$10 Volume XIII, Number 5 January 1992 February

FORTH
DIMENSIONS

Yerk Comes to the PC

Object-Oriented Forth

Simple Object-Oriented Forth

Contents

Features

Departments

Editorial

Objets d'Art

Object-oriented programming has been slow to excite the collective imagination of the Forth community. It's hard to say why, because OOP and Forth techniques seem very congruous; each sheds light on the other and suggests further refinements. