Ultimate CASE Statement#

by Wil Baden, Costa Mesa, California

appeared in VD February 1987

Many citizens of the Forth community have lamented the lack of a **case** statement in standard Forth language specifications. Since the first rule of Forth programming is, "If you don't like it, change it," there have been many proposals, and Forth Dimensions even held "The Great CASE Contest in Volume II". Although the winning entry of that contest, submitted by Charles Eaker, has been widely implemented and even offered as part of many vendors' systems, the flood of proposals has not ceased. There have been many articles and letters on the subject in Forth Dimensions.

All proposals to date have had problems. Portability is one. Another is that they all have been too specialized and restricted in their area of application. Generalization is accomplished by designing another special case of **case**.

Strictly speaking, a **case** statement is unnecessary. It is " syntactic sugar" to make a program easier to write, read and understand. It is so helpful in doing this that it is a standard feature of all other modern programming languages.

Figure 1a

```
craps (n)
int n;
{   if (n==7) printf ("You win");
    else if (n == 11) printf ("You win");
    else if (n == 2)  printf ("You lose");
    else if (n == 3)  printf ("You lose");
    else if (n == 12) printf ("You lose");
    else printf ("%d is your point",n);
}
```

Figure 1a is a rather futile program written in C to illustrate a common pattern of logical decisions in many programs.("==" is "equal to" for comparing two things, to distinguish it from "=" for assignment as in Fortran or Basic.) An equivalent Forth version would look something like Figure 1b.

Figure 1b

```
: CRAPS ( n -- )
  DUP 7 =
  IF DROP ." You win"
  ELSE DUP 11 =
    IF DROP ." You win"
    ELSE DUP 2 =
      IF DROP ." You lose"
      ELSE DUP 3 =
        IF DROP ." You lose"
        ELSE DUP 12 =
          IF DROP ." You win"
          ELSE . ." is your point" THEN
          THEN THEN THEN THEN THEN ;
```

Most people will agree that Figure 1a would be better written as in Figure 2a. An even better way is found in some dialects of C, illustrated by Figure 2a. In this extension, following syntax from Pascal, values separated by "," indicate a set of values, and values separated by ".." indicate a range.

Figure 2a

```
craps (n)
int (n);
{
    switch(n) {
    case  7: printf("You win"); break;
    case 11: printf("You win"); break;
    case  2: printf("You lose"); break;
    case  3: printf("You lose"); break;
    case 12: printf("You lose"); break;
    default: printf("%d is your point",n);
    }
}
```

Some Forth proposals have one definition for individual values and another definition for a range of values. There would have to be another definition for a set of values. No earlier Forth proposal that I know of allows sets and ranges together, as in:

```
case 2..3, 12:
```

What is proposed here is a single **case** statement for Forth which will include all these variations, and many more, that can be implemented in fig-FORTH, Forth-79, Forth-83 and any other Forth.

Figure 2b

```
craps(n)
int n;
{  switch(n)  {
   case 7, 11:    printf("You win"); break;
   case 2..3, 12: printf("You lose"); break;
   default:       printf("%d is you point",n);
   }
}
```

Figure 3

```
: CASE  DUP ;

: CRAPS ( n -- )
   CASE   7 = IF DROP ." You win"  EXIT THEN
   CASE  11 = IF DROP ." You win"  EXIT THEN
   CASE   2 = IF DROP ." You lose" EXIT THEN
   CASE   3 = IF DROP ." You lose" EXIT THEN
   CASE  12 = IF DROP ." You lose" EXIT THEN
   . . is your point" ;
```

Figure 2a would look as shown in Figure 3. Let's add two more spoons of syntactic sugar, as in Figure 4. As has been noted elsewhere, too much syntactic sugar causes semantic diabetes. Our **case** is sweet enough. Figure 5 is an example to show some of the possibilities.

Figure 3

```
: OF  ( n flag -- ) [COMPILE] IF  COMPILE DROP ; IMMEDIATE
: =OR ( n flag n -- n flag ) 2 PICK = DROP ;

: CRAPS ( n -- )
  CASE  7 =  11 =OR             OF ." You win"  EXIT THEN
  CASE 2 3 BETWEEN  12 =OR      OF ." You lose" EXIT THEN
  . ." is your point" ;
```

Figure 4

```
: WHATEVER  ( n -- )
  CASE 0=                       OF ." zero" EXIT THEN
  CASE 0<                       OF ." Negative" EXIT THEN
  CASE DUP 1- AND 0=            OF ." Power of 2" EXIT THEN
  CASE ASCII 0  ASCII 9 BETWEEN OF ." Digit" EXIT THEN
  CASE ASCII ,  ASCII / BETWEEN
       ASCII : =OR              OF ." Punctuation ,-./: " EXIT THEN
  DROP ." Whatever" ;
```

Now for a real life example. Figure 6 is a recension of a word in John James "Universal Text File Reader" (Forth Dimensions VII/3). One of my favorite examples is "Thirty days hath September, April, June and November ..." See Figure 7. If **NUMBER** in your system is vectored, you may want to replace it in some applications with a version that selects the numerical radix according to the first character. Figure 8 implements a convention used on Motorola systems (e.g., 68000). Laxen's **CLASSIFY** example (FD VII/I) can be written without redundant classes with no additional definitions, as in Figure 9.

Figure 6

```
: ?OUT ( c -- )   127 AND
   CASE  0=  13 ( return ) 0O
     OF  ?NEW-LINE EXIT THEN
   CASE 10 ( linefeed ) = 12 ( formfeed ) = OR
     OF #BLANK-LINES @ 0=
        IF  ?NEW-LINE  THEN
        EXIT THEN
   0 #BLANK-LINES !
   CASE 32 < OF ( do nothing ) EXIT THEN
   EXIT ;
```

Figure 7

```
( #YEAR is a variable holding the year )
:LEAPYEAR? ( -- tf : true if the year is a leap year )
#YEAR @
   CASE 400 MOD 0=  OF  TRUE  EXIT THEN
   CASE 100 MOD 0=  OF  FALSE EXIT THEN
   CASE   4 MOD 0=  OF  TRUE  EXIT THEN
   DROP FALSE ;

: DAYS ( month# -- days-in-month )
  CASE  9 =  4 =OR  6 =OR  1 =OR  OF   30 EXIT THEN
  CASE  2 = NOT                   OF   31 EXIT THEN
  DROP LEAPYEAR? IF  29  ELSE  28  THEN ;
```

Figure 8

```
: CBASE!  ( a c -- a' )
  CASE  ASCII $ =  OF  HEX     1+ EXIT THEN
  CASE  ASCII @ =  OF  OCTAL   1+ EXIT THEN
  CASE  ASCII % =  OF  BINARY  1+ EXIT THEN
  CASE  ASCII & =  OF  DECIMAL 1+ EXIT THEN
  DROP ;

: BASE-NUMBER  ( a -- d )
  BASE @ >R  DUP 1+ C@ CBASE!
  NUMBER?   R> BASE !  0= ABORT" ?" ;
```

Figure 9

```
HEX
: CLASSIFY ( n -- )
  CASE  20 < 7F =OR          OF ." Control character" EXIT THEN
  CASE  20 2f BETWEEN
  OVER  3A 40 BETWEEN OR
  OVER  5B 60 BETWEEN OR
  OVER  7B 7E BETWEEN OR     OF ." Punctuation" EXIT THEN
  CASE  30 39 BETWEEN        OF ." Digit" EXIT THEN
  CASE  41 5A BETWEEN        OF ." Upper case letter" EXIT THEN
  CASE  61 7A BETWEEN        OF ." Lower case letter" EXIT THEN
  DROP                          ." Not a character" ;
```

Since **DUP** is assembler code, in most systems you can optimize its definition with something like that in Figure 10a. The Forth-79 definition of **=OR** is given in Figure 10b. If you do not have **PICK**, as in fig-FORTH, or if **PICK** is not an assembler code definition, see Figure 10c.

Figure 10a

```
CREATE CASE  ' DUP ( CFA )  @  ' CASE (CFA ) !
```

Figure 10b

```
: =OR  ( n tf n -- n tf ) 3 PICK = OR ;
```

Figure 10c

```
: =OR ( n tf n -- n tf ) >R OVER R> = OR ;
```

A **case** statement in any programming language is intended for a series of tests to classify a value. To do this in other languages without using a **case** structure would require repeating the value at each test, giving a tedious appearance to the source In Forth, the data stack allows us to avoid such explicit references to the value In Forth, a **CASE** statement has the pattern **DUP ... IF DROP ...** We have sweetened this to **CASE ... OF ...** .

The trivial nature of the implementation emphasizes that a **case** statement is not essential to Forth. Those Forth practitioners who pride themselves on how lean and mean their Forth is will find it superfluous. My intent is not to propose this definition of **case** for standardization; but on the other hand, any further **case** proposal should be as simple to implement, as portable and as powerful.

Auxiliary Definitions#

You may already have some of these. Your definitions may be different from those shown in Figure 11a. **#BLANK-LINES** and **?NEW-LINE** are words peculiar to the application. **#BLANK-LINES** is a variable counting the number of successive blank lines. **?NEW-LINE** does a **CR** when the value of **#BLANK-LINES** is less than two.

Figure 11b provides definitions for several fundamental Forth words. It also presents a naive version of **NUMBER?** that ignores details such as sign and punctuation, and is not in tended for actual use.

Figure 11a

```
: WITHIN  ( n n1 n2 -- tf  : true when n1 <= n & n < n2 )
  over - >R  - R> U< ;

:BETWEEN ( n n1 n2 -- tf : true when n1 <= n & n <= n2 )
 WITHIN 1+ ;

: ASCII ( ^ c -- c : integer value for character c )
```

```
  BL WORD COUNT 1- ABORT" ?"  C@ STATE @
  IF [COMPILE] LITERAL THEN ; IMMEDIATE
```

Figure 11b

```
: HEX ( -- ) 16 BASE ! ;
: OCTAL ( -- ) 8 BASE ! ;
: BINARY ( -- ) 2 BASE ! ;
: DECIMAL ( -- ) 10 BASE ! ;

: NUMBER? ( addr -- dn tf ) 0 0 ROT CONVERT C@ BL = ;
```