A:\>
A:\>

Screen # 1
(GENERIC OPERATORS Terry Rayburn TGR 16:48 10/25/85)
Suppose we wish to extend the TO concept by preparing a set of words for defining data structures and the operators appropriate for those structures. Simple operations like fetch, store, display and input are desired for several different data types, e.g.: integer, double, register or vector. It would be nice to use operate on instances of these types with generic opators like FROM and TO instead of type-specific words like B@, @ or 2@. What would we like such a structure to do for us?
First, we should like to have readable syntax for the application program. In English this suggests a prefixed form as " Operator Operand"; e.g.: DISPLAY NARF. Second, we would like for the declaration of the instances of the data type to be as concise as possible, as easy as VARIABLE NARF. Third, we would like to hide all the methods of access with the type definition and bind them automatically to the instances of the

Screen # 2
(TGR 16:03 10/25/85)
of the data type. Third, we would like the definition of data types to be as readable as possible, preferably an analog to an approach with which we are already familiar. Finally, we would like to minimize the penalty for using such an approach.
As a starting point, suppose each instance of a data type pointed to a table of methods for that type. The first method might be the fetch operation selected by the generic operator FROM, the second might be the TO method, etc. This table could be a list of CFAs. Since the front end of such a type definition is a CREATE and some data allocation, and the tail end is a list of CFAs, we choose a sytax that looks like:
: REGISTER CREATE 0 , METHODS> [list of 1 - n operations] ;
We let the compiler do the work of building our table instead of creating some complex code of such as:
  ' BYTE.FETCH.METHOD ,

Screen # 3
(TGR 16:16 10/25/85)
It might be desirable to let the naming of an instance of a data type have some default behavior. Most would like that the naming of an INTEGER would put its value on the stack like a CONSTANT does. Since all operators can be defined as METHODS> there is no need for the address. If an instance is named, we let the first method be the default, then the programmer may pick his favorite operation.
We see then that the definition for METHODS> is just a version of DOES> that only executes the first Forth word following rather than all words following. This allows implementation of this code piece by looking at and understanding the behavior of DOES> in the Forth implementation at hand.
We prefer that the operator have minimal impact at run-time. Rather than have an operator look into the METHODS> table at run-time, we do it at compile time. The generic operators

Screen # 4
(TGR 16:31 10/25/85)
IMMEDIATE and look up and compile the CFA of the type-specific operator given in the METHODS> table. For minimum space, we would like the form: FROM NARF
to compile: / cfa.of.fetch.method / pfa.of.narf
This is an implementation detail left as an exercise for the reader. I choose to waste a little space to preserve the analog with CREATE ... DOES> and compile:
/ LITERAL / pfa.of.narf / cfa.of.fetch.method /
so that all the methods may expect the address of the data area to be on the stack.
The original version of this was done under polyForth II for the IBM PC and the PDP-11. For educational purposes, here is the MS-DOS F83 version. I intend to create a version for Laboratory Microsystems, Inc. PC-FORTH shortly.
Note on screen 3 the word DO-METHOD. This is the

Screen # 5
(TGR 16:45 10/25/85)
interactive version. I usually define SO as DO-METHOD. Then I enter SO DISPLAY NARF to see a value during debugging. A colleague defines // as DO-METHOD. It would be perfectly acceptable to me to make METHOD: decide whether to compile or execute.
Finally, I present an idea for future work. I consider the data types that have ortogonal methods tables to be members of the same super class. For example, INTEGER and BOOLEAN types might have the operators shown on screen 4. A data type of LINKED-LIST would have generic operators nothing like this at all, but its operators might look very much like those of type B-TREE. So far, I have no way to make such a relationship explicit.
These techniques have been in use about 4 months by 4 different programmers on two projects. The result has been a dramatic improvement of the readability of our code.

```
  6
Scr # 1          B:METHODS.BLK
  0 \ methods                                                  9-24-85tgr
  1
  2 code domethods
  3    ax pop    w inc    w inc    w push
  4    ax w mov    0 [w] w mov    0 [w] jmp
  5 end-code
  6
  7 : methods> compile (;code)    232 ( CALL ) C,
  8     [ ' domethods 2+ ]    literal    here 2+ - , ; immediate
  9
 10
 11
 12
 13
 14
 15
```

```
Scr # 2          B:METHODS.BLK
  0    \ [method], method:  do-method                          9-21-85tgr
  1 : [method] ( n)
  2    defined if     dup 2+ [compile] literal @ 3 + + @ ,
  3        else    count type abort" ?"
  4        then ;
  5
  6 : method:    create , immediate  does> @ [method] ;
  7
  8 : -defined ( pfa -)    1 abort" method not defined" ;
  9
 10
 11
 12
 13
 14
 15
```

```
Scr # 3          B:METHODS.BLK
  0    \ do-method                                             9-24-85tgr
  1 : do-method
  2    DEFINED
  3       IF >BODY @                          \ method offset
  4       DEFINED
  5          IF   DUP >BODY SWAP @   ROT  3 + + @  EXECUTE
  6          ELSE COUNT TYPE  ABORT" ?"
  7          THEN
  8       ELSE COUNT TYPE  ABORT" ?"
  9       THEN ;
 10
 11
 12
 13
 14
 15
```

```
Scr # 4          B:METHODS.BLK
  0 \ operators for class DATA                                  9-24-85tgr
  1  0 method: from
  2  2 method: to
  3  4 method: enter
  4  6 method: display
  5  8 method: tally                8 method: set
  6 10 method: clear               10 method: reset
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Scr # 5          B:METHODS.BLK
  0    \ data: register                                         9-21-85tgr
  1 : r.enter ( a-)
  2    space pad 6 blank  pad 5 expect  pad 1- number drop  swap ! ;
  3
  4 : r.display ( a-)   @ 0 6 d.r ;
  5
  6 : r.tally ( a-)   dup @ 1+ swap ! ;
  7
  8 : register   create 0 ,   methods> @
  9                                    !
 10                                    r.enter
 11                                    r.display
 12                                    r.tally  ;
 13
 14
 15
```

```
Scr # 6          B:METHODS.BLK
  0     \ methods testing                                       9-21-85tgr
  1
  2 register narf
  3 register mod.narf
  4 : barf   tally narf  display narf enter narf
  5          narf 3 mod to mod.narf    display mod.narf ;
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```