?NOUN (– F)

Is the current word a noun?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a noun.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is noun, then store the word number of the current word in the variable NOUN–FOUND.

Suggested Extensions: None.

Definition:

```
: ?NOUN
   P.O.S. NOUN = DUP IF
      #-OF-W NOUN-FOUND C!
   ENDIF ;
```

?DET (– F)

Is the current word a determinant?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a determinant.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is a determinant, then store the word number of the current word in the variable DET–FOUND.

Suggested Extensions: None.

Definition:

```
: ?DET
   P.O.S. DET = DUP IF
      #-OF-W DET-FOUND C!
   ENDIF ;
```

?ADJ (– F)

Is the current word an adjective?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is an adjective.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is an adjective, then store the word number of the current word in the variable ADJ–FOUND.

Suggested Extensions: None.

Definition:

```
: ?ADJ
   P.O.S. ADJ = DUP IF
     #-OF-W ADJ-FOUND C!
   ENDIF ;
```

?ADVERB (– F)

Is the current word an adverb?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is an adverb.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is an adverb, then store the word number of the current word in the variable ADVERB–FOUND.

Suggested Extensions: None.

Definition:

```
: ?ADVERB
   P.O.S. ADVERB = DUP IF
     #-OF-W ADVERB-FOUND C!
   ENDIF ;
```

FAKE-ADJ (- A)

Leave the address of a fake vocabulary table entry for adjective number 255.

Stack on Entry: Empty.

Stack on Exit: (A) - The address of the fake entry.

Example of Use: See words defined below.

Algorithm: This entry will be used for handling numbers in the input sentence.

Suggested Extensions: None.

Definition:

```
4 CVARIABLE FAKE-ADJ 255 C,
```

#ADJ (- A)

A variable which will hold a number found in the input sentence.

Stack on Entry: Empty.

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

Example of Use:

```
#ADJ ?
```

This would print the number found in the input sentence. Since it would be extremely inefficient to store all the possible numbers in the vocabulary table, we will use this variable and return an adjective number of 255 to signify that a number was found in the input sentence.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE #ADJ
```

#? (A1 – (A2) F)

Is the string on the stack a number?

Stack on Entry: (A1) – The string to check.

Stack on Exit: (A2) – The address of the vocabulary table entry for a numeric adjective.
(F) – A Boolean flag, true if the string on the stack could be converted to a number.

Example of Use: See words defined below.

Algorithm: Use >BINARY to try to convert the string to a number. If it was successful, store the number in #ADJ and leave the FAKE–ADJ entry on the stack.

Suggested Extensions: None.

Definition:

```
: #?
  0. ROT >BINARY C@ DUP BL = SWAP 0=
  OR IF
    DROP #ADJ ! FAKE-ADJ -1
  ELSE
    2DROP 0
  ENDIF ;
```

0–FOUNDS (–)

Zero all the registers used by the ATN.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use C0SET on all the byte variables.

Suggested Extensions: None.

Definition:

```
: 0-FOUNDS
  VERB-FOUND C0SET NOUN-FOUND C0SET
  DET-FOUND C0SET ADJ-FOUND C0SET
  ADVERB-FOUND C0SET ;
```

/INIT/ (-)

Init the arrays for INPUT$LOOKUP.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Erase all 32 bytes of WPS-LIST and W#-LIST.

Suggested Extensions: None.

Definition:

```
: /INIT/
   W#-LIST 32 ERASE WPS-LIST 32 ERASE ;
```

INPUT$LOOKUP (- F)

Input a line of text from the keyboard and attempt to look up all the words in the sentence.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if all the words in the sentence could be found in the vocabulary table.

Example of Use: See words defined below.

Algorithm: Use /INIT/ to erase W#-LIST and WPS-LIST. Input a string from the keyboard and loop on the input words until there are no more. For each word, first check to see if it can be converted to a number. If it cannot be converted to a number, try to look it up in the vocabulary table. If it cannot be found in the vocabulary table, exit the loop. If a word was found, store the word number and part of speech in W#-LIST and WPS-LIST.

Suggested Extensions: None.

Definition:

```
: INPUT$LOOKUP
   /INIT/ QUERY >IN 0SET 0 BEGIN
      BL WORD DUP C@
   WHILE
      DUP #? NOT IF
         DUP S-V-T NOT IF
            TYPE2 ." is not in the vocabulary list.
            " DROP 0 EXIT
      ENDIF ENDIF
      SWAP DROP DUP C@ 3 PICK WPS-LIST + C!
      1+ C@ OVER W#-LIST + C! 1+
   REPEAT 2DROP -1 ;
```

GET-GOOD-INPUT (-)

Get a good sentence from the keyboard.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Loop calling INPUT$LOOKUP until it finds a good sentence.

Suggested Extensions: None.

Definition:

```
: GET-GOOD-INPUT BEGIN INPUT$LOOKUP UNTIL ;
```

(N2) (- F)

Node N2 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) - A Boolean flag, true if N2 found a transition to another node.

Example of Use: See words defined below.

Algorithm: Node N2 needs to find a noun to be successful. If N2 finds a noun, it should advance past the noun in the input list. NE is a node that does nothing and is always successful so for efficiency, is is not actually coded.

Suggested Extensions: None.

Definition:

```
: (N2) ." N2 " ?NOUN DUP IF
     ADVANCE
   ENDIF ;
```

(N1) (– F)

Node N1 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if N1 found a transition to another node.

Example of Use: See words defined below.

Algorithm: Node N1 can move to N2 if an adjective is found or to NE if a noun is found. NE is a node that does nothing and is always successful so for efficiency, is is not actually coded.

Suggested Extensions: None.

Definition:

```
: (N1) ." N1 " ?ADJ IF
     ADVANCE (N2)
   ELSE
     ?NOUN DUP IF
       ADVANCE
     ENDIF
   ENDIF ;
```

(NP) (– F)

Node NP of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if NP found a transition to another node.

Example of Use: See words defined below.

Algorithm: NP can advance if it finds a determinant (to N1), an adjective (to N2), or a noun (to NE).

Suggested Extensions: None.

Definition:

```
:(NP) ." NP " ?DET IF
      ADVANCE (N1)
   ELSE
      ?ADJ IF
         ADVANCE (N2)
      ELSE
         ?NOUN DUP IF
            ADVANCE
   ENDIF ENDIF ENDIF ;
```

(3) (– F)

Node 3 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 3 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 3 can advance if an NP is found. But it is also successful if it cannot advance, because it is a terminal node. Note that when an arc is another graph, as NP is, no advance is done to the next node.

Suggested Extensions: None.

Definition:

```
: (3) ." 3 " (NP) DROP -1 ;
```

(2) (– F)

Node 2 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 2 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 2 can advance if an adverb is found. But it is also successful if it cannot advance because it is a terminal node.

Suggested Extensions: None.

Definition:

```
: (2) ." 2 " ?ADVERB IF
      ADVANCE
   ENDIF -1 ;
```

(1) (– F)

Node 1 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 1 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 1 can advance if a noun phrase or an adverb is found. But it is also successful if it cannot advance because it is a terminal node.

Suggested Extensions: None.

Definition:

```
: (1) ." 1 " ?ADVERB IF
      ADVANCE (3)
   ELSE
      (NP) IF
         (2)
      ELSE
      -1
      ENDIF
   ENDIF ;
```

(S) (– F)

Node S of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if S found a transition to another node.

Example of Use: See words defined below.

Algorithm: S can advance if a verb is found.

Suggested Extensions: None.

Definition:

```
: (S) ." S " ?VERB IF
      ADVANCE (1)
  ENDIF ;
```

PARSE (– F)

Attempt to apply the ATN to an input sentence.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the sentence was parsed correctly.

Example of Use: See words defined below.

Algorithm: Zero all the ATN registers, zero POINTER, and start the ATN. If the ATN completes successfully and the sentence is complete, or POSITION points to a conjuction, the parse is successful.

Suggested Extensions: None.

Definition:

```
: PARSE POINTER C0SET
  0-FOUNDS (S)
  P.O.S. DUP 0= SWAP 6. = OR
  AND ;
```

XPOS (– A)

A variable that will hold our robot's X coordinate position.

Stack on Entry: Empty.

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

Example of Use:

```
XPOS ?
```

This would print the X coordinate location of our robot.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE XPOS
```

YPOS (– A)

A variable that will hold our robot's Y coordinate position.

Stack on Entry: Empty.

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

Example of Use:

```
YPOS ?
```

This would print the Y coordinate location of our robot.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE YPOS
```

FACING (- A)

A byte variable that will hold the direction our robot is facing.

Stack on Entry: Empty.

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

Example of Use:

```
FACING C?
```

This would print the direction our robot is facing (0–North / 1–East / 2–South / 3–West).

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE FACING
```

MOVE (N -)

Move the robot N feet in its current direction. (This word uses the case words from Chapter 2.)

Stack on Entry: (N) – How many feet to move the robot.

Stack on Exit: Empty.

Example of Use:

```
10 MOVE
```

If the robot was facing north, this would increment its Y coordinate by 10.

Algorithm: Use a case statement to add the value to the proper variable.

Suggested Extensions: None.

Definition:

```
: MOVE FACING C@ CASE
    0 =OF YPOS +! END-OF
    1 =OF XPOS +! END-OF
    2 =OF NEGATE YPOS +! END-OF
    3 =OF NEGATE XPOS +! END-OF
  ENDCASE ;
```

TURN (N –)

Turn the robot one compass direction. (This word uses the case words from Chapter 2.)

Stack on Entry: (N) – Positive one if the robot is turning to its right; negative one if it is turning to its left.

Stack on Exit: Empty.

Example of Use:

```
1 TURN
```

If the robot was facing north before the above code was executed, it would be facing east following its execution.

Algorithm: Add the value on the stack to the variable FACING. Check to see if it has overflowed the range 0–3 and adjust it accordingly.

Suggested Extensions: None.

Definition:

```
: TURN
  FACING C+! FACING C@ CASE
    4 =OF 0 FACING C! END-OF
    255 =OF FACING C0SET END-OF
  ENDCASE ;
```

MOVE-FORWARD? (– F)

Move the robot forward if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved forward.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase MOVE FORWARD. If they are found, check the adjective and noun to see if they specify how many feet to move forward. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: MOVE-FORWARD?
   NOUN-FOUND C@ DUP 0=
   SWAP 12 = OR
   VERB-FOUND C@ 1 = AND
   ADVERB-FOUND C@ 7 = AND IF
      ADJ-FOUND C@ DUP 255 <>
      SWAP 0 <> AND IF
         0 EXIT
      ENDIF
      ADJ-FOUND C@ IF
         #ADJ @
      ELSE
         1
      ENDIF
      MOVE -1 EXIT
   ENDIF 0 ;
```

MOVE-BACK? (– F)

Move the robot backwards if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved backwards.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase MOVE BACKWARDS. If they are found, check the adjective and noun to see if they specify how many feet to move backwards. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: MOVE-BACK?
   NOUN-FOUND C@ DUP 0=
   SWAP 12 = OR
   VERB-FOUND C@ 1 = AND
   ADVERB-FOUND C@ 8 = AND IF
      ADJ-FOUND C@ DUP 255 <>
      SWAP 0 <> AND IF
         0 EXIT
      ENDIF
      ADJ-FOUND C@ IF
         #ADJ @
      ELSE
         1
      ENDIF
      NEGATE MOVE -1 EXIT
   ENDIF 0 ;
```

TURN-LEFT? (- F)

Turn the robot to the left if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was turned left.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase TURN LEFT. If they are found, turn the robot to the left with TURN.

Suggested Extensions: None.

Definition:

```
: TURN-LEFT?
   NOUN-FOUND C@ 0=
   2VERB-FOUND C@ 2 = AND
   ADVERB-FOUND C@ 9 = AND IF
      -1 TURN -1
   ELSE
      0
   ENDIF ;
```

TURN-RIGHT? (- F)

Turn the robot to the right if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was turned right.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase TURN RIGHT. If they are found, turn the robot to the right with TURN.

Suggested Extensions: None.

Definition:

```
: TURN-RIGHT?
  NOUN-FOUND C@ 0=
  VERB-FOUND C@ 2 = AND
  ADVERB-FOUND C@ 10 = AND IF
    1 TURN -1
  ELSE
    0
  ENDIF ;
```

BACKUP? (- F)

Back up the robot if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved backwards.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb BACKUP. If it is found, check the adjective and noun to see if they specify how many feet to move back. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: BACKUP?
   NOUN-FOUND C@ DUP 0=
   SWAP 12 = OR
   VERB-FOUND C@ 3 = AND IF
   ADJ-FOUND C@ DUP 255 <>
   SWAP 0 <> AND IF
      0 EXIT
   ENDIF
   ADJ-FOUND C@ IF
      #ADJ @
   ELSE
      1
   ENDIF
   NEGATE MOVE -1 EXIT
ENDIF 0 ;
```

ROBOT-POS (-)

Print the position of the robot.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
ROBOT-POS
```

Would print:

```
Robot Position:
X => 0
Y => 0
```

If the robot's X and Y position was zero.

Algorithm: Print out the values of XPOS and YPOS.

Suggested Extensions: None.

Definition:

```
: ROBOT-POS
   CR ." Robot Position: " CR
   ." X => " XPOS ? CR
   ." Y => " YPOS ? ;
```

ROBOT-FACING (-)

Print the facing direction of the robot. (This word uses the case words from Chapter 2.)

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

ROBOT-FACING
Would print:
"Facing North."

If the robot was facing north.

Algorithm: Branch on the value held in facing.

Suggested Extensions: None.

Definition:

```
: ROBOT-FACING CR." Facing"
   FACING C@ CASE
      0 =OF ." North." END-OF
      1 =OF ." East." END-OF
      2 =OF ." South." END-OF
      3 =OF ." West." END-OF
   ENDCASE;
```

HANDLE-INPUT (-)

Attempt to make sense of an input sentence.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Call each of the words defined previously that attempt to process input. If any of them complete successfully, the word will exit. If none are able to deal with the input, print out an error message.

Suggested Extensions: To extend how much our robot understands, add more processing phrases to this word.

Definition:

```
: HANDLE-INPUT
   MOVE-FORWARD? IF EXIT ENDIF
   MOVE-BACK? IF EXIT ENDIF
   TURN-LEFT? IF EXIT ENDIF
   TURN-RIGHT? IF EXIT ENDIF
   BACKUP? IF EXIT ENDIF
   CR ." Sorry, I don't understand you."
   CR ;
```

RUN-ROBOT (-)

Demonstrate natural language processing.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: None.

Algorithm: Print out the position and facing direction of the robot. If the input pointer does not point to a conjunction, get more input. Attempt to parse the input. If the parse is unsuccessful print an error message; otherwise, allow our input handler to try to make sense of it.

Suggested Extensions: None.

Definition:

```
: RUN-ROBOT 3 0 DO
     ROBOT-POS ROBOT-FACING
     P.O.S. CONJ. = IF
        ADVANCE
     ELSE
        CR GET-GOOD-INPUT
     ENDIF
     PARSE IF
        HANDLE-INPUT
     ELSE
        CR ." That wasn't very well said."
     ENDIF
  LOOP ;
```

11

Data Structures

Words Defined in This Chapter:

1ARRAY	Define a one-dimensional cell array.
1CARRAY	Define a one-dimensional byte array.
1ARRAY–RNG	Define a one-dimensional cell array, with specified lower and upper bounds.
1CARRAY–RNG	Define a one-dimensional byte array, with specified lower and upper bounds.
OBM	Print the array out-of-bounds error message.
1ARRAY–BC	Define a one-dimensional cell array with bounds checking.
1CARRAY–BC	Define a one-dimensional byte array with bounds checking.
1ARRAY–RNG–BC	Define a one-dimensional cell array, with specified lower and upper bounds. Include bounds checking.
1CARRAY–RNG–BC	Define a one-dimensional byte array, with specified lower and upper bounds. Include bounds checking.
2ARRAY	Define a two-dimensional cell array.
2CARRAY	Define a two-dimensional byte array.

2ARRAY-RNG	Define a two-dimensional cell array, with specified lower and upper bounds.
2CARRAY-RNG	Define a two-dimensional byte array, with specified lower and upper bounds.
2ARRAY-BC	Define a two-dimensional cell array with bounds checking.
2CARRAY-BC	Define a two-dimensional byte array with bounds checking.
2ARRAY-RNG-BC	Define a two-dimensional cell array, with specified lower and upper bounds. Include bounds checking.
2CARRAY-RNG-BC	Define a two-dimensional byte array, with specified lower and upper bounds. Include bounds checking.
1?ARRAY	Define a one-dimensional array with a specified element size.
1?ARRAY-RNG	Define a one-dimensional array with a specified element size. The lower and upper bounds will also be specified.
1?ARRAY-BC	Define a one-dimensional array with a specified element size. Include bounds checking.
1?ARRAY-RNG-BC	Define a one-dimensional array with a specified element size. The lower and upper bounds will also be specified. Include bounds checking.
ARRAY	Define an array of any dimensionality with a specified element size.
DISK-2-MEM	Move data from disk to memory.
MEM-2-DISK	Move data from memory to disk.
DISK-ARRAY	Define a disk array of any dimensionality with a specified element size.
RECORD	Start a record definition.
FIELD	Define a field in a record.
VARIANT	Start a variant in a record definition.
START-VARIANTS	Start a set of variants in a record definition.
END-VARIANT	End the definition of a variant portion of a record.
END-ALL-VARIANTS	Complete the definition of a set of variants in a record.
INSERT	Insert a record as a sub-record in a record being defined.
END-RECORD	Complete the definition of a record.

1AFIELD	Define an array field in a record.
INSTANCE	Create an instance of a record.
M–INSTANCE	Create an array of instances of a record.
TO	Cause a "to" type variable to perform a store.
TVARIABLE	Define a cell sized "to" type variable.
TCVARIABLE	Define a byte sized "to" type variable.
FIFO	Create a queue.
(#FI)	Fetch the start and end pointers of a queue.
?FIFO	Determine if data are stored in a queue.
CNO	Abort with an error message if a queue is empty.
@FIFO	Remove data from a queue.
!FIFO	Store data in a queue.
FIFO–RESET	Clear a queue.

This chapter presents a set of words for managing data, both in memory and on disk. Arrays of every type and size are presented, as is a record structure similar to Pascal's. We also throw in a new kind of variable, "to" type variables, and some words for storing and manipulating queues.

RECORD STRUCTURES

The record structure presented in this chapter will enable us to define records with multiple fields, variant fields, and subrecords. Here is an example of a definition of a simple record:

```
RECORD PERSON
  16 FIELD NAME
   8 FIELD SS-#
END-RECORD
```

PERSON is now the name of a record with two fields. NAME is a field in PERSON that is 16 bytes in length. SS-# is an 8-byte field. RECORD defines a template. To create an actual instance of a record, we use the word INSTANCE. It works like this:

```
INSTANCE JAMES OF PERSON
```

An instance of the record type PERSON called JAMES is created. Now the fields of the record JAMES can be accessed like this:

```
JAMES NAME

JAMES SS-#
```

If we want to create a one-dimensional array of instances of a record, we use the word M-INSTANCE like this:

```
10 M-INSTANCE COMPANY OF PERSON
```

This creates a set of 10 instances of the record PERSON, numbered zero to nine. Here is how we would print out the name of the seventh person:

```
7 COMPANY NAME 16 TYPE
```

Records can have variants; these are identical portions of the record mapped in different ways. This saves space by not wasting any bytes. Our record could be expanded like this:

```
RECORD PERSON
   16 FIELD NAME
   8 FIELD SS-#
   1 FIELD SEX
   START-VARIANTS
      VARIANT
         16 FIELD WIFE-NAME
      END-VARIANT
      VARIANT
         16 FIELD MAIDEN-NAME
         2 FIELD #CHILDREN
         1 FIELD PREG?
      END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

We save 16 bytes by using variant records in the above example. Any number of variants can exist in our records. There can also be multiple sets of variants in a single record.

At times we may wish to include certain records as part of another record. We can accomplish this with the following code:

```
RECORD CAR
   2 FIELD MAKE
   2 FIELD MODEL
   INSERT OWNER OF PERSON
   6 FIELD PLATE
END-RECORD
```

If ZAQ is an instance of the record CAR, this would print the name of its owner:

```
ZAQ OWNER NAME 16 TYPE
```

Records can be nested to any depth using this technique.

DISK ADDRESS

Words to move data between disk and memory are presented in this chapter. They make use of a double-length number called a *disk address.* This representation just assigns each byte in mass storage a unique number. The bytes on block zero are numbered 0-1023, the bytes on block one 1024-2047, etc. DISK-2-MEM and MEM-2-DISK enable us to deal with disk storage without being concerned with block boundaries.

"To" Type Variables

"To" type variables are an attempt to get rid of some of the strange syntax that store and fetch introduce into Forth. They remove some flexibility in trade for some clarity. The replacement is shown in the chart that follows. "To" type variables don't allow operations like +!, and OSET, so they are somewhat less powerful than normal variables.

Suggested Extensions: If you like "to" type variables, most of the data structure words in this chapter could be modified to use the "to" technique. Only one-dimensional arrays of records are presented. The instance words could be expanded just like the array words.

Operation	Normal Variables	"To" Type Variables
Define	0 VARIABLE STUFF	0 TVARIABLE STUFF
Store	99 STUFF !	99 TO STUFF
Fetch	STUFF @	STUFF

1ARRAY (N –) (N – A)

Define and allocate space for a one-dimensional cell array.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
10 1ARRAY HI-TMT
```

This would define a 10-cell array, called HI-TMT. This would print the value of the second element of HI-TMT:

```
1 HI-TMT ?
```

Note that the array elements are numbered 0-9, so 1 is the second element of the array.

Algorithm: At compile time, allocate space for the array, two bytes for each element. At run time, multiply the element number by two and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1ARRAY <BUILDS
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    SWAP 2* + ;
```

1CARRAY (N -) (N - A)

Define and allocate space for a one-dimensional byte array.

Stack on Entry: (Compile Time) (N) - Number of entries in the array.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
50 1CARRAY HI-MOM
```

This would define a 50 cell array, called HI-MOM. This word would sum all the elements of HI-MOM:

```
: SUM-HI-MOM 0 HI-MOM C@ 50 1 DO
    I HI-MOM C@ +
  LOOP ;
```

Note that the array elements are numbered 0-49.

Algorithm: At compile time, allocate space for the array, one byte for each element. At run time, add the element number to the base address.

Suggested Extensions: None.

Definition:

```
: 1CARRAY <BUILDS
   HERE SWAP DUP ALLOT ERASE
DOES>
   + ;
```

1ARRAY-RNG (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional cell array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) - Lower bound of the array.
(N2) - Upper bound of the array.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
1940 1986 1ARRAY-RNG ROYS/YEAR
```

This would define a 47-cell array, called ROYS/YEAR. The lower bound of the array is 1940, the upper bound is 1985. This would access the twenty-second element of the array:

```
1962 ROYS/YEAR
```

Algorithm: At compile time, store the lower bound. Allocate space for the array, two bytes for each element. At run time, subtract the lower bound from the passed element. Multiply by two and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1ARRAY-RNG <BUILDS
    SWAP DUP , - 1+
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP @ ROT SWAP - 2* 2+ + ;
```

1CARRAY-RNG (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional byte array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
1964 1986 1CARRAY-RNG SJT/TICKETS/YEAR
```

This would define a 23-cell array, called SJT/TICKETS/YEAR. The lower bound of the array is 1964; the upper bound is 1986. This would print out the twenty-second element of the array.

```
1985 SJT/TICKETS/YEAR C?
```

Algorithm: At compile time, store the lower bound. Allocate space for the array, one byte for each element. At run time, subtract the lower bound from the passed element and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1CARRAY-RNG <BUILDS
    SWAP DUP , - 1+
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ ROT SWAP - + 2+ ;
```

OBM (N -)

Print the array out-of-bounds error message.

Stack on Entry: (N) – The offending array element.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Print the message, print the offending element, and abort out.

Suggested Extensions: None.

Definition:

```
: OBM ." Array Out of Bounds ". ABORT;
```

1ARRAY-BC (N -) (N - A)

Define and allocate space for a one-dimensional cell array with bounds checking.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
5 1ARRAY-BC SU+KEV
```

This would define a 5-cell array, called SU+KEV. If this array is passed an element out of range, it will abort out, like this:

```
7 SU+KEV ?

Array Out of Bounds 7
```

Note that the array elements are numbered 0–4.

Algorithm: At compile time, allocate space for the array, two bytes for each element. Store the upper bound of the array. At run time, check the passed array element to see if it falls in the proper range. If the array element is in range, multiply the element number by two and add it to the base address plus two (to skip the upper bound).

Suggested Extensions: None.

Definition:

```
: 1ARRAY-BC <BUILDS
```

```
    DUP ,
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP 2+ >R @ OVER > OVER -1 > AND IF
      2* R> +
    ELSE
      OBM
    ENDIF ;
```

1CARRAY-BC (N -) (N - A)

Define and allocate space for a one-dimensional byte array with bounds checking.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
100 1CARRAY-BC DAVE []
```

This would define a 100-byte array, called DAVE []. If this array is passed an element out of range, it will abort out, like this:

```
307 DAVE [] C?

Array Out of Bounds 307
```

Note that the array elements are numbered 0–99.

Algorithm: At compile time, allocate space for the array, one byte for each element. Store the upper bound of the array. At run time, check the passed array element to see if it falls in the proper range. If the array element is in range, add it to the base address plus two (to skip the upper bound).

Suggested Extensions: None.

Definition:

```
: 1CARRAY-BC <BUILDS
    DUP ,
    HERE SWAP DUP ALLOT ERASE
```

```
DOES>
  DUP 2+ >R @ OVER > OVER -1 > AND IF
    R> +
  ELSE
    OBM
  ENDIF ;
```

1ARRAY-RNG-BC (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional cell array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
-10 10 1ARRAY-RNG-BC >DAN []
```

This would define a 21-cell array, called >DAN []. The lower bound of the array is -10, the upper bound is 10. This would print out the second element of the array:

```
-9 >DAN []?
```

Since bounds checking is also included in this array, the following code would abort with an array bounds error:

```
-15 >DAN []?

Array out of bounds -15
```

Algorithm: At compile time, allocate space for the array, two bytes for each element. Store the lower and upper bounds. At run time, see if the passed element falls in the range specified by the lower and upper bounds. If the element is within the proper range, subtract the lower bound from the passed element. Multiply by two and add it to the base address plus four (to skip the lower and upper bounds).

Suggested Extensions: None.

Definition:

```
: 11ARRAY-RNG-BC <BUILDS
    SWAP DUP , OVER , - 1+
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R OVER >R 2DUP @ >= >R
    2+ @ <= R> AND IF
      R> R> DUP @ ROT SWAP - 2* 4 + +
    ELSE
      R> OBM
    ENDIF ;
```

1CARRAY-RNG-BC (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional byte array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
1983 1986 1CARRAY-RNG-BC LISA-PER-YEAR()
```

This would define a 4-byte array, called LISA-PER-YEAR(). The lower bound of the array is 1983, the upper bound is 1986. This would print out the second element of the array:

```
1984 LISA-PER-YEAR() C?
```

Since bounds checking is also included in this array, the following code would abort with an array bounds error:

```
1982 LISA-PER-YEAR() C?

Array out of bounds 1982
```

Algorithm: At compile time, allocate space for the array, one byte for each element. Store the lower and upper bounds. At run time, see if the passed element falls in the range specified by the lower and upper bounds. If the element

is within the proper range, subtract the lower bound from the passed element and add it to the base address plus four (to skip the lower and upper bounds).

Suggested Extensions: None.

Definition:

```
: 1CARRAY-RNG-BC <BUILDS
    SWAP DUP , OVER , - 1+
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER >R 2DUP @ >= >R
    2+ @ <= R> AND IF
      R> R> DUP @ ROT SWAP - + 4 +
    ELSE
      R> OBM
    ENDIF ;
```

2ARRAY (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
5 10 2ARRAY D+ANN
```

This would define a 5-by-10 cell array, called D+ANN. This would print the value of the second row and third column of D+ANN:

```
1 2 D+ANN ?
```

Note that the rows are numbered 0–4, and the columns are numbered 0–9.

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements if found by multiplying the number of

rows by the number of columns. Store the size of the rows. At run time, multiply the row number by the row size and add the column number times two. Add this to the base address plus two (to skip the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY <BUILDS
    DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP @ 4 ROLL * + SWAP 2* + 2+ ;
```

2CARRAY (N1 N2 –) (N1 N2 – A)

Define and allocate space for a two-dimensional byte array.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
20 3 2CARRAY I+Y
```

This would define a 20-by-3 byte array, called I+Y. This would print the value of the seventh row and third column of I+Y:

```
6 2 I+Y C?
```

Note that the rows are numbered 0–19, and the columns are numbered 0–2.

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store ths size of the rows. At run time, multiply the row number by the row size and add the column number. Add this to the base address plus two (to skip the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY <BUILDS
    DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ 4 ROLL * + + 2+ ;
```

2ARRAY-RNG (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
1929 1939 1 365 STOCK-PRICES
```

This would define an 11-by-386 cell array, called STOCK-PRICES, which would print the value of the second row and seventy-third column of STOCK-PRICES:

```
1930 73 STOCK-PRICES ?
```

Note that the rows are numbered 1929–1939, and the columns are numbered 1–365.

Algorithm: At compile time, allocate space for the array; two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower bounds of the rows and columns. At run time, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number times two. Add this to the base address plus six (to skip the lower bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-RNG <BUILDS
    OVER , 4 PICK , SWAP - 1+ >R
    SWAP - 1+ R> DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R DUP LROT @ - LROT 2+ @ -
    SWAP R> DUP 4 + @ 4 ROLL * +
    SWAP 2* + 6 + ;
```

2CARRAY-RNG (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
1 8 1 8 2CARRAY-RNG CHESS-BOARD
```

This would define an 8-by-8 byte array, called CHESS-BOARD. This would print the value of the third row and third column of CHESS-BOARD:

```
3 3 CHESS-BOARD C?
```

Note that the rows are numbered 1–8, and the columns are numbered 1–8.

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store ths size of the rows. Store the lower

bounds of the rows and columns. At run time, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number. Add this to the base address plus six (to skip the lower bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-RNG <BUILDS
    OVER , 4 PICK , SWAP - 1+ >R
    SWAP - 1+ R> DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R DUP LROT @ - LROT 2+ @ -
    SWAP R> DUP 4 + @ 4 ROLL * +
    + 6 + ;
```

