
45 CONSTANT .KEY

: INT-INP
  DIGITS LEGAL-CHARS $! BEGIN
    GET-INPUT INPUT 0$
    INPUT $@ DUP 1+ C@ .KEY = IF
      -1 >R 1+
    ELSE
      1 >R
    ENDIF
    0. ROT >BINARY C@ .KEY = IF
      R> DROP 2DROP ?BEEP /KILL/ 0
    ELSE
      DROP R> * -1
    ENDIF
  UNTIL ;
```

DINT-INP (- D)

Input a double-length integer.

Stack on Entry: Empty.

Stack on Exit: (D) The double-length integer input.

Example of Use:

```
1 MININ C! 20 MAXIN C! DINT-INP D
```

This would get a double-length integer from the keyboard; at least one, and a maximum of twenty characters would be input. The result would then be printed on the display.

Algorithm: Set the legal characters to the digits and the negative sign. Start a loop calling GET-INPUT. Take the string returned and see if it has a leading negative sign. If it does, place a negative one on the return stack. If it does not, place a one there. Call >BINARY to convert the string to an integer. If >BINARY has stopped on a zero byte (not an ASCII zero), then the conversion was successful. If the conversion was successful multiply the result by the one or negative one placed on the return stack and leave a true on the stack to terminate the loop: If the conversion was unsuccessful clear the stacks and

call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET-INPUT will be executed.

Suggested Extensions: None.

Definition:

```
: DINT-INP
   DIGITS LEGAL-CHARS $! BEGIN
      GET-INPUT INPUT 0$
   INPUT $@ DUP 1+ C@ .KEY = IF
      -1 >R 1+
   ELSE
      1 >R
   ENDIF
   0, ROT >BINARY C@ .KEY = IF
      R> DROP 2DROP ?BEEP /KILL/ 0
   ELSE
      R> 1 M*/ -1
   ENDIF
UNTIL ;
```

FP-INP (- R)

Input a floating-point number.

Stack on Entry: Empty.

Stack on Exit: (R) The floating-point number input.

Example of Use:

```
1 MININ C! 8 MAXIN C! FP-INP R.
```

This would get a floating-point number from the keyboard, at least one, and a maximum of twenty characters would be input. The result would then be printed on the display.

Algorithm: This word requires the floating-point number words to have been loaded. Set the legal characters to the digits, the negative sign and the decimal point. Start a loop calling GET-INPUT. Pass the INPUT string to FNUM. FNUM leaves a flag on the stack indicating whether or not it was able to convert the string to a floating point number. If the conversion was successful, leave a true on the stack to terminate the loop. If the conversion was unsuc-

cessful, clear the stacks and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET-INPUT will be executed.

Suggested Extensions: None.

Definition:

```
$CONSTANT FDIGITS 0123456789-."

    : FP-INP

    FDIGITS LEGAL-CHARS $I BEGIN
      GET-INPUT INPUT 0$
      INPUT $@ FNUM IF
        -1
      ELSE
        ?BEEP /KILL/ 0
      ENDIF
  UNTIL ;
```

MINVAL (- A)

A variable used to specify the minimum allowed integer for INT-BOUNDED-INP.

Stack on Entry: Empty.

Stack on Exit: (A) - The address of MINVAL.

Example of Use:

```
1 MINVAL !
```

Storing a one in MINVAL will cause INT-BOUNDED-INPUT to only allow positive numbers to be input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
-32767 VARIABLE MINVAL
```

MAXVAL (- A)

A variable used to specify the maximum allowed integer for INT-BOUNDED-INP.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MAXVAL.

Example of Use:

```
0 MAXVAL !
```

Storing a zero in MAXVAL will cause INT-BOUNDED-INPUT to only allow negative numbers to be input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
32767 VARIABLE MAXVAL
```

INT-BOUNDED-INP (- N)

Input an integer that falls within a specified range.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input.

Example of Use:

```
7 MINVAL ! 8 MAXVAL ! 1 MININ C! 1 MAXIN C! INT-BOUNDED-INP
```

This code would force a seven or an eight to be input by setting the minimum value to seven and the maximum value to eight.

Algorithm: Start a loop calling INT-INP. Check the returned value against MINVAL and MAXVAL. If the result does not lie between them, clear the stack and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Continue the loop. If the result was within the proper range, exit the loop.

Suggested Extensions: Clone this word for double-length or floating-point numbers as needed.

Definition:

```
: INT-BOUNDED-INP
   BEGIN
      INT-INP DUP DUP MINVAL @ >=
      SWAP MAXVAL @ <= AND NOT
   WHILE
      ?BEEP /KILL/ DROP
   REPEAT ;
```

MONTH (- N)

Prompt the user for, and input, a month.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input, in the range 1 to 12.

Example of Use: See MDY-INPUT below.

Algorithm: Set the minimum field length to one, the maximum to two. Set the minimum input value to one, the maximum to twelve. Print out the prompt and call INT-BOUNDED-INP.

Suggested Extensions: None.

Definition:

```
: MONTH
   1 MININ C! 2 MAXIN C!
   1 MINVAL ! 12 MAXVAL !
   ." MONTH: " INT-BOUNDED-INP ;
```

DAY (N1 - N2)

Prompt the user for, and input, the day.

Stack on Entry: (N1) The month as returned by MONTH.

Stack on Exit: (N2) The day input.

Example of Use: See MDY-INPUT below.

Algorithm: Set the minimum field length to one, the maximum to two. Set the minimum input value to one. Look up the maximum value in the array DPM (days per month). Print out the prompt and call INT-BOUNDED-INP.

Suggested Extensions: None.

Definition:

```
31 CVARIABLE DPM 29 C, 31 C, 30 C,
31 C, 30 C, 31 C, 31 C, 30 C,
31 C, 30 C, 31 C,

: DAY
   1- DPM + C@ MAXVAL !
   ." DAY: " INT-BOUNDED-INP ;
```

YEAR (- N)

Prompt the user for, and input, the year.

Stack on Entry: Empty.

Stack on Exit: (N) The year input.

Example of Use: See MDY-INPUT below.

Algorithm: Set the minimum field length to one, the maximum to five. Call INT-INP to get an integer.

Suggested Extensions: None.

Definition:

```
: YEAR
   1 MININ C! 5 MAXIN C!
   ." YEAR: " INT-INP ;
```

MDY-INP (- N1 N2 N3)

Input a date.

Stack on Entry: Empty.

Stack on Exit: (N1) The year input.

(N2) The day input.
(N3) The month input.

Example of Use: None.

Algorithm: First call MONTH. Then call DAY with a copy of the month. Finally, call YEAR.

Suggested Extensions: None.

Definition:

```
: MDY-INP
   MONTH DUP DAY YEAR ;
```

AM/PM (N1 – N2)

Ask for the strings AM or PM, and adjust the time on the stack accordingly.

Stack on Entry: (N1) A military time in the range zero to 1159.

Stack on Exit: (N2) The time adjusted for AM or PM.

Example of Use: See TIME-INP below.

Algorithm: Loop until either "AM" or "PM" is input by the user. If "PM" is input, add 1200 to the time on the stack.

Suggested Extensions: None.

Definition:

```
$CONSTANT AMPM$ APM"
$CONSTANT AM$ AM"
$CONSTANT PM$ PM"

:AM/PM?
   AMPM$ LEGAL-CHARS $!
   2 MININ C! 2 MAXIN C!
   ." AM OR PM: " BEGIN
      GET-INPUT
      INPUT $@ DUP AM$ $= SWAP
      PM$ $= OR NOT
   WHILE
```

```
   ?BEEP /KILL/
REPEAT
INPUT $@ PM$ $= IF
   1200 +
ENDIF ;
```

HOUR (– N)

Prompt for, and input, the hour from the user.

Stack on Entry: Empty.

Stack on Exit: (N) A military time for the hour input.

Example of Use: See TIME–INP below.

Algorithm: Allow only values from one to twelve to be input. If a twelve is input, convert it to zero. Multiply the value by 100.

Suggested Extensions: None.

Definition:

```
: HOUR
   1 MININ C! 2 MAXIN C!
   20 MINVAL ! 12 MAXVAL !
   ." HOUR: " INT-BOUNDED-INP
   DUP 12 = IF 12 - ENDIF
   100 * ;
```

MINUTE (N1 – N2)

Prompt for, and input, the minute from the user.

Stack on Entry: (N1) A military time for the hour just input.

Stack on Exit: (N2) The minutes added to the time.

Example of Use: See TIME–INP below.

Algorithm: Allow only values from zero to 59 to be input.

Suggested Extensions: None.

Definition:

```
: MINUTE
   59 MAXVAL !
   ." MINUTE: " INT-BOUNDED-INP
   + ;
```

TIME-INP (- N)

Input a time in military format.

Stack on Entry: Empty.

Stack on Exit: (N) The time in military format.

Example of Use: None.

Algorithm: Call HOUR, MINUTE, and AM/PM?.

Suggested Extensions: None.

Definition:

```
: TIME-INP
   HOUR MINUTE AM/PM? ;
```

(Y/N) (- F)

Give the user the prompt "(Y/N)" and input a Y or N key. Leave a true flag if a "Y" was hit, false if a "N" was hit.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if "Y" was hit, false if "N" was hit.

Example of Use: (Y/N)

This code would print a −1 if "Y" is hit, zero if "N" is hit after the prompt "(Y/N)" is printed on the display.

Algorithm: Print the prompt. Start a loop of inputting a key.

Convert the key hit to uppercase. If it is a "Y" or an "N," exit the loop. EMIT

the character onto the display. Compare the character to "Y" and leave the flag on the stack.

Suggested Extensions: None.

Definition:

```
: (Y/N) ." (Y/N) " 1 BEGIN
      DROP KEY DUP 95 > IF 32 - ENDIF
      DUP DUP 89 = SWAP 78 = OR
   UNTIL
   DUP EMIT 89 = ;
```

SINGLE-KEY (- C)

Allow a user to hit a single key, specified by the characters in the string LEGAL-CHAR.

Stack on Entry: Empty.

Stack on Exit: (C) The character hit.

Example of Use:

```
ABC LEGAL-CHAR $!
." RATE MEAL (A/B/C) " SINGLE-KEY
```

The string ABC is "ABC'. This code would allow the user to press either "A", "B", or "C" after the prompt "RATE MEAL (A/B/C)" was printed on the display.

Algorithm: Start a loop of inputting a key. Convert the key to uppercase. See if the key is in the string LEGAL-CHARS. If it is not, continue the loop. If it is, exit the loop and EMIT the character on the display. Leave the key on the stack.

Suggested Extensions: None.

Definition:

```
: SINGLE-KEY 1 BEGIN
      DROP KEY DUP 95 > IF 32 - ENDIF
      DUP CHRS LEGAL-CHARS FINDS
   UNTIL DUP EMIT ;
```

$->UPPER (A -)

Convert a string to all uppercase.

Stack on Entry: (A) The address of the string to convert.

Stack on Exit: Empty.

Example of Use:

```
CITY $->UPPER CITY $?
```

If the string CITY contained "New York" prior to the above code being executed, then the value "NEW YORK" would be printed on the display by the above code.

Algorithm: Loop through the string. If a character has an ASCII code above 95, it is lowercase. Subtract 32 to convert it to uppercase and store it back in the string.

Suggested Extensions: None.

Definition:

```
: $->UPPER
  DUP C@ 0 DO
    1+ DUP C@ 95 > IF -32 OVER C+! ENDIF
  LOOP DROP ;
```

Displays and Output Formating

Words Defined in This Chapter:

ATTRIBUTE	A byte variable holding the current attribute byte.
WT []	Access window table array.
FREE_TABLE	Find a free entry in window table.
CREATE_ WINDOW	Create a window.
W_TABLE_ ROTATE	Rearrange the order of the window table.
SAVE_ WINDOW	Save the contents of a window.
RESTORE_ WINDOW	Restore the contents of a window.
SCROLL	Scroll the current window.
WEMIT	Emit a character onto the current window.
WTYPE	Type a string of characters onto the current window.

W"	Enclose a literal to be printed on the current window.
WQUERY	Input a line on the current window.
NORMAL	Display normal characters.
REVERSE	Display reverse or inverse characters.
BLINKING	Display blinking characters.
BLUE	Display blue characters.
GREEN	Display green characters.
RED	Display red characters.
CYAN	Display cyan characters.
BROWN	Display brown characters.
MAGENTA	Display magenta characters.
GRAY	Display gray characters.
L__BLUE	Display light blue characters.
L__GREEN	Display light green characters.
L__CYAN	Display light cyan characters.
L__RED	Display light red characters.
L__MAGENTA	Display light magenta characters.
YELLOW	Display yellow characters.
BRIGHT__ WHITE	Display bright white characters.

In this chapter we present a complete set of words to manage display windows on your IBM-PC. Normally the screen of your IBM-PC consists of 25 lines of 80 characters. A window is a rectangular portion of that screen in which all character input and output will take place. A window could be as large as the normal screen, or as small as one line of one character. The windows we will define can overlay each other and be stacked to any depth. The use of windows has become popular in computers such as Apple's Macintosh. Windows have proved to be an intuitive model of how people work, and with these words, the IBM-PC can exploit their full power.

We will make extensive use of the extra memory available on the IBM-PC for these window words. We will use a segment outside the normal 64K address space of Forth to hold saved windows. We will use another segment to hold text literals to be displayed on windows. The actual addresses these segments should reside at are highly dependent on your particular version of Forth. The code presented uses the word >X, which returns the segment address of the basic Forth system in Atila. They then use the memory after the base 64K of the language as the extra segments needed. Most IBM-PC Forth's should have a word similar to >X; consult the manual that came with your version.

The windows we define will be able to overlay each other without restriction. In order to make this possible, we will save the complete contents of all

windows that are not currently displayed. The following values will be kept for each window:

Upper Left X Coordinate (0–79)
Upper Left Y Coordinate (0–24)
X Cursor Position (0–79)
Y Cursor Position (0–24)
Current Width of window (1–80)
Current Height of window (1–25)
Maximum Width of window (1–80)
Maximum Height of window (1–25)
Offset screen stored at in extra segment.
Integer identifier for window.

These values will be kept in an array, W_DATA. This array will be accessed by a number of words we define below. The current window being used for character input and output will be kept in the first portion of the array. The identifier given to each window will enable them to be accessed by name.

Suggested extensions: The window word set presented is fairly complete. One extension that might be aesthetically pleasing would be to automatically box all created windows with the double-line characters provided in the IBM character set. This could be done by changing created windows to reduce all dimensions by two, and adding the box drawing to the MAKE_CURRENT word.

Variables Used by Window Words:

```
( Physical screen segment )
( Use B800 for color display )
HEX B000 VARIABLE SCREEN_SEG DECIMAL

( Segment used to save windows )
HEX > X 2000 + VARIABLE SAVE_SEG DECIMAL

( Segment Offset used to save windows )
1 VARIABLE OFFSET

( Physical width of screen )
( Set to 40 on certain displays )
80 CVARIABLE PHYS_WIDTH

( Current attribute byte )
3 CVARIABLE ATTRIBUTE
```

```
( Maximum number of windows )
( Use any value, constrained only by memory)
7 CCONSTANT MAX_WIND

( Window table width )
11 CCONSTANT W_T_W

( Window data table )
0 VARIABLE W_DATA MAX_WIND W_T_W * DUP
ALLOT W_DATA SWAP ERASE
```

Array Accessing Words:

```
( An offset is defined for each table entry)
0 CCONSTANT X_UP_LEFT_OFS
1 CCONSTANT Y_UP_LEFT_OFS
2 CCONSTANT X_CUR_OFS
3 CCONSTANT Y_CUR_OFS
4 CCONSTANT X_CUR_LEN_OFS
5 CCONSTANT Y_CUR_LEN_OFS
6 CCONSTANT X_MAX_LEN_OFS
7 CCONSTANT Y_MAX_LEN_OFS
8 CCONSTANT OFS_OFS
10 CCONSTANT W_NAME_OFS

( These words allow access to the first array entry)
: FIELD <BUILDS C, DOES> C@ W_DATA + ;
X_UP_LEFT_OFS FIELD X_UP_LEFT
Y_UP_LEFT_OFS FIELD Y_UP_LEFT
XCUR_OFS FIELD XCUR
YCUR_OFS FIELD YCUR
X_CUR_LEN_OFS FIELD X_CUR_LEN
Y_CUR_LEN_OFS FIELD Y_CUR_LEN
X_MAX_LEN_OFS FIELD X_MAX_LEN
Y_MAX_LEN_OFS FIELD Y_MAX_LEN
OFS_OFS FIELD OFS
W_NAME_OFS FIELD W_NAME
```

WT [] (N - A)

Find the address of an entry in the window data array.

Stack on Entry: (N) Index to window table.

Stack on Exit: (A) Address of Nth entry in array.

Example of Use:

This would leave the address of the third window's data on the stack.

Algorithm: Multiply value by size of each array entry, then add the start of the array.

Suggested extensions: None.

Definition:

```
: WT [] W__T__W * W__DATA + ;
```

FREE__TABLE (– N)

Find a free entry in the window data array.

Stack on Entry: Empty.

Stack on Exit: (N) Index number of a free entry.

Example of Use:

```
FREE__TABLE .
```

This code would print the index number of a free entry in the window data array.

Algorithm: Place a 9999 on the stack. Loop through the array, looking for an empty name field; this signifies a free entry. If one is found, drop the 9999, leave its number on the stack, and exit the loop. At the end of the loop, see if the top of the stack is 9999; if it is, no free entries were found.

Suggested Extensions: None.

Definition:

```
: FREE__TABLE
  9999 MAX__WIND 0 DO
    I WT [] W__NAME__OFS +
    C@ 0= IF
      DROP I LEAVE
    ENDIF
```

```
LOOP
DUP 9999 = ABORT" Too many windows." ;
```

CREATE_WINDOW (N1 N2 N3 N4 N5 –)

Create an entry in the window data array for a new window.

Stack on Entry: (N1) – Upper left X coordinate of window.
(N2) – Upper left Y coordinate of window.
(N3) – Maximum width of window.
(N4) – Maximum height of window.
(N5) – Integer identifier for window.

Stack on Exit: Empty.

Example of Use:

```
5 5 40 15 5 CREATE_WINDOW
```

This code would create a window of 15 lines of 40 characters. The window would start 5 characters from the left edge of the physical screen, 5 lines from the top of the physical screen. The window would be referenced by the number five.

Algorithm: Search for a free table entry. Erase it when found, and then fill it in with the passed values.

Suggested Extensions: Check for duplicate identifiers.

Definition:

```
: CREATE_WINDOW
   FREE_TABLE WT []
   DUP W_T_W ERASE
   DUP >R W_NAME_OFS + C!
   R> DUP >R Y_MAX_LEN_OFS + C!
   R> DUP >R X_MAX_LEN_OFS + C!
   R> DUP >R Y_UP_LEFT_OFS + C!
   R> X_UP_LEFT_OFS + C! ;
```

W_TABLE_ROTATE (N –)

Rotate a window to the top of the window data array.

Stack on Entry: (N) Entry number to be rotated to the top.

Stack on Exit: Empty.

Example of Use:

```
2 W__TABLE__ROTATE
```

Rotate the third entry to the top of the window data array.

Algorithm: If the top is being rotated to itself, do nothing. Otherwise, save the entry to be rotated to the top in the PAD. Then, move the entries under that entry up by one entry each. When this is complete, move the saved entry from the PAD to the top of the array.

Suggested extensions: None.

Definition:

```
: W__TABLE__ROTATE
  ?DUP IF
    DUP WT [] PAD W__T__W CMOVE
    W__DATA DUP W__T__W + ROT W__T__W *
    <CMOVE
    PAD W__DATA W__T__W CMOVE
  ENDIF ;
```

SAVE__WINDOW

Save the contents of the current window to the save segment area.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
SAVE__WINDOW
```

This code would save the contents of the current window.

Algorithm: Loop on the current height of the window. Determine the start of each line using the upper left coordinates and then move the data to the extra segment using XCMOVE.

Suggested Extensions: None.

Definition:

```
: SAVE_WINDOW
  Y_CUR_LEN C@ 0 DO
    Y_UP_LEFT C@ I + PHYS_WIDTH C@ 2* *
    2X_UP_LEFT C@ 2* + SCREEN_SEG @
    OFS @ X_CUR_LEN C@ 2* I * +
    SAVE_SEG @
    X_CUR_LEN C@ 2* XCMOVE
  LOOP ;
```

RESTORE_WINDOW

Restore the contents of the current window from the save segment area.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RESTORE_WINDOW
```

This code would restore the contents of the current window.

Algorithm: Loop on the current height of the window. Determine the start of each line using the upper left coordinates and then move the data from the extra segment using XCMOVE.

Suggested Extensions: None.

Definition:

```
: RESTORE_WINDOW
  Y_CUR_LEN C@ 0 DO
    OFS @ X_CUR_LEN C@ 2* I * +
    SAVE_SEG @
    Y_UP_LEFT C@ I + PHY_SWIDTH C@ 2* *
    X_UP_LEFT C@ 2* + SCREEN_SEG @
    X_CUR_LEN C@ 2* XCMOVE
  LOOP ;
```

SCROLL

Scroll the current window.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
SCROLL
```

This code would scroll the current window.

Algorithm: Move each line up one by using the screen segment and X-CMOVE. Blank the bottom line using the constant BLANK, which is a blank character and a normal attribute.

Suggested Extensions: None.

Definition:

```
HEX 0320 CONSTANT BLANK DECIMAL

: LINE-ADDR
   Y_UP_LEFT C@ +
   PHYS_WIDTH C@ 2* *
   X_UP_LEFT C@ 2* +
   SCREEN_SEG @ ;

: SCROLL
   Y_CUR_LEN C@ 1 DO
      I LINE-ADDR
      I 1- LINE-ADDR
      X_CUR_LEN C@ 2* XCMOVE
   LOOP
   Y_CUR_LEN C@ 1- LINE-ADDR DROP
   X_CUR_LEN C@ 0 DO
      BLANK OVER
      SCREEN-SEG @ X! 2+
   LOOP DROP ;
```

WEMIT (C -)

Print a character on the current window.

Stack on Entry: (C) Character to print.

Stack on Exit: Empty.

Example of Use:

```
65 WEMIT
```

This code would print an 'A' on the current window at the current cursor position.

Algorithm: Check to see if the character is a carriage return. If it is, check to see if we must scroll. Set the X cursor position to zero. If the character was not a carriage return, place it on the screen. Increment the X cursor position. If it reaches the end of a line on the window, set it to zero and check for a possible scroll. If scroll is not needed, increment the Y cursor position.

Suggested Extensions: Add support for the backspace character.

Definition:

```
: WEMIT
  DUP 13 = IF
    Y_CUR_LEN C@ 1- YCUR C@ = IF
      SCROLL XCUR C0SET
    ELSE
      1 YCUR C+! XCUR C0SET
    ENDIF
  ELSE
    YCUR C@ LINE-ADDR DROP XCUR C@ 2* +
    DUP >R SCREEN-SEG @ XC!
    ATTRIBUTE C@ R> 1+ SCREEN-SEG @ XC!
    1 XCUR C+! XCUR C@ X_CUR_LEN C@ = IF
      Y_CUR_LEN C@ 1- YCUR C@ = IF
        SCROLL XCUR C0SET
      ELSE
        1 YCUR C+! XCUR C0SET
      ENDIF
    ENDIF
  ENDIF ;
```

WTYPE (A N -)

Print a string of characters on the current window.

Stack on Entry: (A) Address of characters to print.

(N) Number of bytes to print.

Stack on Exit: Empty.

Example of Use:

```
DISK_NAME 15 WTYPE
```

This code would print the 15 characters at DISK_NAME on the current window.

Algorithm: Make sure the count is greater than zero. If it is, loop through the characters calling WEMIT to print them on the display.

Suggested Extensions: None.

Definition:

```
: WTYPE
   DUP 0= IF EXIT ENDIF
   0 DO I OVER + C@ WEMIT LOOP DROP ;
```

W"

Print a literal on the current window.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: FINISH W" This is the end! " ;
```

This word would print "This is the end! " on the current window, when it is executed. W" can only be used inside a word definition.

Algorithm: Use WORD to parse the string. Store it in the extra memory pointed to by OFS_LITERAL and SEG_LITERAL. Cause the offset the word was saved at to be placed on the stack when the word is executed. W"EX will use this address and move the literal to the PAD. From the PAD the literal will be printed.

Suggested Extensions: None.

Definition:

```
HEX >X 2200 + VARIABLE SEG__LITERAL
0 VARIABLE OFS__LITERAL
DECIMAL

: W"EX SEG__LITERAL @ PAD >X 64 XCMOVE
   PAD COUNT WTYPE ;

34 CCONSTANT L"
: W" L" WORD DUP C@ 1+ >R
   >X OFS__LITERAL @ SEG__LITERAL @
   R> DUP >R XCMOVE
   OFS__LITERAL @ [COMPILE] LITERAL
   R> OFS__LITERAL +!
   COMPILE W"EX ;
IMMEDIATE
```

CLEAR__WINDOW

Fill the current window with blanks.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
CLEAR__WINDOW
```

This code would clear the current window.

Algorithm: Loop through the screen height and width storing the constant BLANK.

Suggested Extensions: Make a faster version by filling PAD with BLANK and usung XCMOVE.

Definition:

```
: CLEAR__WINDOW
     Y__CUR__LEN C@ 0 DO
       I LINE-ADDR DROP
```

```
    X_CUR_LEN C@ 0 DO
      BLANK OVER
      SCREEN-SEG @ X! 2+
    LOOP DROP
  LOOP ;
```

MAKE_CURRENT (N -)

Cause a specific window to become the current window.

Stack on Entry: (N) Identifier of window to be made current.

Stack on Exit: Empty.

Example of Use:

```
5 MAKE_CURRENT
```

This code would cause the window with the identifier five to become current.

Algorithm: Find the window in the table. Once found, determine whether this window has ever been used before by checking the offset field. If the offset is empty, this is the first time for this window. If this is the case, set the height and width to the their maximum sizes and clear the window. Then, allocate space in the save segment for the window. If the window had been displayed previously, restore the data from the save area to the screen.

Suggested Extensions: None.

Definition:

```
: MAKE_CURRENT
  9999 SWAP MAX_WIND 0 DO
    I WT [] W_NAME_OFS +
    C@ OVER = IF
      2DROP I I LEAVE
    ENDIF
  LOOP
  SWAP 0099 = ABORT" Window not found."
  SAVE_WINDOW
  W_TABLE_ROTATE
  OFS @ 0= IF
    Y_MAX_LEN C@ DUP Y_CUR_LEN C!
```

```
    X_MAX_LEN C@ DUP X_CUR_LEN C!
    XCUR C0SET YCUR C0SET
    2* * OFFSET @ OFS ! OFFSET +!
    CLEAR_WINDOW
  ELSE
    RESTORE_WINDOW
  ENDIF ;
```

WQUERY (–)

Input a string of characters from the keyboard and display it on the current window. Store it in the terminal input buffer.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
WQUERY
```

This code would allow a string of characters to be input.

Algorithm: Erase the input buffers and store a zero in the buffer position variable. Input characters until encountering a carriage return. If the end of the buffer is reached, force a carriage return.

Suggested Extensions: Implement a backspace.

Definition:

```
64 CCONSTANT MAX_CHAR
0 CVARIABLE TIB_POS

: WQUERY
  TIB MAX_CHAR ERASE TIB_POS C0SET BEGIN
    KEY DUP 13 <> TIB_POS C@ MAX_CHAR <> AND
  WHILE
    DUP WEMIT TIB TIB_POS C@ + C!
    1 TIB_POS C+!
  REPEAT DROP ;
```

Various Display Control Words:

These words modify the display attribute byte to allow various types of text output. Most are self-explanatory.

```
( Normal display, white on black )
: NORMAL 7 ATTRIBUTE C! ;

( Reverse, black on white)
: REVERSE 112 ATTRIBUTE C! ;

( Blinking, added to current )
: BLINKING ATTRIBUTE C@ 128 AND ATTRIBUTE C! ;

( Various colors )
: BLUE 1 ATTRIBUTE C! ;
: GREEN 2 ATTRIBUTE C! ;
: RED 4 ATTRIBUTE C! ;
: CYAN 3 ATTRIBUTE C! ;
: BROWN 6 ATTRIBUTE C! ;
: MAGENTA 5 ATTRIBUTE C! ;
: GRAY 8 ATTRIBUTE C! ;
: L__BLUE 9 ATTRIBUTE C! ;
: L__GREEN 10 ATTRIBUTE C! ;
: L__CYAN 11 ATTRIBUTE C! ;
: L__RED 12 ATTRIBUTE C! ;
: L__MAGENTA 13 ATTRIBUTE C! ;
: YELLOW 14 ATTRIBUTE C! ;
: BRIGHT__WHITE 15 ATTRIBUTE C! ;
```

10

Natural Language Processing

Words Defined in This Chapter:

VERB	A constant for the part-of-speech verb.
NOUN	A constant for the part-of-speech noun.
DET	A constant for the part-of-speech determinant.
ADJ	A constant for the part-of-speech adjective.
ADVERB	A constant for the part-of-speech adverb.
CONJ.	A constant for the part-of-speech conjunction.
WORD#	A variable used to keep track of the number of words in the vocabulary table.
VDATA	An array used to hold vocabulary data.
VTABLE	A variable that points to the next free position in VDATA.
SYNONYM	Cause the word entered next into the vocabulary table to be a synonym of the word previously entered in the vocabulary table.
VERB–FOUND	Word number of the verb found by the ATN.
NOUN–FOUND	Word number of the noun found by the ATN.
DET–FOUND	Word number of the determinant found by the ATN.
ADJ–FOUND	Word number of the adjective found by the ATN.
ADVERB–FOUND	Word number of the adverb found by the ATN.

!VOCABULARY	Store a word in the vocabulary table.
W#-LIST	An array of word numbers in the input sentence.
WPS-LIST	An array of the parts of speech of the words in the input sentence.
S-V-T	Search the vocabulary table.
POINTER	A pointer to the array's W#-LIST and WPS-LIST.
ADVANCE	Advance POINTER to the next word in the input sentence.
P.O.S.	Find the part of speech of the current word for the ATN.
#-OF-W	Find the word number of the current word for the ATN.
?VERB	Is the current word a verb?
?NOUN	Is the current word a noun?
?DET	Is the current word a determinant?
?ADJ	Is the current word an adjective?
?ADVERB	Is the current word an adverb?
FAKE-ADJ	A fake vocabulary entry for a numeric adjective.
#ADJ	The value of a numeric adjective.
#?	Is the string on the stack a number?
0-FOUNDS	Zero all the registers used by the ATN.
/INIT/	Initialize the arrays used by INPUT$LOOKUP.
INPUT$ LOOKUP	Input a sentence and look up the words in it.
GET-GOOD-INPUT	Get a good sentence from the keyboard.
(N2)	Node N2 of the ATN.
(N1)	Node N1 of the ATN.
(NP)	Node NP of the ATN.
(3)	Node 3 of the ATN.
(2)	Node 2 of the ATN.
(1)	Node 1 of the ATN.
(S)	Node S of the ATN.
PARSE	Attempt to apply the ATN to an input sentence.
XPOS	The × position of the robot.
YPOS	The y position of the robot.
FACING	The compass facing of the robot.
MOVE	Move the robot.
TURN	Turn the robot.
MOVE-FORWARD?	Examine the ATN registers for MOVE FORWARD.
MOVE-BACK?	Examine the ATN registers for MOVE BACK.

TURN-LEFT?	Examine the ATN registers for TURN LEFT.
TURN-RIGHT?	Examine the ATN registers for TURN RIGHT.
BACKUP?	Examine the ATN registers for BACKUP.
ROBOT-POS	Print out the robot's position.
ROBOT-FACING	Print out the robot's facing.
HANDLE-INPUT	Attempt to make sense of an input sentence.
RUN-ROBOT	Demonstrate natural language parsing.

Computers can be very difficult for people to use. For the average person, direct use of a computer is impossible. As we have seen in this book, to use a computer effectively we must learn to speak one of its languages. Forth is one such language. However, in order for more people to be able to make use of computers, people with no specialized training, computers will have to learn how to understand human, or natural language. This chapter will present a set of words and techniques that make this possible, at least in part.

The scope of programming a computer to comprehend the entire English language is obviously too large for this chapter, so we'll limit our attempt to a small subset of English. Our small subset will be commands for directing the movement of a robot. We'll be able to understand sentences like this:

```
GO FORWARD 10 FEET.
BACKUP.
TURN LEFT.
MOVE 4 FEET BACKWARDS.
```

The techniques presented here are the same that would be used if we we're going to extend our program to understand a larger part of English. This is probably one of the most interesting chapters in the book to extend.

The problem of programming a computer to understand English can be divided into two parts, syntax and meaning. The method we will use to dissect the syntax of the sentences presented to us is known as "Augmented Transition Networks" or ATNs. They look somewhat like the sentence diagrams one learns in grammar school. ATNs are directed graphs whose arcs can involve other graphs and/or processing. Figure 10-1 is a diagram of the particular ATN we will be using.

As the words in the sentence are examined we move along the nodes and arcs of the ATN. Nodes (the circles) that are marked with a small f are those that processing can validly terminate. They are known as terminal nodes. As the graph is traversed, we can fill in various variables with information, such as what verb was just found. In the parlance of ATN's these variables are known as registers.

When the graph has been traversed successfully we have a syntactically valid sentence. This doesn't necessarily mean we have one that makes sense.

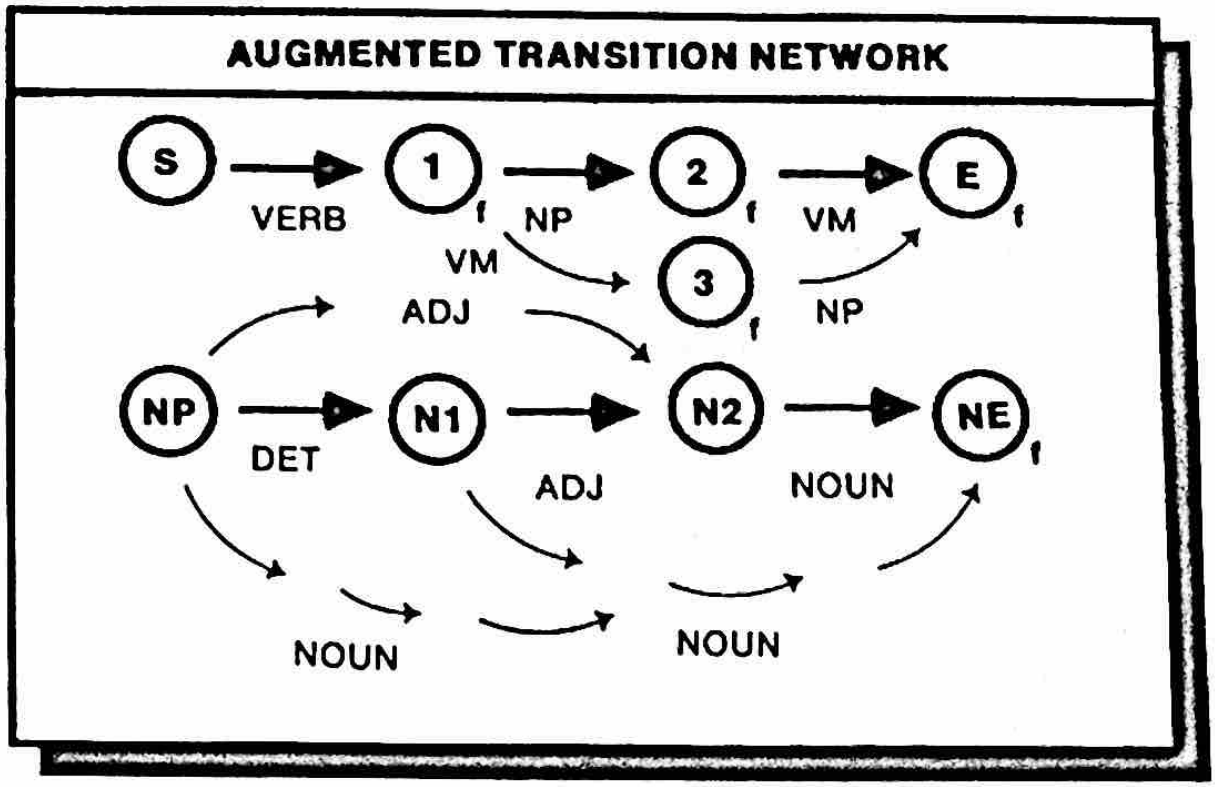

Figure 10-1

Here are some of the sentences that would be considered syntactically valid by our ATN:

DROP THE FEET FORWARD
BACKUP 10 BALLS

Obviously, these won't mean anything to our robot. The second part of our natural language processing will examine the variables or registers filled in by the ATN and try to make sense of them. There is nothing fancy about this second part, it just looks for particular combinations of registers.

We will use a vocabulary table in our processing. Each entry in the table will have a part of speech, a word number, and the word text. We can use synonyms by assigning different words the same number.

Suggested Extensions: This chapter presents a great opportunity for experimentation and extension. The techniques used could be expanded on to give our robot some more intelligence, or could be developed for an entirely different problem domain.

VERB (– N)

A constant for the part-of–speech verb.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a verb.

Example of Use:

```
P.O.S. VERB =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a verb as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
1 CCONSTANT VERB
```

NOUN (– N)

A constant for the part-of-speech noun.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a noun.

Example of Use:

```
P.O.S. NOUN =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a noun as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
2 CCONSTANT NOUN
```

DET (– N)

A constant for the part-of-speech determinant.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a determinant.

Example of Use:

```
P.O.S. DET =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a determinant as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
3 CCONSTANT DET
```

ADJ (– N)

A constant for the part-of-speech adjective.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent an adjective.

Example of Use:

```
P.O.S. ADJ =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned an adjective as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
4 CCONSTANT ADJ
```

ADVERB (-- N)

A constant for the part-of-speech adverb.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent an adverb.

Example of Use:

```
P.O.S. ADVERB =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned an adverb as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
5 CCONSTANT ADVERB
```

CONJ. (-- N)

A constant for the part-of-speech conjunction.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a conjunction.

Example of Use:

```
P.O.S. CONJ. =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a conjunction as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
6 CCONSTANT CONJ.
```

WORD# (– A)

A variable used to keep track of the number of words in the vocabulary table.

Stack on Entry: Empty.

Stack on Exit: (A) The address of WORD#.

Example of Use:

```
... WORD# @ ...
```

This code would leave the current number of words in the vocabulary table on the stack.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
1 CVARIABLE WORD#
```

VDATA (– A)

An array used to hold vocabulary data.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VDATA.

Example of Use:

```
... 64 VDATA + ...
```

Each entry in the vocabulary table will be 32 bytes long. This piece of code would leave the address of the third entry in the vocabulary table on the stack.

Algorithm: Allocate 802 bytes and then erase them.

Suggested Extensions: None.

Definition:

```
0 VARIABLE VDATA 800 ALLOT
VDATA 802 ERASE
```

VTABLE (– A)

A variable which points to the next free position in VDATA.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VTABLE.

Example of Use:

```
... VTABLE @ VDATA + ...
```

This piece of code would leave the address of the next free entry in VDATA on the stack.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
VDATA VARIABLE VTABLE
```

SYNONYM (–)

Cause the word entered next into the vocabulary table to be a synonym of the word previously entered in the vocabulary table.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
VERB !VOCABULARY SCREAM
```

```
VERB SYNONYM !VOCABULARY YELL
```

This use of SYNONYM would cause SCREAM and YELL to be regarded as synonyms by our natural language processor.

Algorithm: Decrement the value held in WORD#. This will cause both word to have the same "word number" in the vocabulary table.

Suggested Extensions: None.

Definition:

```
: SYNONYM -1 WORD# C+! ;
```

VERB-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the verb found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VERB-FOUND.

Example of Use:

```
VERB-FOUND C@ 5 =
```

This code would leave a true flag on the stack if the verb found by the ATN was word number five.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE VERB-FOUND
```

NOUN-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the noun found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of NOUN-FOUND.

Example of Use:

```
NOUN-FOUND C@ 2 =
```

This code would leave a true flag on the stack if the noun found by the ATN was word number two.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE NOUN-FOUND
```

DET-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the determinant found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of DET-FOUND.

Example of Use:

```
DET-FOUND C@ 1 =
```

This code would leave a true flag on the stack if the determinant found by the ATN was word number one.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE DET-FOUND
```

ADJ-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the adjective found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of ADJ–FOUND.

Example of Use:

```
ADJ–FOUND C@ 4 =
```

This code would leave a true flag on the stack if the adjective found by the ATN was word number four.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE ADJ–FOUND
```

ADVERB–FOUND (– A)

A byte variable used as a register by the ATN. It will hold the word number of the adverb found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of ADVERB–FOUND.

Example of Use:

```
ADVERB–FOUND C@ 4 =
```

This code would leave a true flag on the stack if the adjective adverb by the ATN was word number four.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE ADVERB–FOUND
```

!VOCABULARY (N –)

Place a word in the vocabulary table. The word is next in the input stream.

Stack on Entry: (N) The part of speech this word will be.

Stack on Exit: Empty.

Example of Use:

```
NOUN !VOCABULARY FIRETRUCK
```

This would place the noun FIRETRUCK in our vocabulary list.

Algorithm: The entries in the vocabulary table look like this:

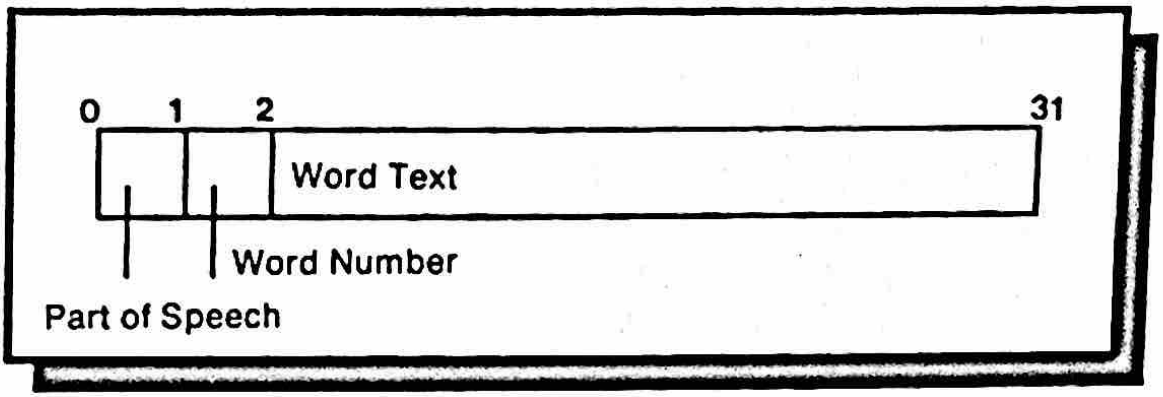

Figure 10-2

VTABLE points to the current free entry in the vocabulary table. The part of speech is stored in the first byte. WORD# is then stored in the second byte. The actual vocabulary word is then taken from the input stream using a blank as the delimiter. The word is then stored in the vocabulary table. WORD# is then incremented and 32 is added to VTABLE to point to the next entry.

Suggested Extensions: None.

Definition:

```
: !VOCABULARY VTABLE @ C!
  WORD# C@ VTABLE @ 1+ C!
```

```
  BL WORD VTABLE @ 2+ 30 CMOVE
  32 VTABLE +! 1 WORD# C+! ;

( Now store the words we'll use)
VERB !VOCABULARY GO ( 1)
VERB SYNONYM !VOCABULARY MOVE ( 1)
VERB !VOCABULARY TURN ( 2)
VERB !VOCABULARY BACKUP ( 3)
VERB !VOCABULARY PICK ( 4)
VERB !VOCABULARY DROP ( 5)
VERB !VOCABULARY KICK ( 6)
ADVERB !VOCABULARY FORWARD ( 7)
ADVERB !VOCABULARY BACK ( 8)
ADVERB SYNONYM !VOCABULARY BACKWARDS ( 8)
ADVERB !VOCABULARY LEFT ( 9)
ADVERB !VOCABULARY RIGHT ( 10)
DET !VOCABULARY THE ( 11)
NOUN !VOCABULARY FEET ( 12)
NOUN !VOCABULARY BALL ( 13)
ADJ !VOCABULARY RED ( 14)
ADJ !VOCABULARY BLUE ( 15) ->
CONJ. !VOCABULARY AND ( 16)
CONJ. SYNONYM !VOCABULARY THEN ( 16)
```

W#-LIST (- A)

An array used to hold the word numbers of the input sentence.

Stack on Entry: Empty.

Stack on Exit: (A) The address of W#-LIST.

Example of Use:

```
4 W#-LIST + C@
```

This code would leave word number of the fifth word in the input sentence.

Algorithm: Allocate 32 bytes for the array.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE W#-LIST 31 ALLOT
```

WPS-LIST (- A)

An array used to hold the part of speech of each word in the input sentence.

Stack on Entry: Empty.

Stack on Exit: (A) The address of WPS-LIST.

Example of Use:

```
4 W#-LIST + C@
```

This code would leave the part of speech of the fifth word in the input sentence.

Algorithm: Allocate 32 bytes for the array.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE WPS-LIST 31 ALLOT
```

S-V-T (A1 - (A2) F)

Search the vocabulary table for the string at the address on the stack.

Stack on Entry: (A1) – The address of the string to search for.

Stack on Exit: (A2) – The string's entry in the vocabulary table if found.
(F) – A Boolean flag, true if the string was found.

Example of Use:

```
: TEST QUERY >IN 0SET BL WORD S-V-T ;
```

This word would search for an input word in the vocabulary table.

Algorithm: Loop through the entire vocabulary list. Exit when a match is found or the end is reached. A match is found by first checking the lengths of the strings being compared. If they are unequal, skip to the next compare. The next compare uses -TEXT to actually compare the strings. If this compare is true, a match has been found.

Suggested Extensions: None.

Definition:

```
: S-V-T
  VDATA BEGIN
    OVER C@ OVER 2+ C@ = IF
      OVER 1+ OVER 2+ DUP C@ SWAP 1+
      -TEXT 0= IF
        SWAP DROP -1 EXIT
      ENDIF
    ENDIF
    32 + DUP C@ 0=
  UNTIL 2DROP 0 ;
```

POINTER (- A)

A variable used by the ATN to point to the current word being examined. Points to values in W#-LIST and WPS-LIST.

Stack on Entry: Empty.

Stack on Exit: (A) The address of POINTER.

Example of Use:

```
POINTER W#-LIST + C@
```

This code would leave the part of speech of the current word in the input sentence.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE POINTER
```

ADVANCE (–)

Increment POINTER to point to the next word in the input sentence.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Increment POINTER using C+!

Suggested Extensions: None.

Definition:

```
: ADVANCE 1 POINTER C+! ;
```

P.O.S. (– N)

Find the part of speech of the current word for the ATN.

Stack on Entry: Empty.

Stack on Exit: (N) – The part of speech.

Example of Use:

```
P.O.S. VERB =
```

This code would leave a true flag on the stack if the current word the ATN is looking at is a verb.

Algorithm: Access the array WPS–LIST using POINTER.

Suggested Extensions: None.

Definition:

```
: P.O.S.
   POINTER C@ WPS–LIST + C@ ;
```

#-OF-W (- N)

Find the number of the current word the ATN is processing.

Stack on Entry: Empty.

Stack on Exit: (N) – The word number.

Example of Use:

```
#-OF-W 7 =
```

This code would leave a true flag on the stack if the number of the current word was seven.

Algorithm: Access the array W#-LIST using POINTER.

Suggested Extensions: None.

Definition:

```
: #-OF-W
   POINTER C@ W#-LIST + C@ ;
```

?VERB (- F)

Is the current word a verb?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a verb.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is verb, then store the word number of the current word in the variable VERB-FOUND.

Suggested Extensions: None.

Definition:

```
: ?VERB
   P.O.S. VERB = DUP IF
      #-OF-W VERB-FOUND C!
   ENDIF ;
```