2ARRAY-BC (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with bounds checking.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
100 2 2ARRAY-BC GALLONS
```

This would define a 100-by-2 cell array, called GALLONS, which would print the value of the eighty-second row and first column of GALLONS:

```
81 0 GALLONS ?
```

Because this array has bounds checking, passing it a value out of range will result in an abort, like this:

```
8 22 GALLONS ?
```

Array out of bounds 22

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the upper bounds of the row and columns. At run time, determine if the passed elements lie within the valid range specified by the row and column sizes. If they do, multiply the row number by the row size and add the column number times two. Add this to the base address plus six (to skip the row and column bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-BC <BUILDS
    2DUP , , DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R @ OVER > OVER -1 > AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R 2+ @ OVER >
      OVER -1 > AND NOT IF
        OBM
      ELSE
        SWAP R> DUP 4 + @ 4 ROLL * +
        SWAP 2* + 6 +
    ENDIF ENDIF ;
```

2CARRAY-BC (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with bounds checking.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
3 3 2CARRAY-BC TTT-BOARD
```

This would define a 3-by-3 byte array, called TTT-BOARD. This would print the value of the first row and first column of TTT-BOARD:

```
0 0 TTT-BOARD C?
```

Because this array has bounds checking, passing it a value out of range will result in an abort, like this:

```
0 3 TTT-BOARD C?
Array out of bounds 3
```

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the upper bounds of the row and columns. At run time, determine if the passed elements lie within the valid range specified by the row and column sizes. If they do, multiply the row number by the row size and add the column number. Add this to the base address plus six (to skip the row and column bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-BC <BUILDS
    2DUP , , DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R @ OVER > OVER -1 > AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R 2+ @ OVER >
       OVER -1 > AND NOT IF
         OBM
       ELSE
         SWAP R> DUP 4 + @ 4 ROLL * +
         + 6 +
       ENDIF
     ENDIF ;
```

2ARRAY-RNG-BC (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
1 9 0 2 2ARRAY-RNG-BC RUNS
```

This would define a 9-by-3 cell array, called RUNS. This would print the value of the ninth row and third column of RUNS:

```
9 2 RUNS ?
```

Because this array has bounds checking, it will abort when it is passed an element not within its defined bounds. For example:

```
5 -1 RUNS ?

Array out of bounds -1
```

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower and upper bounds of the rows and columns. At run time, determine if the passed row and column numbers lie within the range defined at compile time. If they do, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number times two. Add this to the base address plus ten (to skip the lower and upper bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-RNG-BC <BUILDS
    5 1 DO I PICK , LOOP SWAP
    - 1+ >R SWAP - 1+ R> DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ >=
```

```
>R @ <= R> AND NOT IF
  OBM
ELSE
  SWAP R> DUP >R OVER SWAP 2DUP 6 +
  @ >= >R 4 + @ <= R> AND NOT IF
    OBM
  ELSE
    SWAP R> DUP >R 2+ @ -
    SWAP R> DUP >R 6 + @ -
    SWAP R> DUP 8 + @ 4 ROLL * +
    SWAP 2* + 10 +
  ENDIF
ENDIF ;
```

2CARRAY-RNG-BC (N1 N2 N3 N4 -)

(N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

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

Example of Use:

```
75 80 -10 0 2CARRAY-RNG-BC #BROKEN
```

This would define a 6-by-11 byte array, called #BROKEN, which would print the value of the sixth row and eleventh column of #BROKEN:

```
80 0 #BROKEN C?
```

Because this array has bounds checking, it will abort when it is passed an element not within its defined bounds. For example:

```
80 1 #BROKEN C?

Array out of bounds 1
```

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower and upper bounds of the rows and columns. At run time, determine if the passed row and column numbers lie within the range defined at compile time. If they do, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number. Add this to the base address plus ten (to skip the lower and upper bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-RNG-BC <BUILDS
    5 1 DO I PICK , LOOP SWAP
    - 1+ >R SWAP - 1+ R> DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ >=
    >R @ <= R> AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R OVER SWAP 2DUP 6 +
      @ >= >R 4 + @ <= R> AND NOT IF
        OBM
      ELSE
        SWAP R> DUP >R 2+ @ -
        SWAP R> DUP >R 6 + @ -
        SWAP R> DUP 8 + @ 4 ROLL * +
        + 10 +
      ENDIF
    ENDIF ;
```

1?ARRAY (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional array with a specified element size.

Stack on Entry: (Compile Time) (N1) - Number of entries in the array.
(N2) - Size, in bytes, of each array entry.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
20 4 1?ARRAY BIG#S
```

This would define a 20-element array, each element being four bytes in length, called BIG#S. The following would dump the four bytes that make up the second element of BIG#S:

```
1 BIG#S 4 DUMP
```

Note that the array elements are numbered 0–19.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. At run time, find the address of the element by multiplying the passed element by the element size. Add this to the base address plus two (to skip the element size).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY <BUILDS
    DUP ,*
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ ROT * + 2+ ;
```

1?ARRAY-BC (N1 N2 –) (N – A)

Define and allocate space for a one-dimensional array with a specified element size. Include array bounds checking.

Stack on Entry: (Compile Time) (N1) – Number of entries in the array.
(N2) – Size, in bytes, of each array entry.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
5 10 1?ARRAY TREALS []
```

This would define a 5-element array, each element being ten bytes in length, called TREALS []. The following would dump the ten bytes that make up the fifth element of TREALS []:

```
4 TREALS [] 10 DUMP
```

Array bounds checking will catch any attempts to access array elements not in the specified range. For example:

```
10 TREALS [] 10 DUMP

Array out of bounds 10
```

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. Store the upper bound of the array. At run time, see if the passed element number is within the range specified at compile time. If it is, find the address of the element by multiplying the passed element by the element size. Add this to the base address plus four (to skip the element size and the upper bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-BC <BUILDS
    DUP , OVER , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R 2+ @ OVER > OVER -1 >
    AND NOT IF
      OBM
    ELSE
      R> DUP @ ROT * + 4 +
    ENDIF ;
```

1?ARRAY-RNG (N1 N2 N3 -) (N - A)

Define and allocate space for a one-dimensional array with a specified element size. The lower and upper bounds of the array will also be specified.

Stack on Entry: (Compile Time) (N1) - The lower bound of the array.
(N2) - The upper bound of the array.
(N3) - Size, in bytes, of each array entry.
(Run Time) (N) - Position in the array to be accessed.

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

Example of Use:

```
1900 1999 4 1?ARRAY POPULATION
```

This would define a 100-element array, each element being four bytes in length, called POPULATION. If each entry was a double-length number, this would print out the value of the eighty-seventh element of the array:

```
1986 POPULATION 2@ D.
```

Note that the array elements are numbered 1900–1999.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. Store the lower bounds of the array. At run time, convert the passed element to an absolute number by subtracting the lower bound. Find the address of the element by multiplying the absolute element by the element size. Add this to the base address plus four (to skip the element size and the lower bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-RNG <BUILDS
    DUP , 3 PICK , LROT SWAP - 1+ *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R 2+ @ - R> DUP @ ROT * + 4 + ;
```

1?ARRAY-RNG-BC (N1 N2 N3 -) (N - A)

Define and allocate space for a one-dimensional array with a specified element size. The lower and upper bounds of the array will also be specified. Bounds checking will be included.

Stack on Entry: (Compile Time) (N1) – The lower bound of the array.
(N2) – The upper bound of the array.
(N3) – Size, in bytes, of each array entry.
(Run Time) (N) – Position in the array to be accessed.

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

Example of Use:

```
42 51 2 1?ARRAY BUILD/ST
```

This would define a 10-element array, each element being two bytes in length, called BUILD/ST. The following would print the value of the first entry of BUILD/ST:

```
42 BUILD/ST ?
```

Since this array has bounds checking, an error will occur if it is passed an invalid element, like this:

```
66 BUILD/ST ?
Array out of bounds 66
```

Algorithm: At compile time, store the lower and upper bounds. Store the size of the array entries. Allocate space for the array. The number of bytes needed is the number of elements times the size of each element. At run time, see if the passed element is within the range specified at compile time. If it is, convert the passed element to an absolute number by subtracting the lower bound. Find the address of the element by multiplying the absolute element by the element size. Add this to the base address plus six (to skip the element size and the upper and lower bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-RNG-BC <BUILDS
    DUP , 3 PICK , OVER , LROT SWAP - 1+
    * HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ >= >R
    4 + @ <= R> AND NOT IF
      OBM
    ELSE
      R> DUP >R 2+ @ -
      R> DUP @ ROT * + 6 +
    ENDIF ;
```

ARRAY (Nx N1 N2 –) (Nx – A)

Define and allocate space for an array of any dimensionality with each entry a specified size.

Stack on Entry: (Compile Time) (Nx) – One size for each dimension.
(N1) – The number of dimensions.
(N2) – Size, in bytes, of each array entry.
(Run Time) (Nx) – One position for each dimension of the array.

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

Example of Use:

```
5 4 3 2 4 2 ARRAY EIGEN-VS
```

This would define a 5-by-4-by-3-by-2 four-dimensional array called EIGEN-VS. This word can handle arrays of any size, since it is limited only by memory. Each element is two bytes in length. This array has 120 elements. The code below compares the first and last elements of the array:

```
0 0 0 0 EIGEN-VS @ 4 3 2 1 EIGEN-VS @ =
```

Note that the array elements are numbered 0–5, 0–4, 0–3, and 0–2.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. At run time multiply, each array element by the size of the array indices below it. Multiply that number by the entry size. Add it to the base address. Add 4 plus 2 times the number of array elements minus one, to skip the element sizes.

Suggested Extensions: Extend this word to use memory outside the 64K address space.

Definition:

```
0 VARIABLE ADR
0 VARIABLE ECOUNT
0 VARIABLE COUNTER
0 VARIABLE #ROLL

: ARRAY <BUILDS
    DUP , >R DUP , 1- BEGIN
      DUP 0 <>
    WHILE
      >R DUP , *
      R> 1-
    REPEAT DROP
```

```
    R> *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    ECOUNT 0SET DUP ADR ! 2 + @ DUP #ROLL !
    DUP 1- 2* ADR @ + 2+ COUNTER !
      1- BEGIN
      DUP 0 <>
    WHILE
      >R COUNTER @ @ -2 COUNTER +!
      #ROLL @ 1+ ROLL * ECOUNT +!
      -1 #ROLL +! R> 1-
    REPEAT DROP
    ECOUNT +! ECOUNT @ ADR @ @ *
    ADR @ 2+ @ 1- 2* 4 + + ADR @ + ;
```

DISK-2-MEM (D N1 N2 -)

Move data from disk to memory.

Stack on Entry: (D) – The disk address to move from.
(N1) – The memory address to move to.
(N2) – Number of bytes to move.

Stack on Exit: Empty.

Example of Use:

```
0, BOOT-SECTOR 512 DISK-2-MEM
```

This would move the first 512 bytes from a disk to the array of memory pointed to by BOOT-SECTOR.

Algorithm: Store the number of bytes and memory address in variables. Convert the disk address to a block and offset by dividing by 1024. Start a loop and fetch the block. Add the offset. Move the number of bytes left in the block or the number of bytes left to move, whichever is smaller. Decrement the number of bytes left to move by the number of bytes moved. Add one to the block being moved and zero the offset. Continue the loop if there are any bytes left to move.

Suggested Extensions: None.

Definition:

```
0 VARIABLE BYTES     0 VARIABLE MEM
```

```
0 VARIABLE #BLOCK     0 VARIABLE OA

: DISK-2-MEM
  BYTES ! MEM ! 1024 U/MOD #BLOCK !
  OA ! BEGIN
    #BLOCK @ BLOCK OA @ + MEM @
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

MEM-2-DISK (N1 D N2 -)

Move data from disk to memory.

Stack on Entry: (N1) – The memory address to move from.
(D) – The disk address to move to.
(N2) – Number of bytes to move.

Stack on Exit: Empty.

Example of Use:

```
BOOT-SECTOR 0, 512 MEM-2-DISK
```

This would move 512 bytes to the first block of a disk from the array of memory pointed to by BOOT-SECTOR.

Algorithm: Store the number of bytes and memory address in variables. Convert the disk address to a block and offset by dividing by 1024. Start a loop. Move the number of bytes left in the block (after adding the offset) or the number of bytes left to move, whichever is smaller. Decrement the number of bytes left to move by the number of bytes moved. Add one to the block being moved and zero the offset. Continue the loop if there are any bytes left to move.

Suggested Extensions: None.

Definition:

```
: MEM-2-DISK
  BYTES ! 1024 U/MOD #BLOCK ! OA !
  MEM ! BEGIN
```

```
    MEM @ #BLOCK @ BLOCK UPDATE OA @ +
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

(This version flushes all buffers before it begins, but is better for larger moves.)

```
: MEM-2-DISK
  FLUSH
  BYTES ! 1024 U/MOD #BLOCK ! OA !
  MEM ! BEGIN
    MEM @
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    DUP 1024 = IF
      #BLOCK @ BUFFER SWAP CMOVE UPDATE
    ELSE
      #BLOCK @ BLOCK OA @ + SWAP CMOVE
      UPDATE
    ENDIF
    R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

DISK-ARRAY (Nx N1 N2 D -) (Nx - D)

Define and allocate space for an array of any dimensionality with each entry a specified size. The array will reside on disk at a specified address.

Stack on Entry: (Compile Time) (Nx) - One size for each dimension.
(N1) - The number of dimensions.
(N2) - Size, in bytes, of each array entry.
(D) - The starting disk position for the entry.
(Run Time) (Nx) - One position for each dimension of the array.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The disk address of the array element.

Example of Use:

```
100 2 16 100,000 DISK-ARRAY QAZ
```

This would define a 100-by-2 disk array named QAZ. Each element of QAZ is sixteen bytes long. This would move an element of QAZ into a sixteen-byte buffer called 16BB.

```
8 1 QAZ 16BB 16 DISK-2-MEM
```

Algorithm: At compile time, store the size of each element. Store the starting disk address. At run time, multiply each array element by the size of the array indices below it. Multiply that number by the entry size. Add it to the base disk address.

Suggested Extensions: None.

Definition:

```
0 VARIABLE ADR
0 VARIABLE ECOUNT
0 VARIABLE COUNTER
0 VARIABLE #ROLL

: DISK-ARRAY <BUILDS
    ,,
    , DUP , 1- BEGIN
      DUP 0 <>
    WHILE
      >R DUP , *
      R> 1-
    REPEAT
    2DROP
  DOES> [ ->
    4 +
    ECOUNT 0SET DUP ADR ! 2 + @ DUP #ROLL !
    DUP 1- 2* ADR @ + 2+ COUNTER !
    1- BEGIN
      DUP 0 <>
    WHILE
      >R COUNTER @ @ -2 COUNTER +!
      #ROLL @ 1+ ROLL * ECOUNT +!
      -1 #ROLL +! R> 1-
```

```
    REPEAT DROP
    ECOUNT +! ECOUNT @ ADR  @ @ *
    ADR @ 4 - 2 @ ROT M+ ;
```

RECORD (-) (- N)

Start a record definition.

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

Stack on Exit: (Compile Time) Empty.
(Run Time) (N) – The number of bytes in the record.

Example of Use:

```
RECORD PERSON
```

This would start the definition of a record called PERSON.

Algorithm: At compile time, store a zero in the dictionary. This cell will be filled in by END–RECORD with the size of the record. Store the address of the cell in FILL–ADDR for END–RECORD.

Suggested Extensions: None.

Definition:

```
0 VARIABLE CUR-SIZE
0 VARIABLE FILL-ADDR
0 VARIABLE VAR-START
0 VARIABLE %VAR-END

: RECORD <BUILDS
     HERE FILL-ADDR !
     CUR-SIZE 0SET 0 ,
  DOES>
     @ ;
```

FIELD (N -) (A1 - A2)

Define a field in a record.

Stack on Entry: (Compile Time) (N) – The number of bytes to allocate to the field.

(Run Time) (A1) – The base address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) – The address of the field in the record.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
```

This would define NAME, a field of 16 bytes in the record PERSON. If JAMES was an instance of the record PERSON, this code would leave the address of the name field of JAMES on the stack:

```
JAMES NAME
```

Algorithm: Store the current size of the record at compile time. Then add the size of the current field to CUR–SIZE. At run time, add the number stored to the address on the stack.

Suggested Extensions: None.

Definition:

```
: FIELD <BUILDS
    CUR-SIZE DUP @ , +!
  DOES>
    @ + ;
```

VARIANT (–)

Start a variant in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
  1 FIELD SEX
  START-VARIANTS
  VARIANT
    2 FIELD #WIVES
```

```
  END-VARIANT
  VARIANT
    2 FIELD #HUSBANDS
    2 FIELD #CHILDREN
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

VARIANT starts two variants in this record.

Algorithm: Store the current size of the record in the variant start. This will be used by END-VARIANT to reset CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: VARIANT
  CUR-SIZE @ VAR-START ! ;
```

START-VARIANTS (-)

Start a set of variants in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PART
  2 FIELD LOCATION
  1 FIELD TYPE
  START-VARIANTS
  VARIANT
    2 FIELD ALENGTH
  END-VARIANT
  VARIANT
    2 FIELD BLENGTH
    2 FIELD WEIGHT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

START-VARIANTS must be used before any variants are defined.

Algorithm: Zero %VAR–END. This will be used to hold the largest variant defined, so the CUR–SIZE can be correctly set when the variants are complete.

Suggested Extensions: None.

Definition:

```
: START-VARIANTS
   %VAR-END 0SET ;
```

END-VARIANT (–)

End the definition of a variant portion of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD QUASAR
   16 FIELD NAME
   1 FIELD RAD-TYPE
   START-VARIANTS
   VARIANT
      1 FIELD ALPHA
      1 FIELD BETA
   END-VARIANT
   VARIANT
      1 FIELD GAMMA
   END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

END–VARIANT ends two variants in this record.

Algorithm: Store the larger of CUR–SIZE and %VAR–END in %VAR–END. Store the VAR–START in CUR–SIZE.

Suggested Extensions: None.

Definition:

```
: END-VARIANT
```

```
  CUR-SIZE DUP >R @
% VAR-END DUP >R @
  MAX R> !
  VAR-START @ R> ! ;
```

END-ALL-VARIANTS (-)

Complete the definition of a set of variants in a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PITCH
   4 FIELD NAME
   2 FIELD ROTATION
   START-VARIANTS
   VARIANT
      1 FIELD XROT
   END-VARIANT
   VARIANT
      1 FIELD YROT
      1 FIELD ZROT
   END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

END-ALL-VARIANTS must be used after a set of variants has been defined.

Algorithm: Set CUR-SIZE to the end of the largest variant.

Suggested Extensions: None.

Definition:

```
: END-ALL-VARIANTS
   %VAR-END@ CUR-SIZE ! ;
```

INSERT (–) (A1 – A2)

Insert a record as a subrecord in a record being defined.
Used as: INSERT <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) Empty.
(Run Time) (A1) – The base address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) – The address of the sub-record in the record.

Example of Use:

```
RECORD PLAYER
  INSERT PINFO OF PERSON
  1 FIELD POSITION
  2 FIELD RATING
END-RECORD
```

If BOUTON was an instance of PLAYER, the following code would leave the address of the name field, from the subrecord, on the stack:

```
BOUTON PINFO NAME
```

Algorithm: At compile time, skip the word AS. Look up the record word with "tick" and execute it. Store the size of the record. At run time, add the start of the sub-record to the address on the stack.

Suggested Extensions: None.

Definition:

```
: INSERT <BUILDS
     BL WORD DROP ' EXECUTE
     CUR-SIZE DUP @ , + !
  DOES>
     @ + ;
```

END-RECORD (–)

Complete the definition of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD SALE
   3 FIELD SALESPERSON
   2 FIELD CLIENT
   4 FIELD AMOUNT
END-RECORD
```

END-RECORD must be used for each record.

Algorithm: Fill in the size of the record.

Suggested Extensions: None.

Definition:

```
: END-RECORD
   CUR-SIZE @ FILL-ADDR @ ! ;
```

1AFIELD (N1 N2 –) (A1 N – A2)

Define an array field in a record.

Stack on Entry: (Compile Time) (N1) – The number of entries.
(N2) – The number of bytes to allocate to each entry.
(Run Time) (N1) – The base address of the record.
(N) – The entry to access.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) – The address of the field in the record.

Example of Use:

```
RECORD BBALL-SCORE
   4 2 FIELD HOME-SCORE
   4 2 FIELD AWAY-SCORE
END-RECORD
```

HOME-SCORE and AWAY-SCORE will de defined as array field with

four entries each. The code below would compare two scores stored in an instance of this record called 1GAME.

```
1GAME 2 HOME-SCORE @ 1GAME 2 AWAY-SCORE @ =
```

Algorithm: At compile time, store the size of each entry. Add the size of the total field to CUR-SIZE. At run time, multiply the requested entry number by the entry size and add it to the base address on the stack.

Suggested Extensions: Bounds checking, range arrays, and multiple-dimension array fields could be defined if desired.

Definition:

```
: 1AFIELD <BUILDS
    DUP, CUR-SIZE DUP @ ,
    >R * R> +!
  DOES>
    DUP @ ROT * SWAP 2+ @ + + ;
```

INSTANCE (-) (- A)

Define an instance of a record.
Used as: INSTANCE <name> OF <RECORD-NAME>

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

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – The base address of the record.

Example of Use:

```
INSTANCE RUTH OF PLAYER
```

This would create an instance of the record PLAYER, called RUTH.

Algorithm: At compile time, skip the OF. "Tick" the record word and determine how many bytes long it is. Allocate space for the record. At run time, leave the address on the stack.

Suggested Extensions: None.

Definition:

```
: INSTANCE <BUILDS
     BL WORD DROP ' EXECUTE ALLOT
  DOES> ;
```

M-INSTANCE (N -) (N - A)

Define a multiple instance of a record.
Used as: <N> M-INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (N) The number of instances to define.
(Run Time) (N) The number of the instance desired.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – The base address of the record.

Example of Use:

```
25 M-INSTANCE YANKEES OF PLAYER
```

This would create an 25 element array of the record PLAYER, called YANKEES. This would print the name of the fourth player:

```
3 YANKEES NAME 16 TYPE
```

Algorithm: At compile time, skip the OF. "Tick" the record word and determine how many bytes long it is. Store the record length. Allocate space for all of the records. At run time, multiply the requested position by the length of each record. Leave the address on the stack.

Suggested Extensions: Bounds checking, ranges, and multiple dimensions can be defined, if needed.

Definition:

```
: M-INSTANCE <BUILDS
     BL WORD DROP
     'EXECUTE DUP , * ALLOT
  DOES>
     DUP @ ROT * + 2 + ;
```

RECORD (–) (– N)

(Disk Version)
Start a record definition.

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

Stack on Exit: (Compile Time) Empty.
(Run Time) (N) – The number of bytes in the record.

Example of Use:

```
RECORD PERSON
```

This would start the definition of a record called PERSON.

Algorithm: At compile time store a zero in the dictionary. This cell will be filled in by END–RECORD with the size of the record. Store the address of the cell in FILL–ADDR for END–RECORD.

Suggested Extensions: None.

Definition:

```
0 VARIABLE CUR-SIZE
0 VARIABLE FILL-ADDR
0 VARIABLE VAR-START
0 VARIABLE %VAR-END

: RECORD <BUILDS
     HERE FILL-ADDR !
     CUR-SIZE 0SET 0 ,
  DOES>
     @ ;
```

FIELD (N –) (D1 – D2)

(Disk Version)
Define a field in a record.

Stack on Entry: (Compile Time) (N) – The number of bytes to allocate to the field.
(Run Time) (D1) – The base disk address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the field in the record.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
```

This would define NAME, a field of 16 bytes in the record PERSON. If JAMES was an instance of the record PERSON, this code would leave the disk address of the name field of JAMES on the stack:

```
JAMES NAME
```

Algorithm: Store the current size of the record at compile time. Then add the size of the current field to CUR–SIZE. At run time, add the number stored to the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: FIELD <BUILDS
  CUR-SIZE DUP @ , + !
DOES>
  @ M+ ;
```

VARIANT (–)

(Disk Version)
Start a variant in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
  1 FIELD SEX
  START-VARIANTS
  VARIANT
    2 FIELD #WIVES
  END-VARIANT
  VARIANT
    2 FIELD #HUSBANDS
```

```
    2 FIELD #CHILDREN
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

VARIANT starts two variants in this record.

Algorithm: Store the current size of the record in the variant start. This will be used by END-VARIANT to reset CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: VARIANT
  CUR-SIZE @ VAR-START ! ;
```

START-VARIANTS (-)

(Disk Version.)
Start a set of variants in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PART
  2 FIELD LOCATION
  1 FIELD TYPE
  START-VARIANTS
  VARIANT
    2 FIELD LENGTH
  END-VARIANT
  VARIANT
    2 FIELD LENGTH
    2 FIELD WEIGHT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

START-VARIANTS must be used before any variants are defined.

Algorithm: Zero %VAR-END. This will be used to hold the largest variant

defined, so the CUR-SIZE can be correctly set when the variants are complete.

Suggested Extensions: None.

Definition:

```
: START-VARIANTS
   %VAR-END 0SET ;
```

END-VARIANT (-)

(Disk version.)
End the definition of a variant portion of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD QUASAR
   16 FIELD NAME
   1 FIELD RAD-TYPE
   START-VARIANTS
   VARIANT
      1 FIELD ALPHA
      1 FIELD BETA
   END-VARIANT
   VARIANT
      1 FIELD GAMMA
   END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

END-VARIANT ends two variants in this record.

Algorithm: Store the larger of CUR-SIZE and %VAR-END in %VAR-END. Store the VAR-START in CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: END-VARIANT
```

```
  CUR-SIZE DUP >R @
  %VAR-END DUP >R @
  MAX R> !
  VAR-START @ R> ! ;
```

END-ALL-VARIANTS (-)

(Disk version.)
Complete the definition of a set of variants in a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PITCH
  4 FIELD NAME
  2 FIELD ROTATION
  START-VARIANTS
  VARIANT
    1 FIELD XROT
  END-VARIANT
  VARIANT
    1 FIELD YROT
    1 FIELD ZROT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

END-ALL-VARIANTS must be used after a set of variants has been defined.

Algorithm: Set CUR-SIZE to the end of the largest variant.

Suggested Extensions: None.

Definition:

```
: END-ALL-VARIANTS
  %VAR-END@ CUR-SIZE ! ;
```

INSERT (–) (D1 – D2)

(Disk version.)
Insert a record as a subrecord in a record being defined.
Used as: INSERT <name> OF <RECORD–NAME>

Stack on Entry: (Compile Time) Empty.
(Run Time) (D1) – The base disk address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the sub-record in the record.

Example of Use:

```
RECORD PLAYER
   INSERT PINFO OF PERSON
   1 FIELD POSITION
   2 FIELD RATING
END–RECORD
```

If BOUTON was an instance of PLAYER, the following code would leave the disk address of the name field, from the subrecord, on the stack:

```
BOUTON PINFO NAME
```

Algorithm: At compile time, skip the word AS. Look up the record word with "tick" and execute it. Store the size of the record. At run time, add the start of the sub-record to the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: INSERT <BUILDS
     BL WORD DROP ' EXECUTE
     CUR–SIZE DUP @ , +!
  DOES>
     @ M+ ;
```

END–RECORD (–)

(Disk version.)
Complete the definition of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD SALE
  3 FIELD SALESPERSON
  2 FIELD CLIENT
  4 FIELD AMOUNT
END-RECORD
```

END-RECORD must used for each record.

Algorithm: Fill in the size of the record.

Suggested Extensions: None.

Definition:

```
: END-RECORD
  CUR-SIZE @ FILL-ADDR @ ! ;
```

1AFIELD (N1 N2 –) (D1 N – D2)

(Disk version.)
Define an array field in a record.

Stack on Entry: (Compile Time) (N1) – The number of entries.
(N2) – The number of bytes to allocate to each entry.
(Run Time) (D1) – The base disk address of the record.
(N) – The entry to access.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the field in the record.

Example of Use:

```
RECORD BBALL-SCORE
  4 2 FIELD HOME-SCORE
  4 2 FIELD AWAY-SCORE
END-RECORD
```

HOME-SCORE and AWAY-SCORE will be defined as array fields with

four entries each. The code below would compare two scores stored in an instance of this record called 1GAME.

```
1GAME 2 HOME-SCORE 1B 2 DISK-2-MEM 1B @
1GAME 2 AWAY-SCORE 1B 2 DISK-2-MEM 1B @ =
```

Algorithm: At compile time, store the size of each entry. Add the size of the total field to CUR-SIZE. At run time, multiply the requested entry number by the entry size and add it to the base address on the stack.

Suggested Extensions: Bounds checking, range arrays, and multiple dimension array fields could be defined if desired.

Definition:

```
: 1AFIELD <BUILDS
    DUP , CUR-SIZE DUP @ ,
    >R * R> +!
  DOES>
    DUP @ ROT * SWAP 2+ @ + M+ ;
```

INSTANCE (D –) (– D)

(Disk version.)
Define an instance of a record.
Used as: <D> INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (D) – The starting disk address of the record.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The base disk address of the record.

Example of Use:

```
45,000 INSTANCE RUTH OF PLAYER
```

This would create an instance of the record PLAYER, called RUTH. It would start at position 45,000 in disk storage.

Algorithm: At compile time, store the start position. Skip the OF. "Tick" the record word and determine how many bytes long it is. At run time, fetch the start and leave the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: INSTANCE <BUILDS
    ' '
    BL WORD DROP ' EXECUTE DROP
  DOES> DUP @ SWAP 2+ @ SWAP ;
```

M-INSTANCE (D N -) (N - D)

(Disk version.)
Define a multiple instance of a record.
Used as: <N> <D> M-INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (N) – The number of instances to define.
(D) – The starting disk address of the records.
(Run Time) (N) – The number of the instance desired.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The base disk address of the record.

Example of Use:

```
25 250,000 M-INSTANCE YANKEES OF PLAYER
```

This would create an 25-element array of the record PLAYER, called YANKEES. It would be stored starting at disk location 250,000. This would print the name of the fourth player:

```
3 YANKEES NAME 16B 16 DISK-2-MEM 16B 16 TYPE
```

Algorithm: At compile time, store the start disk address. Skip the OF. "Tick" the record word and determine how many bytes long it is. Store the record length. At run time, multiply the requested position by the length of each record. Add it to the disk start address. Leave the address on the stack.

Suggested Extensions: Bounds checking, ranges, and multiple dimensions can be defined if needed.

Definition:

```
: M-INSTANCE <BUILDS
```

```
  , , BL WORD DROP DROP
    'EXECUTE ,
DOES>
  DUP 4 + @ ROT * >R DUP @ SWAP 2+ @ SWAP R> M+ ;
```

TO (–)

Cause a "to" type variable to execute a store.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
0 TO TIME
```

If TIME has been defined as a "to" type variable, the above code will store a zero in TIME.

Algorithm: Set the variable &TO to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE &TO: TO &TO C1SET ;
```

TVARIABLE (N –) ((N1) – (N2))

Define and initialize a cell-sized "to" type variable.

Stack on Entry: (Compile Time) (N) – The initial value of the variable.
(Run Time) (N1) – The value to store in the variable, if TO is being used.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N2) – The value of the variable, if TO is not used.

Example of Use:

```
0 TVARIABLE TIME
```

TIME would be defined as a cell sized "to" type variable.

Algorithm: At compile time, store the initial value. At run time, check the value of &TO. If it is true, store the value on the stack in the variable and reset &TO to false. If &TO was false, fetch the value of the variable.

Suggested Extensions: None.

Definition:

```
: TVARIABLE <BUILDS , DOES> &TO C@
  IF &TO C0SET ! ELSE @ ENDIF ;
```

TCVARIABLE (N –) ((N1) – (N2))

Define and initialize a byte sized "to" type variable.

Stack on Entry: (Compile Time) (N) – The initial value of the variable.
(Run Time) (N1) – The value to store in the variable, if TO is being used.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N2) – The value of the variable, if TO is not used.

Example of Use:

```
0 TVARIABLE BONZO
```

BONZO would be defined as a byte sized "to" type variable.

Algorithm: At compile time, store the initial value. At run time, check the value of &TO. If it is true, store the value on the stack in the variable and reset &TO to false. If &TO was false, fetch the value of the variable.

Suggested Extensions: None.

Definition:

```
: TCVARIABLE <BUILDS C, DOES> &TO C@
  IF &TO C0SET C! ELSE C@ ENDIF ;
```

FIFO (N –) (– A)

Create a queue or fifo stack.

Stack on Entry: (Compile time) (N) – The number of entries in the queue.
(Run Time) Empty.

Stack on Exit: (Compile time) Empty.
(Run time) (A) – The address of the queue.

Example of Use:

```
10 FIFO TASKS
```

TASKS would be defined as a queue or fifo stack with room for 10 cell-sized entries.

Algorithm: At compile time, store space for the start and end pointers for the queue. Store the queue size. Initialize the current start and end to 3. At run time, just leave the address.

Suggested Extensions: Words for different–sized elements would be a useful extension.

Definition:

```
: FIFO <BUILDS 1+ DUP 2 + HERE ! HERE
  SWAP 3 + 2* ALLOT 2+ DUP 3 SWAP ! 2+ 3
  SWAP ! DOES> ;
```

(#FI) (A – A N1 N2)

Fetch the start and end pointers of a queue.

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

Stack on Exit: (A) – The address of the queue.
(N1) – The start pointer of the queue.
(N2) – The end pointer of the queue.

Example of Use: See words defined below.

Algorithm: The start is stored at the queue address plus two. The end is at the

queue address plus four. Fetch the two values.

Suggested Extensions: None.

Definition:

```
: (#FI) DUP DUP 2+ @ SWAP 4 + @ ;
```

?FIFO (A – F)

Determine if anything is stored in a queue.

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

Stack on Exit: (F) – A Boolean flag, true if any data are being held in the queue.

Example of Use:

```
TASKS ?FIFO .
```

This would print out a true flag if any data were being held in the queue TASKS.

Algorithm: If the start and the end pointers are equal, the queue is empty.

Suggested Extensions: None.

Definition:

```
: ?FIFO (#FI) <> SWAP DROP ;
```

CNO (A – F)

Abort with an error message if a queue is empty.

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

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the start and the end pointers are equal, the queue is empty. If this is the case, abort with an error message.

Suggested Extensions: None.

Definition:

```
: CNO (#FI) = ABORT" Fifo Error" ;
```

@FIFO (A – N)

Remove data from a queue.

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

Stack on Exit: (N) – The last cell entered in the queue not yet fetched.

Example of Use:

```
TASKS @FIFO
```

Algorithm: Check to make sure the queue is not empty. If it is not, fetch the cell pointed to by the start pointer. Increment the start pointer. Wrap around to three if it passes the end of the queue.

Suggested Extensions: None.

Definition:

```
: @FIFO C@ DUP DUP 2+ @ 2* + @ SWAP DUP
  2+ 1 SWAP +! DUP DUP @ SWAP 2+ @ <
  IF 3 SWAP 2+ ! ELSE DROP ENDIF ;
```

!FIFO (N A –)

Store data in a queue.

Stack on Entry: (N) – The cell to store in the queue.
(A) – The address of the queue.

Stack on Exit: Empty.

Example of Use:

```
99 TASKS !FIFO
```

This would store a 99 in the queue TASKS.

Algorithm: Store the number in the position pointed to by the end pointer. Increment the pointer and make sure it does not pass the the start pointer. If it does, the queue has overflowed.

Suggested Extensions: None.

Definition:

```
: !FIFO DUP DUP 4 + @ 2* + ROT SWAP !
   DUP 4 + 1 SWAP +! DUP DUP @ SWAP 4 +
   @ < IF DUP 3 SWAP 4 + ! ENDIF CNO
   DROP ;
```

FIFO-RESET (A -)

Empty a queue.

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

Stack on Exit: Empty.

Example of Use:

```
TASKS FIFO-RESET
```

This would set the queue TASKS at empty.

Algorithm: Store a 3 in the start and end pointers.

Suggested Extensions: None.

Definition:

```
: FIFO-RESET DUP 2 + 3 SWAP! 4 + 3
   SWAP! ;
```

12

Expert Systems

Words Defined in This Chapter:

LTEXT	A string array that holds the text for conditions in the expert system.
LT-STATE	A byte array that holds the state of each string condition in the expert system.
RULES	A cell array that holds the address of each rule defined.
R	A byte array that holds the logical value of the level being analyzed.
R#LU	A variable holding the number of rules defined.
LT#LU	A variable holding the number of strings defined.
RULE	Start the definition of a rule.
END-RULE	Complete the definition of a rule.
GETS	Store a string, delimited by braces, in the variable STEMP.
EXISTS	Determine if a string is already stored in the expert system.
!LT	Add a string to the expert system.
SCOMPILE	Compile a string in a rule definition.
WHEN	Compile a WHEN condition in a rule definition.
&	Compile an AND condition in a rule definition.
|	Compile an OR condition in a rule definition.
HYPOTH	Compile an hypothesis in a rule definition.
EXPLAIN	Compile an explanation in a rule definition.
T/F	Prompt the user for and return a Boolean value.
F-S	Convert a Boolean flag to a literal state.

ASK-USER	Determine the state of a string in the expert system by asking the user.
[FIND-RULE]	Find a rule with a specific string as an hypothesis.
RULE-EVAL	A forward definition of the word RULE-EVAL.
UNKNOWN$	Attempt to set the state of an unknown string.
S-F	Convert a literal state to a Boolean flag.
$-EVAL	Leave the state of a string in the expert system.
COND-EVAL	Determine the truth of a condition.
+WHEN	Evaluate a WHEN condition in a rule.
-WHEN	Evaluate a WHEN NOT condition in a rule.
+AND	Evaluate an AND condition in a rule.
-AND	Evaluate an AND NOT condition in a rule.
+OR	Evaluate an OR condition in a rule.
-OR	Evaluate an OR NOT condition in a rule.
C/RULE-EVAL	Evaluate the conditional statements of a rule.
DO-EXPLAIN	Cause the expert system rule interpreter to explain its conclusions.
DON'T-EXPLAIN	Cause the expert system rule interpreter to suppress explanations.
.EXPLAIN	Execute the EXPLAIN clause of a rule definition.
DO-MON	Cause the expert system rule interpreter to print out all hypotheses reached.
DON'T-MON	Cause the expert system rule interpreter only to print the final conclusion reached.
.MON	Print the result of an HYPOTH statement.
TELL?	Should a Forth word hypothesis print out what it is doing?
$APPLY-RULE	Set the string hypothesis of a rule.
WORD-APPLY-RULE	Set the word hypothesis of a rule.
RULE-EVAL	Evaluate a rule.
RESET-SYSTEM	Clear all variables used by the expert system.
RT?	Is a rule's hypothesis known?
APPLY	Apply the rules of an expert system.

Expert systems are computer programs that attempt to duplicate the ability of human experts in a particular area of knowledge. Some of the most interesting work in computer science is taking place in Artificial Intelligence, or AI, and expert systems are among the most intriguing aspects of AI. Expert systems exist for medical diagnosis, searching for oil, and even configuring computer systems.

This chapter presents a set of words that will enable you to design your own expert system. Included in this chapter is a simple sample expert system, one that advises a baseball manager when to attempt a sacrifice bunt. The words presented in this chapter could be used to design an expert system on any subject.

Conceptually, an expert system can be viewed as two distinct parts. The first part is what is known as the rules. This is the format in which the expert knowledge we are trying to put to use is encoded. Here is a simple rule in the format of this chapter's expert system:

```
WHEN | A PERSON OWNS A ROLLS ROYCE |
HYPOTH | THE PERSON IS RICH |
```

Our rule says if a person owns a Rolls Royce, he or she must be rich. As we can see from this example, an expert system can only be as accurate as its rules. Obviously, a person could own a Rolls Royce and not be rich. If the hypothesis of this rule could be retracted later when other evidence was examined, then this rule might be more acceptable. A more sophisticated expert system than the one presented in this chapter would allow this back-tracking. In this expert system, once a conclusion is reached, it cannot be changed. Since this is the case, our rule should be more complete. Our one rule probably should be expanded.

```
WHEN | A PERSON OWNS AN EXPENSIVE CAR |
& | A PERSON OWNS AN EXPENSIVE HOUSE |
HYPOTH | THE PERSON IS RICH |

WHEN CASH>1,000,000
HYPOTH | THE PERSON IS RICH |

WHEN | A PERSON OWNS A ROLLS ROYCE |
|  | A PERSON OWNS A BENTLY |
|  | A PERSON OWNS A LOTUS |
HYPOTH | A PERSON OWNS AN EXPENSIVE CAR |
```

This set of rules has a few new wrinkles thrown in. Our conditions now include the logical operators AND (represented by &) and OR (represented by |). This allows the first rule, which is: if a person owns an expensive car, and a person owns an expensive house, then the person must be rich.

The second rule in the above example includes something not in between braces, which our expert system uses for literals. This is a normal Forth word. Forth words will be available in our expert system so that computation can be performed. All Forth words used in rules should leave a Boolean flag on the stack. Here is how CASH>1,000,000 might be written:

```
: CASH>1,000,000 CASH 2@ 1.000000 D> ;
```

A NOT operator (represented by --) is also available. Here is how it might be used:

```
WHEN -- | A PERSON OWNS AN EXPENSIVE CAR |
& -- | A PERSON OWNS AN EXPENSIVE HOUSE |
HYPOTH | THE PERSON DOES NOT HAVE AN OPULENT LIFE STYLE |
```

It will be important in our expert system to represent opposite conditions with the same literal, and to use the negation operator. If this is not done, our rule interpreter, which we'll get to in a moment, will not know we are talking about the same thing. For example:

```
WHEN | A PERSON IS MALE |
HYPOTH | THE PERSON HAS AN XY CHROMOSOME PAIR |

WHEN | A PERSON IS FEMALE |
HYPOTH | THE PERSON HAS AN XX CHROMOSOME PAIR |
```

These rules would get the computer to first ask if the person was male, and then to ask if the person was female. A better way to encode these rules would be:

```
WHEN | A PERSON IS MALE |
HYPOTH | THE PERSON HAS AN XY CHROMOSOME PAIR |

WHEN -- | A PERSON IS FEMALE |
HYPOTH | THE PERSON HAS AN XX CHROMOSOME PAIR |
```

In this way, the expert system can take advantage of knowledge it has already accumulated.

The second part of our expert system is the rule compiler and the rule interpreter. The rule compiler will read in rules as described previously and store them in memory. Our rule interpreter, the part of an expert system sometimes known as an inference engine, will try to use the rules to reach some conclusion. Here is how a rule will appear in memory:

```
0-2 Hypothesis Condition: 9 - Hypothesis True
                          10 - Hypothesis False
3-5 Hypothesis: Word/String Entry.
6-8 Explanation: Word/String Entry.
```

Then any number of the following:

```
9-11 Condition: 0 - End of List.
                3 - WHEN
4 - WHEN NOT
5 - AND
6 - AND NOT
7 - OR
8 - OR NOT
```

12–15 Word/String Entry

A Word/String Entry looks like this:

```
Byte 0: 1 – String
        2 – Word

Bytes 1–2: String Number or Word Address
```

Strings are held in a string array, and the state of each string (true, false, or unknown) is also held in an array.

Suggested Extensions: This chapter has numerous possibilities for extension. Among the most useful extension would be the inclusion of variables in the string rules. This would allow rules like:

```
WHEN | *A IS RICH |
HYPOTH | *A COULD OWN AN EXPENSIVE CAR |
```

This type of rule has much more power than the constant literals used in the expert system presented in this chapter.

Another useful extension would be to allow the retraction of hypotheses, assigning probabilities to conclusions, and parenthesized conditions in the rules. All in all, one could spend a lot of time with this chapter.

Please note: The words in this chapter make use of case statements from Chapter 2, strings from Chapter 7, and array words from Chapter 11.

LTEXT (N – A)

A string array which holds the text for conditions in the expert system.

Stack on Entry: (N) – The array member to access.

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

Example of Use:

```
8 LTEXT $?
```

This would print the value of string number eight in the expert system.

Algorithm: Define a string array that allows each string to be up to 64 characters in length.

Suggested Extensions: None.

Definition:

```
100 64 SARRAY LTEXT
```

LT-STATE (N - A)

A byte array that holds the state of each string literal in the expert system.

Stack on Entry: (N) – The array member to access.

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

Example of Use:

```
8 LT-STATE C?
```

This would print the state of string eight in our expert system.

Algorithm: Use 1CARRAY. The following values are used:

0 – Literal Unknown.
1 – Literal Known to be True.
2 – Literal Known to be False.

Suggested Extensions: None.

Definition:

```
100 1CARRAY LT-STATE
```

RULES (N - A)

A cell array that holds the address of each rule defined.

Stack on Entry: (N) – The array member to access.

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

Example of Use:

```
8 RULES @ 20 DUMP
```

This would dump the first 20 bytes of rule eight.

Algorithm: Use 1ARRAY.

Suggested Extensions: None.

Definition:

```
100 1ARRAY RULES
```

R [] (N – A)

A byte array that holds the logical value of the level being analyzed.

Stack on Entry: (N) – The array member to access.

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

Example of Use:

```
1 R [] C?
```

This would print the Boolean value at level one.

Algorithm: Use 1CARRAY.

Suggested Extensions: Thirty-two levels is probably enough to handle even quite complex rule systems. More could be added if desired by enlarging the array.

Definition:

```
32 1CARRAY R []
```

R#LU (– A)

A variable holding the number of rules defined.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of R#LU.

Example of Use:

```
R#LU ?
```

This would print the number of rules defined in the expert system.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE R#LU
```

LT#LU (– A)

A variable holding the number of strings defined.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of LT#LU.

Example of Use:

```
1 LT#LU +!
```

This would increment the number of strings stored.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE LT#LU
```

RULE (–)

Start the definition of a rule.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RULE
   WHEN | CLOUDS ARE IN THE SKY |
   HYPOTH | IT IS A BAD DAY TO TAN |
END-RULE
```

RULE must be used to start each rule definition.

Algorithm: Print a message. Store the current dictionary pointer in the array RULES. This is the start address of the rule being defined. Allot 9 bytes at this address to hold the hypothesis and the explanation.

Suggested Extensions: Have this word abort if too many rules are defined.

Definition:

```
: RULE
   CR ." Defining Rule ."
   R#LU DUP ? @
   HERE SWAP RULES !
   HERE 9 ERASE 9 ALLOT ;
```

ENDRULE (-)

Complete the definition of a rule.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RULE
   WHEN | THE EYES ARE CLOSED |
      & | BREATHING IS REGULAR |
   HYPOTH | THE PATIENT IS ASLEEP |
END-RULE
```

END-RULE must be used to finish each rule definition.

Algorithm: Store zero in the dictionary. This terminates the list of conditions. Increment the number of rules.

Suggested Extensions: None.

Definition:

```
: END-RULE
   0 C, 1 R#LU +! ;
```

GET$ (–)

Store a string, delimited by braces, in the variable $TEMP.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use WORD.

Suggested Extensions: None.

Definition:

```
125 CCONSTANT    | KEY

64 $VARIABLE $TEMP

: GETS
   | KEY WORD $TEMP $! ;
```

EXIST$ (– (N) F)

Determine if a string is already stored in the expert system.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the string is found, false if it is not.
(N) – The number of the string if it does not already exist.

Example of Use: See words defined below.

Algorithm: If no strings are defined yet, exit the word with a false flag. Loop through all the strings defined. Compare the strings with $=. Exit the loop if a match is found.

Suggested Extensions: None.

Definition:

```
: $EXIST?
   LT#LU @ ?DUP IF
      0 SWAP 0 DO
         I LTEXT $@ STEMP $@ $= IF
            DROP I -1 LEAVE
         ENDIF
      LOOP
   ELSE
      0
   ENDIF ;
```

!LT (-)

Add a string to the expert system.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... EXITSS NOT IF !LT ENDIF ...
```

This would add the string in STEMP to the expert system if it does not yet exist in the system.

Algorithm: Print a message. Store the string in the LTEXT array. Increment the number of strings in the system.

Suggested Extensions: Have this word abort if too many strings are defined.

Definition:

```
: !LT
  CR ." Compiling " $TEMP $?
  ." as string # " LT#LU ?
  $TEMP $@ LT#LU @ LTEXT $!
  1 LT#LU +! ;
```

$COMPILE (–)

Compile a string in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the string exists, just store its number in the dictionary. If it does not, add the string to the system, then store its number in the dictionary.

Suggested Extensions: None.

Definition:

```
: $COMPILE
   $EXIST? IF

   ELSE
      LT#LU @ , !LT
   ENDIF ; –>
```

WHEN (–)

Compile a WHEN condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
WHEN – – | THE WALL IS GREEN |
```

This would compile a "while not string" condition in the expert system.

Algorithm: Search for a negation. If one is found, compile a WHEN NOT condition and parse the next token in the input stream. If no negation is found, compile a WHEN condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled, make sure it is not

&, | , HYPOTH, or END-RULE. These are common error conditions.

Definition:

```
: -- ;

: { ;

: WHEN
   FIND [ ' -- ] LITERAL OVER = IF
      4 C, DROP FIND
   ELSE
      3 C,
   ENDIF 0 ,
   [ ' { ] LITERAL OVER = IF
      DROP 1 C, GET$ $COMPILE
   ELSE
      2 C, ,
   ENDIF ;
```

& (-)

Compile an AND condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
& 10%-INTEREST-RATE
```

This would compile an AND WORD condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an AND NOT condition and parse the next token in the input stream. If no negation is found, compile an AND condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled make sure it is not &, | , HYPOTH, or END-RULE. These are common error conditions.

Definition:

```
: &
```

```
FIND [ ' -- ] LITERAL OVER = IF
   6 C, DROP FIND
ELSE
   5 C,
ENDIF 0 ,
[ ' { ] LITERAL OVER = IF
   DROP 1 C, GET$ $COMPILE
ELSE
   2 C, ,
ENDIF ;
```

| (-)

Compile an OR condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
|   -- { PATIENT IS BLEEDING }
```

This would compile an OR NOT string condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an OR NOT condition and parse the next token in the input stream. If no negation is found, compile an OR condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled, make sure it is not &, | , HYPOTH, or END-RULE. These are common error conditions.

Definition:

```
:|
  FIND [ ' -- ] LITERAL OVER = IF
     8 C, DROP FIND
  ELSE
     7 C,
  ENDIF 0 ,
  [' { ] LITERAL OVER = IF
     DROP 1 C, GET$ $COMPILE
  ELSE
```

```
    2 C, ,
ENDIF ;
```

HYPOTH (–)

Compile an hypothesis in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
HYPOTH -- { RECESSION EXISTS }
```

This would compile an "hypothesis false string" condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an "hypothesis false" condition and parse the next token in the input stream. If no negation is found, compile an "hypothesis true" condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found. Store all these values in the space allocated when the rule was defined.

Suggested Extensions: When the next token is compiled, make sure it is not &, | , HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: RAD R#LU @ RULES @ ;

: HYPOTH
   FIND [' -- ] LITERAL OVER = IF
      10 RAD C! DROP FIND
   ELSE
      9 RAD C!
   ENDIF
   [ ' { ] LITERAL OVER = IF
      DROP 1 RAD 3 + C!
      GET$ $EXIST? IF
         RAD 4 + !
      ELSE
         LT#LU @ RAD 4 + ! !LT
      ENDIF
   ELSE
```

```
    RAD 4 + ! 2 RAD 3 + C!
  ENDIF ;
```

EXPLAIN (–)

Compile an explanation in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
EXPLAIN | THE MOON IS RISING |
```

This would compile an explain statement in the expert system.

Algorithm: Find the next token in the input stream. If is a left brace, compile a string. If it is not a brace, compile the word found. Store all these values in the space allocated when the rule was defined.

Suggested Extensions: When the next token is compiled, make sure it is not &, | , HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: EXPLAIN
   FIND [ ' { ] LITERAL OVER = IF
      DROP 1 RAD 6 + C!
      GET$ $EXIST? IF
         RAD 7 + !
      ELSE
         LT#LU @ RAD 7 + ! !LT
      ENDIF
   ELSE
      RAD 7 + ! 2 RAD 6 + C!
   ENDIF ;
```

T/F (– F)

Prompt the user for and return a Boolean value.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if a T is entered, false if an F is entered.

Example of Use: T/F .

This would prompt the user for a Boolean value, input it, and print it on the display.

Algorithm: Print the prompt "(T/F)". Input a keystroke. Continue until a "T" or "F" is input.

Suggested Extensions: None.

Definition:

```
: T/F
   CR ." (T/F) " 1 BEGIN
      DROP KEY DUP 95 > IF 32 - ENDIF
      DUP DUP 84 = SWAP 70 = OR
   UNTIL
   DUP EMIT 84 = ;
```

F-S (F – N)

Convert a Boolean flag to a literal state.

Stack on Entry: (F) – The Boolean flag to convert.

Stack on Exit: (N) – The state, 1 = True, 2 = False.

Example of Use:

```
T/F F-S 8 LT-STATE C!
```

This would set string eight's state to the value input by the user in T/F.

Algorithm: If the flag is true, leave a one; if it is false, leave a two.

Suggested Extensions: None.

Definition:

```
: F-S IF 1 ELSE 2 ENDIF ;
```

ASK-USER (N -)

Determine the state of a string in the expert system by asking the user.

Stack on Entry: (N) – The string to ask about.

Stack on Exit: Empty.

Example of Use:

```
8 ASK-USER
```

This would ask the user to set the state of string eight.

Algorithm: Print the string and prompt. Get a true or false from the user. Store it in LT-STATE.

Suggested Extensions: Extend the system to allow a don't know input. More than just this word would have to be modified for this extension.

Definition:

```
: ASK-USER
  CR ." Is the following condition true
  or false ? "
  CR DUP LTEXT $? T/F
  F-S SWAP LT-STATE C! ;
```

IND-RULE (N1 N2- (N3) F)

Find a rule with a specific string as an hypothesis.

Stack on Entry: (N1) – The string to ask about.
(N2) – The start position in the rule list to search.

Stack on Exit: (N3) – The rule number is one is found.
(F) – A Boolean flag, true if a rule is found; false if no rule is found.

Example of Use:

```
8 0 FIND-RULE
```

This would search for a rule that has string eight in its hypothesis.

Algorithm: Search through the array RULES. Start at the position passed on the stack. Check the hypothesis field of each rule, first for a string, and then for the particular string being sought. If it is found, exit the loop.

Suggested Extensions: None.

Definition:

```
: FIND-RULE
0 LROT R#LU @ SWAP DO
  I RULES @ 3 + C@ 1 = IF
    I RULES @ 4 + @ OVER = IF
      2DROP I -1 0 LEAVE
    ENDIF
  ENDIF
LOOP DROP ;
```

[RULE-EVAL] (-)

A forward definition of the word RULE-EVAL.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: The address of the word RULE-EVAL will be stored in /RULE-EVAL/. This word fetches that address and executes it.

Suggested Extensions: None.

Definition:

```
: DUMMY ;

' DUMMY VARIABLE /RULE-EVAL/

: [RULE-EVAL] /RULE-EVAL/ @ EXECUTE ;
```

S-F (N - F)

Convert a literal state to a Boolean flag.

Stack on Entry: (N) - The state, 1 = True, 2 = False.

Stack on Exit: (F) – The Boolean flag.

UNKNOWN$ (N –)

Attempt to set the state of an unknown string.

Stack on Entry: (N) – The number of the string to set.

Stack on Exit: Empty.

Example of Use:

```
8 UNKNOWNS
```

This would attempt to ascertain the state of string number eight.

Algorithm: Use FIND–RULE to see if any rules have this string as an hypothesis. If a rule is found, attempt to evaluate it. If the rule sets the state of the string, exit the word. If it does not, continue looking for other rules that may have the string as an hypothesis. If no rules are found, or none set the state of the string, the string will be left in an unknown state.

Suggested Extensions: None.

Definition:

```
: UNKNOWNS 0 BEGIN
     OVER LROT FIND-RULE
  WHILE
     DUP [RULE-EVAL]
     OVER LT-STATE C@ IF
        2DROP EXIT
     ENDIF
     1+ DUP R#LU @ = IF
        2DROP EXIT
     ENDIF
  REPEAT DROP ;
```

Example of Use:

```
8 LT-STATE C@ S-F
```

This would leave a Boolean flag on the stack. The flag will be true if the state of rule eight was true.

Algorithm: If the flag is one, leave a true; otherwise leave a false.

Suggested Extensions: None.

Definition:

```
: S-F 1 = ;
```

$-EVAL (N - F)

Leave the state of a string in the expert system.

Stack on Entry: (N) – The number of the string to evaluate.

Stack on Exit: (F) – A Boolean flag representing the validity of the string.

Example of Use:

```
8 $-EVAL
```

This would leave a flag on the stack, true if string eight is true, false otherwise.

Algorithm: If the state of the string is known, return it and exit. Otherwise, let UNKOWN$ try to set the state of the rule. If it could not ask the user as a last resort, convert the state to a flag and leave it on the stack.

Suggested Extensions: None.

Definition:

```
: $-EVAL
  DUP LT-STATE C@ ?DUP IF
    SWAP DROP S-F EXIT
  ENDIF
  DUP UNKNOWNS
  DUP LT-STATE C@ 0= IF
    DUP ASK-USER
  ENDIF
  LT-STATE C@ S-F ;
```

COND-EVAL (A - F)

Determine the truth of a condition.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: (F) – A Boolean flag representing the truth of the condition.

Example of Use: See words defined below.

Algorithm: If the condition is a string, use $–EVAL. If it is a Forth word, execute it. The Forth word should leave a flag on the stack.

Suggested Extensions: Check to make sure that any Forth word that does execute leaves a flag on the stack.

Definition:

```
: COND-EVAL
  DUP C@ CASE
     1 =OF 1+ @ $-EVAL END-OF
     2 =OF 1+ @ EXECUTE END-OF
  ENDCASE ;
```

+WHEN (A –)

Evaluate a WHEN condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Since a WHEN must start every rule, store the result of the condition evaluation performed by COND–EVAL in the Boolean array for the level being analyzed.

Suggested Extensions: None.

Definition:

```
0 VARIABLE LEVEL

: +WHEN
   COND-EVAL LEVEL C@ R [] C! ;
```

-WHEN (A -)

Evaluate a WHEN NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Since a WHEN must start every rule, store the logical negation of the result of the condition evaluation performed by COND–EVAL in the Boolean array for the level being analyzed.

Suggested Extensions: None.

Definition:

```
: -WHEN
  COND-EVAL NOT LEVEL C@ R [] C! ;
```

+AND (A -)

Evaluate an AND condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An AND requires the current Boolean result for the level it is working on to be true before it attempts to evaluate the condition. If the level flag is false, the truth of this condition cannot change the flag setting. If the level flag is true, the condition will be evaluated using COND–EVAL and the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: +AND
  LEVEL C@ R [] C@ IF
    COND-EVAL LEVEL C@ R [] C!
```

```
  ELSE
    DROP
  ENDIF ;
```

2-AND (A -)

Evaluate an AND NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An AND requires the current Boolean result for the level it is working on to be true before it attempts to evaluate the condition. If the level flag is false, the truth of this condition cannot change the flag setting. If the level flag is true, the condition will be evaluated using COND-EVAL and the logical negation of the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: -AND
  LEVEL C@ R [] C@ IF
    COND-EVAL NOT LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

+OR (A -)

Evaluate an OR condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An OR requires the current Boolean result for the level it is working on to be false before it attempts to evaluate the condition. If the level flag is true, the truth of this condition cannot change the flag setting. If the level flag

is false, the condition will be evaluated using COND–EVAL and the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: +OR
  LEVEL C@ R [] C@ 0= IF
    COND-EVAL LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

–OR (A –)

Evaluate an OR NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An OR requires the current Boolean result for the level it is working on to be false before it attempts to evaluate the condition. If the level flag is true, the truth of this condition cannot change the flag setting. If the level flag is false, the condition will be evaluated using COND–EVAL and the logical negation of the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: –OR
  LEVEL C@ R [] C@ 0= IF
    COND–EVAL NOT LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

C/RULE–EVAL (N – F)

Evaluate the conditional statements of a rule.

Stack on Entry: (N) – The rule number to be evaluated.

Stack on Exit: (F) – The Boolean result of evaluating the conditions.

Example of Use: See words defined below.

Algorithm: This word loops through all the conditions for the rule it finds on the stack. First, it gets the address of the rule from the RULES array. Nine is added to this address to point to the start of the conditions. Each is evaluated until the end of the list is reached. The Boolean flag for the level is removed from the R [] array and left on the stack.

Suggested Extensions: None.

Definition:

```
: C/RULE-EVAL
  RULES @ 9 + BEGIN
    DUP C@ OVER 3 + SWAP CASE
      3 =OF +WHEN END-OF
      4 =OF -WHEN END-OF
      5 =OF +AND END-OF
      6 =OF -AND END-OF
      7 =OF +OR END-OF
      8 =OF -OR END-OF
    ENDCASE
    6 + DUP C@ 0=
  UNTIL DROP
  LEVEL C@ R [] C@ ;
```

DO-EXPLAIN (–)

Cause the expert system rule interpreter to explain its conclusions.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DO-EXPLAIN
```

After this word is executed, whenever the rule interpreter reaches a conclusion, it will use the EXPLAIN clause to explain its actions.

Algorithm: Set the variable EXPLAIN? to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE EXPLAIN?

: DO-EXPLAIN EXPLAIN? C1SET ;
```

DON'T-EXPLAIN (-)

Cause the expert system rule interpreter to suppress explanations.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DON'T-EXPLAIN
```

After this word is executed, no EXPLAIN clauses will be invoked.

Algorithm: Set the variable EXPLAIN? to false.

Suggested Extensions: None.

Definition:

```
: DON'T-EXPLAIN EXPLAIN? C0SET ;
```

.EXPLAIN (A -)

Execute the EXPLAIN clause of a rule definition.

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

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If EXPLAIN? is false, drop the address and exit the word. Otherwise, if the EXPLAIN clause is a string print it out. If the EXPLAIN clause is a word, execute it. If there is no explain clause, drop the address and exit.

Suggested Extensions: None.

Definition:

```
: .EXPLAIN EXPLAIN? C@ IF
    DUP 6 + C@ DUP 1 = IF
      DROP 7 + @ CR ." Because " LTEXT $?
    ELSE
      2 = IF
        7 + @ EXECUTE
      ELSE
        DROP
      ENDIF
    ENDIF
  ELSE
    DROP
  ENDIF ;
```

DO-MON (-)

Cause the expert system rule interpreter to print out all hypotheses reached.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

DO-MON

After this word is executed, all hypotheses set will be printed as they are set.

Algorithm: Set the variable MON? to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE MON?

: DO-MON MON? C1SET ;
```

DON'T-MON (-)

Cause the expert system rule interpreter only to print the final conclusion reached.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DON'T-MON
```

After this word is executed, only the final hypothesis set will be printed as it is set.

Algorithm: Set the variable MON? to false.

Suggested Extensions: None.

Definition:

```
: DON'T-MON MON? C0SET ;
```

.MON (A -)

Print the result of an HYPOTH statement.

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

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the LEVEL is one, or the variable MON? is true, this word will print the value of the hypothesis being set. Only string hypotheses are printed out.

Suggested Extensions: None.

Definition:

```
: .MON MON? C@ LEVEL C@ 1 = OR IF
    DUP C@ 9 = IF
```

```
      CR ." True => "
    ELSE
      CR ." False => "
    ENDIF
    DUP 3 + C@ 1 = IF
      4 + @ LTEXT $?
    ELSE
      DROP
    ENDIF
  ELSE
    DROP
  ENDIF ;
```

TELL? (– F)

Should a Forth word hypothesis print out what it is doing?

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the word should print out what it is doing.

Example of Use:

```
: FEVER P-TEMP C@ 100 MIN P-TEMP C! TELL? IF
   ." Patient must have a temperature of at least 100 degrees "
  ENDIF ;
```

FEVER is a conclusion in an expert system. It uses ?TELL to determine whether or not it should report what action it is taking.

Algorithm: If the LEVEL is one or the variable MON? is true, return a true flag.

Suggested Extensions: None.

Definition:

```
: TELL? MON? C@ LEVEL C@ 1 = OR ;
```

$APPLY-RULE (A –)

Set the string hypothesis of a rule.

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

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Call .MON to print out the setting of the hypothesis. A nine signifies a set hypothesis true, a ten indicates a set hypothesis false. The proper value is stored in the LT–STATE variable.

Suggested Extensions: None.

Definition:

```
: SAPPLY-RULE
   DUP .MON DUP C@ 9 = F-S
   SWAP 4 + @ LT-STATE C! ;
```

WORD–APPLY–RULE (A –)

Set the word hypothesis of a rule.

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

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Execute the word stored in the hypothesis field of a rule.

Suggested Extensions: None.

Definition:

```
: WORD-APPLY-RULE
   4 + @ EXECUTE ;
```

RULE–EVAL (N –)

Evaluate a rule.

Stack on Entry: (N) – The number of the rule to evaluate.

Stack on Exit: Empty.

Example of Use:

```
8 RULE-EVAL
```

This would evaluate rule eight and set its hypothesis if the conditions of the rule we're found to be true.

Algorithm: Increment LEVEL for the new rule being evaluated. Use C/RULE-EVAL to determine the truth of the conditions for the rule. If the conditions evaluate true, set the hypothesis of the rule. This is done by executing either $APPLY-RULE or WORD-APPLY-RULE. Execute .EXPLAIN if the rule is applied. Decrement LEVEL when the word is exited. LEVEL is used since RULE-EVAL can be called recursivly through UNKNOWNS.

Suggested Extensions: None.

Definition:

```
: RULE-EVAL
   1 LEVEL C+!
   DUP RULES @ SWAP C/RULE-EVAL IF
      DUP DUP 3 + C@ CASE
         1 =OF $APPLY-RULE END-OF
         2 =OF WORD-APPLY-RULE END-OF
      ENDCASE .EXPLAIN
   ELSE
     DROP
  ENDIF -1 LEVEL C+! ;
```

(Store the address of RULE-EVAL in /RULE-EVAL/ to handle a forward reference.)

```
' RULE-EVAL /RULE-EVAL/ !
```

RESET-SYSTEM (-)

Clear all the variables used by the expert system.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

RESET-SYSTEM

Executing RESET-SYSTEM will allow the rule interpreter to start over fresh.

Algorithm: The number of rules and strings used is set to zero. The RULES array and LT-STATE arrays are cleared.

Suggested Extensions: None.

Definition:

```
: RESET-SYSTEM
   R#LU 0SET LT#LU 0SET
   0 RULES 200 ERASE
   0 LT-STATE 100 ERASE ;
```

RT? (A – F)

Is a rule hypothesis known ?

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

Stack on Exit: (F) – A Boolean flag, true if the state of the hypothesis of a rule is known.

Example of Use:

```
8 RT?
```

This will tell whether or not a rule's hypothesis has a known state.

Algorithm: Find the state of strings from LT-STATE. Execute words and use the flag they return.

Suggested Extensions: None.

Definition:

```
: RT?
3 + DUP C@ CASE
   1 =OF 1+ @ LT-STATE C@ NOT NOT END-OF
   2 =OF 1+ @ EXECUTE END-OF
ENDCASE ;
```

APPLY (-)

Apply the rules of an expert system.

Stack on Entry: None.

Stack on Exit: None.

Example of Use:

APPLY

This will start the execution of the rule interpreter.

Algorithm: Clear the state array. Execute the word found in /INIT/ that can be used by the expert system to initialize any of its variables. Start by trying to find a conclusion for each rule. Stop when an hypothesis is reached on level one. If all the rules are executed and no conclusion is reached, print a message.

Suggested Extensions: None.

Definition:

```
' DUMMY VARIABLE /INIT/

: APPLY
   0 LT-STATE 100 ERASE /INIT/ @ EXECUTE
   0 R [] 32 ERASE
   0 BEGIN
      DUP RULES @ RT? NOT IF
         DUP RULE-EVAL
      ENDIF
      1 R [] C@ IF DROP EXIT ENDIF
      1 + DUP R#LU @ = Y
   UNTIL DROP
   CR ." No conclusions reached " ;
```

A SAMPLE EXPERT SYSTEM

The expert system presented below will advise a baseball manager when to attempt a sacrifice bunt. It is simple and not that complete, but tries to show an example of each feature of the words presented in this chapter.

In our expert system, the order of the rules is crucial. Rules that reach intermediate conclusion should be placed at the end of the list of rules. If they

are not, the rule interpreter will terminate with a true, but not so interesting, conclusion. Of course, this is one area in which the words in this chapter could be expanded.

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

0 CVARIABLE OUTS

: GET-OUTS
   CR ." How many outs are there? "
   #IN OUTS C! -1 ;

: NONE-OUT OUTS C@ 0= ;
: 1OUT OUTS C@ 1 = ;
: 2OUT OUTS C@ 2 = ;

0 CVARIABLE 1B
0 CVARIABLE 2B
0 CVARIABLE 3B

: 1B?
   CR ." Is there a runner on first ? "
   (Y/N) 1B C! -1 ;

: 1B?
   CR ." Is there a runner on first ? "
   (Y/N) 1B C! -1 ;

: 2B?
   CR ." Is there a runner on second ? "
   (Y/N) 2B C! -1 ;

: 3B?
   CR ." Is there a runner on third ? "
   (Y/N) 3B C! -1 ;

: 1B-HAS-RUNNER 1B C@ NOT NOT ;
: 2B-HAS-RUNNER 2B C@ NOT NOT ;
: 3B-HAS-RUNNER 3B C@ NOT NOT ;

0 CVARIABLE OUR-SCORE
: OUR-SCORE-GET
   CR ." How many runs do we have ? "
   #IN OUR-SCORE C! -1 ;
```

```
0 CVARIABLE THEIR-SCORE
: THEIR-SCORE-GET
   CR ." How many runs do they have ? "
   #IN THEIR-SCORE C! -1 ;


0 CVARIABLE INNING
: INNING-GET
   CR ." What inning is this ? "
   #IN INNING C! 0 ;
```

(INNING-GET will always be false. This allows this rule to input information. /INPUT/ could also be used.

```
RULE
   WHEN GET-OUTS & 1B? & 2B? & 3B?
   & OUR-SCORE-GET & THEIR-SCORE-GET
   & INNING-GET
   HYPOTH | JUNK |
   END-RULE

: AHEAD-BY-MORE-THAN-1
   OUR-SCORE C@ THEIR-SCORE C@ - 1 > ;

RULE
   WHEN | WE ARE HOME |
      & AHEAD-BY-MORE-THAN-1
   HYPOTH -- | BUNT |
END-RULE

: INNING-LATER-THEN-7TH
   INNING C@ 7 > ;

RULE
   WHEN -- | WE ARE HOME |
      & AHEAD-BY-MORE-THAN-1
      & -- INNING-LATER-THEN-7TH
   HYPOTH -- | BUNT |
   EXPLAIN | TOO EARLY TO BUNT |
END-RULE

RULE
   WHEN | BASES ARE EMPTY |
   HYPOTH -- | BUNT |
   EXPLAIN | NO RUNNERS TO MOVE UP |
```

```
END-RULE

RULE
  WHEN -- | WE ARE HOME |
    & AHEAD-BY-MORE-THAN-1
    & -- INNING-LATER-THEN-7TH
  HYPOTH -- | BUNT |
  EXPLAIN | TOO EARLY TO BUNT |
END-RULE

RULE
  WHEN | BASES ARE EMPTY |
  HYPOTH -- | BUNT |
  EXPLAIN | NO RUNNERS TO MOVE UP |
END-RULE

RULE
  WHEN 2OUT
  HYPOTH -- | BUNT |
  EXPLAIN | NO ONE TO MOVE OVER |
END-RULE

: UP-BY-ONE
  OUR-SCORE C@ THEIR-SCORE C@ - 1 = ;

: DOWN-BY-ONE
  THEIR-SCORE C@ OUR-SCORE C@ - 1 = ;

: TIED
  OUR-SCORE C@ THEIR-SCORE C@ = ;

RULE
  WHEN | FAST RUNNER ON FIRST |
    |  | AVERAGE RUNNER ON FIRST |
    & | ONLY 1B OCCUPIED |
    & | WE ARE HOME |
    & DOWN-BY-ONE
    & INNING-LATER-THEN-7TH
  HYPOTH | BUNT |
END-RULE

RULE
  WHEN | ONLY 1B OCCUPIED |
    & | BATTER IS A GOOD BUNTER |
    & | WE ARE HOME |
    & DOWN-BY-ONE
```

```
    & INNING-LATER-THEN-7TH
  HYPOTH | BUNT |
END-RULE

RULE
  WHEN | ONLY 1B OCCUPIED |
    & | BATTER IS A GOOD BUNTER |
    & -- | WE ARE HOME |
    & NONE-OUT
    & INNING-LATER-THEN-7TH
    & TIED
    |  DOWN-BY-ONE
  HYPOTH | BUNT |
  EXPLAIN | CONSERVATIVE ON THE ROAD |
END-RULE

:T -1;
```

(This is used to prevent secondary conclusions as conclusions of the expert system.)

```
RULE WHEN T HYPOTH -- | BUNT | END-RULE

RULE
  WHEN 1B-HAS-RUNNER
    |  2B-HAS-RUNNER
    |  3B-HAS-RUNNER
  HYPOTH -- | BASES ARE EMPTY |
END-RULE

RULE
  WHEN 1B-HAS-RUNNER
    & -- 2B-HAS-RUNNER
    & -- 3B-HAS-RUNNER
  HYPOTH | ONLY 1B OCCUPIED |
END-RULE

RULE
  WHEN -- 1B-HAS-RUNNER
    & -- 2B-HAS-RUNNER
    & -- 3B-HAS-RUNNER
  HYPOTH | BASES ARE EMPTY |
END-RULE
```

13

Debugging Programs

Words Defined in This Chapter:

.S	Nondestructive stack print.
X	Simple debugging word.
ONDEBUG	Turn on complex debugging.
OFFDEBUG	Turn off complex debugging.
?NUMBER	Try to convert a string to a number.
COM-EX	Execute the debug commands.
Y	Debugging word.

The library routines presented in this book will save you a lot of debugging time. After all, the debugging has, hopefully, already been completed for the library words. But as you sit down to use the library routines and write your own programs, you will have to face debugging.

The single most powerful tool you have in debugging your Forth programs is your own programming style. If you keep your words short and self-contained, and you completely test each word as you write it, debugging should be a simple proposition. Forth makes the debugging of individual words simple and straightforward. You can place the arguments your word needs on the stack, store the proper values in the variables it uses, and then just execute your word. It really can be that simple.

The Three Rules:

1. Keep words short. The shorter a word is, the easier it is to debug. There is less that can go wrong. Try not to have words that stretch across more than a

single screen; ideally, they should be even shorter. Try to make each word perform a single logical function.

2. Keep words self-contained. Each word should be as much of a stand alone unit as possible. This means using the stack instead of variables. Many times it is proper to use a variable, but if your words are large and use lots of variables, it is a good indication that they can be broken down into smaller, simpler words.

3. (AND MOST IMPORTANT!!!) Test each word thoroughly as you write it! This is by far the most important principle presented in this chapter. Forth makes it easy for you to test your word. If you spend the time checking your word when you write it, you'll save a tremendous amount of debugging time. There is nothing worse than staring at a nonfunctioning word that looks perfectly correct, only to find out it is and the reason it's not functioning is three levels down. A thoroughly debugged word is a great asset; it can become an important tool for you. A word that has bugs is a time bomb waiting to sabotage your program.

When testing a word, pay particular attention to the upper and lower bounds of its arguments. If you have a string word that can handle strings up to 255 characters in length, check a few 255-character strings. Boundary areas are often a source of bugs.

A Few Helpful Words

When a really obstinate bug is holding you up, these words may be helpful. .S is a nondestructive stack print. It allows you to easily examine the stack without disturbing it. The word X can be placed throughout a section of code you are debugging to get a snapshot of the stack.

The words Y, ONDEBUG, and OFFDEBUG are powerful debugging words. Normally you can only print out the stack, dump memory, and examine the values of variables before and after you execute a word. With this set of words you can do all these things while a word is executing. Placing Y in a word can be thought of as setting a breakpoint. If debugging is trned on when a Y is encountered, the prompt DEBUG> will be printed on the screen. You can execute any word at this prompt. You can use .S to dump the stack, or even change the stack if you desire. Typing the word GO will continue execution. The word SKIP will continue execution and turn off all further breakpoints. To summarize:

Debug Commands:

ONDEBUG – Enable debugging.
OFFDEBUG – Disable debugging.

Y – A breakpoint; place it inside a word definition.

DEBUG> GO – Continue execution.
DEBUG> SKIP – Continue execution, skip all further breakpoints.

Suggested Extensions: The debugging commands are insufficient if you are debugging a word that produces screen output. They could be expanded to save the screen and restore it when a breakpoint is encountered.

.S (–)

Nondestructive stack print.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
1 2 3 .S

3
2
1
<BOTTOM OF STACK>
```

This is what .S would print if the stack was empty before the above line was typed in.

Algorithm: Determine first if the stack has any entries. If it does not, print the stack empty message and exit. If it is not empty, loop through each value on the stack and print it. The word PICK will not destroy the original stack entry.

Suggested Extensions: None.

Definition:

```
: .S CR DEPTH ?DUP IF
     1+ 1 DO I PICK . CR LOOP
     ." <BOTTOM OF STACK>" CR
  ELSE
     ." <STACK EMPTY>" CR
  ENDIF ;
```

X (-)

Dump the stack and pause.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: TEST OVER DUP + X + . ;
```

When TEST is executed, the stack will be dumped after the first addition. Execution will be stopped until a key is hit so the programmer can examine the stack.

Algorithm: Use .S to dump the stack, then wait on a key.

Suggested Extensions: None.

Definition:

```
: X CR .S KEY DROP ;
```

ONDEBUG (-)

Turn on debugging.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
ONDEBUG
```

When this word is executed, all breakpoints encountered will allow the user to enter words at the DEBUG> prompt.

Algorithm: Set the variable /SKIP/ to false.

Suggested Extensions: None.

Definition:

```
1 CVARIABLE /SKIP/

: ONDEBUG /SKIP/ C0SET ;
```

OFFDEBUG (–)

Turn off debugging.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
OFFDEBUG
```

When this word is executed, execution will continue normally past all breakpoints.

Algorithm: Set the variable /SKIP/ to true.

Suggested Extensions: None.

Definition:

```
: OFFDEBUG /SKIP/ C1SET ;
```

?NUMBER (A – (N or D) F)

Attempt to convert a string to a number.

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

Stack on Exit: (N or D) – The number if one is found. Double length if a comma is in the number.
(F) – A Boolean flag, true if a number is found.

Example of Use: See Y.

Algorithm: First, check for a leading negative sign. If one is found, increment the pointer past it. Then; attempt to convert the string to a number. If the con-

version stops on a comma, assume a double-length number and continue the conversion. In either case, if the conversion stops on a zero or blank, it is considered successful. Leave the number (negative, if appropriate) and a true flag on the stack. If conversion is unsuccessful, drop the number and leave a false flag.

Suggested Extensions: If floating-point numbers are being used, extend this word to allow them as input.

Definition:

```
45 CCONSTANT -KEY
44 CCONSTANT ,KEY

: ?NUMBER
  DUP 1+ C@ -KEY = IF
  1+ -1 >R
  ELSE
  0 >R
  ENDIF
  0. ROT >BINARY DUP C@ ,KEY = IF
     >BINARY C@ DUP BL = SWAP 0= OR IF
       R> IF DNEGATE ENDIF -1
     ELSE
       R> DROP 2DROP 0
     ENDIF
  ELSE
     SWAP DROP C@ DUP BL = SWAP 0= OR IF
       R> IF NEGATE ENDIF -1
     ELSE
       R> 2DROP 0
     ENDIF
  ENDIF ; ->
```

COM-EX (A -)

Check for a debug command. Execute a word if one is not found.

Stack on Entry: (A) - The address of a word.

Stack on Exit: Empty.

Example of Use: See Y.

Algorithm: COM-EX checks for the debug commands GO and SKIP. They

are defined as dummy words. If their address is the one on the stack, process them as commands, not as words to be executed. If the address on the stack is not SKIP or GO, execute the word at that address.

Suggested Extensions: New commands can be added to this debug package in this word. Define a dummy word with the name of your new command and then place an IF statement in COM−EX to handle it.

Definition:

```
: GO ;
: SKIP ;

: COM-EX
    [ ' GO ] LITERAL OVER = IF
      DROP R> DROP EXIT
    ELSE
      [ ' SKIP ] LITERAL OVER = IF
        DROP R> DROP OFFDEBUG EXIT
      ELSE
        EXECUTE
      ENDIF
    ENDIF ;
```

Y (-)

Set a breakpoint in a word.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: TEST 10 0 DO
    I EVAL Y
  LOOP ;
```

The word TEST would have a breakpoint after EVAL. Everytime that part of the word is reached and debugging is turned on, Y becomes active.

Algorithm: First, check to see if debugging is on. If it is not, exit the word. Print out the debug prompt and obtain a line of input. Separate the words on the line by using BL WORD. If the first word can be found in the dictionary, pass it to COM−EX to handle. After executing it, determine if there are any words left on the line. If there are not, leave a false flag on the stack so a new

line of input will be obtained. If FIND could not locate the word in the dictionary, try to convert it to a number. If it is converted to a number, leave it on the stack and check for the end of line condition. If the word is not found in the dictionary and cannot be converted to a number, print out an error message, and leave a flag on the stack that will cause a new line of input to be obtained. The only exit from this word once the debug loop has begun is from COM-EX.

Suggested Extensions: None.

Definition:

```
0 VARIABLE H>IN

: Y /SKIP/ C@ IF EXIT ENDIF BEGIN
   >IN 0SET BLK 0SET CR
   ." DEBUG> " QUERY BEGIN
      >IN @ H>IN ! FIND ?DUP IF
         COM-EX
         >IN @ BL WORD C@ SWAP >IN !
      ELSE
         H>IN @ >IN ! BL WORD DUP C@ IF
            DUP >R ?NUMBER IF
               R> DROP
               >IN @ BL WORD C@ SWAP >IN !
            ELSE
               R> TYPE2 ." Eh? " CR 0
            ENDIF
         ELSE
            DROP 0
         ENDIF
      ENDIF
   NOT UNTIL
0 UNTIL ;
```

Appendix A

Stacks for Beginners

The word stack may conjure up visions of a horribly complex entity for the uninitiated. In reality, stacks are among the simplest and most straightforward data structures found in computer science. A stack is a last in, first out (or LIFO) data structure. The most common real-world analogy is a plates dispenser in a cafeteria. Plates are placed in and removed from the top of the container (or stack) that holds them.

The stack is a central feature of Forth, and we can use Forth to learn about stacks. If you type in the word, .S (found in Chapter 13), Forth can help you learn about stacks. .S prints what Forth is holding on its stack. When you type a number in Forth, it places that number on the stack. For example:

```
ATILA OK 23 RETURN
```

If you type a 23 and hit return, Forth will place a 23 on its stack. This is just like writing a 23 on a plate and placing it in the cafeteria dispenser. If a person came and removed the plate from the dispenser, he would find a 23 written on it. In Forth the word . ("dot") does this for us.

```
ATILA OK . [RETURN]

23 ATILA OK
```

Dot removes the top number from the stack and prints it on the display.

Now let's see what happens with more than a single plate, or number. We'll write 67 on a plate and place it in the dispenser. Then we'll deface a plate with a 91 and place it in the dispenser. If someone came along and removed the top plate, what number would he find on it? Let's use Forth to figure it out.

```
ATILA OK 67 [RETURN]
(Put the plate numbered 67 in the dispenser)

ATILA OK 91 [RETURN]
(Put the plate numbered 91 in the dispenser)

ATILA OK . [RETURN]
(Dot will tell us what the top plate has on it)

91 ATILA OK 91
```

is on the top plate. What's on the next plate? Let's use Forth again:

```
ATILA OK . [RETURN]
67 ATILA OK
```

67 is, of course, the next plate. What if we ask "dot" to look at the next plate?

```
ATILA OK . [RETURN]
1234 STACK UNDERFLOW
```

Stack underflow is "dot's" way of telling us there are no more numbers on the stack. .S can be used to look at the stack without changing its contents. Here's how:

```
ATILA OK 67 91 .S [RETURN]
(Everything on one line!)

91
67
<BOTTOM OF STACK>

ATILA OK . . [RETURN]
(Now print them)

91  67  ATILA OK
```

This is just how the plates would look also.

Forth includes a large number of words to manipulate words on the stack. DUP (duplicate) is among the most basic. It makes a copy of the top number on the stack, like so:

```
ATILA OK 4 .S [RETURN]

4
```

```
<BOTTOM OF STACK>
ATILA OK DUP .S [RETURN]

4
4
<BOTTOM OF STACK>
```

Now we have two fours. Let's print them:

```
ATILA OK . . .S [RETURN]

4  4

<STACK EMPTY>

ATILA OK
```

After we printed what was on the stack, there was nothing left, so .S told us the stack was empty. .S will enable you to play with the stack and learn by getting hands-on experience. Here are some other words you might want to experiment with:

SWAP	Swap the top two numbers on the stack.
DROP	Remove the top number from the stack.
OVER	Move a copy of the second number on the stack to the top.
DEPTH	Leave the number of numbers on the stack on the stack.
ROT	Rotate the top three numbers on the stack.

HAVE FUN!

Appendix B

Extra Atila Words

Any time you enter a word from this book and you find that your version of Forth doesn't know it, this is the place to look. A few of the Atila words can be written in Forth, and these should present you with no problems. Most will also have a corresponding word in your version of Forth. Some, however, really require assembler. For those we'll present the assembler code from MASM, the standard IBM assembler, and how they would be written in the assembler presented in Chapter 5. You will have to refer to the documentation provided with your version of Forth to determine how to implement them.

ENDIF	: ENDIF COMPILE THEN ; IMMEDIATE
<BUILDS	: <BUILDS COMPILE CREATE ;
LROT	: LROT SWAP ROT SWAP ;
HOME	: HOME CLS ;
(Clear the Screen)	
ATILA	: ATILA FORTH ;

VTAB

```
CODE VTAB BH 0 # MOV
   AH 3 # MOV 16 # INT AX POP DH AL MOV
   AH 2 # MOV 16 # INT
NEXT
```

```
vtab:

dw      $+2
mov     bh,0
mov     ah,3
int     010h
pop     ax
mov     dh,al
mov     ah,2
int     010h
jmp     NEXT
```

HTAB

```
CODE HTAB BH 0 # MOV
   AH 3 # MOV 16 # INT AX POP DL AL MOV
   AH 2 # MOV 16 # INR
NEXT

htab:

dw      $+2
mov     bh,0
mov     ah,3
int     010h
pop     ax
mov     dl,al
mov     ah,2
int     010h
jmp     NEXT
```

Appendix C

Stack Notation

The following stack notation is used in this book:

(Before Execution/After Execution)
(Least Accessible ... Most Accessible/Least Accessible ... Most Accessible)
N/A signed 16-bit number
UN/An unsigned 16-bit number.
D/A signed 32-bit number.
UD/An unsigned 32-bit number.
A/A 16-bit address F/A Boolean Flag.
R/A 32-bit real number.
Multiple instances of the same data type are numbered sequentially.
An example:
-TEXT (A1 N1 A2/N2)
-TEXT expects an address, then an integer, then an address as arguments. It returns an integer result.

INDEX

Boldface indicates a Forth word that is defined in the text.

Library of Forth Routines and Utilities

ORDER FORM

TERRY BROTHERS SOFTWARE

CREATOR OF ATILA - THE FORTH FOR THE IBM PC™

42 Princeton Arms South
Crenbury, New Jersey 08512
(609) 426-0977

THE LIBRARY OF FORTH ROUTINES	UNIT PRICE	X # OF COPIES	= UNIT TOTAL
ATILA IBM SCREEN FORMAT	$19.95		
MS-DOS TEXT FILE FORMAT	$19.95		
ATILA APPLE SCREEN FORMAT	$19.95		
ATILA COMMODORE 64/128 SCREEN FORMAT	$19.95		
ATILA MacINTOSH™ SCREEN FORMAT	$19.95		
ATILA ATARI 520 ST SCREEN FORMAT*	$19.95		
ATILA AMIGA™ SCREEN FORMAT*	$19.95		
ATILA...THE FORTH FOR YOUR COMPUTER!			
IBM PC VERSION (AND 100% COMPATABLES)	$69.95		
APPLE II VERSION	$69.95		
COMMODORE 64/128 VERSION	$69.95		
APPLE MacINTOSH VERSION	$69.95		
ATARI 520 ST VERSION*	$69.95		
AMIGA VERSION*	$69.95		
		SUB-TOTAL	
		NJ RESIDENTS ADD 6% SALES TAX	
		SHIPPING AND HANDLING	$3.00
		TOTAL	

*ATARI AND AMIGA VERSIONS AVALIABLE 7/1/86

CHECK___ MONEY ORDER___ VISA___ MASTERCARD___

CARD #_____ EXP DATE _____

NAME _____

ADDRESS _____

CITY_____ STATE_____ ZIP_____

ON DISK!

The complete source for every word defined in this book is available on diskette: already typed in . . . already debugged!

COMPUTERS • Z5841 • $22.95
CANADA • $31.95

THE READY-TO-USE TIME-SAVING CODE FOR FORTH PROGRAMMERS

The LIBRARY OF FORTH ROUTINES AND UTILITIES is the only comprehensive collection of professional-quality computer code for Forth, the programming language of the future. Full of creative, time-saving routines, this book offers programs that can be put to use in almost any Forth application, including expert systems and natural language interfaces. Among the hundreds of applications included are:

- A unique collection of Forth routines written for the IBM PC or any 8088 computer system
- Hundreds of efficient routines that are easily tailored to applications in expert systems and artificial intelligence.
- Completely tested and debugged code ready to splice into any version of Forth
- Assembly language routines for the 8088 and the math co-processor
- A highly flexible mini-database management package
- A sentence parser that can dialogue with the user
- And much more

LIBRARY OF FORTH ROUTINES AND UTILITIES

FULLY ILLUSTRATED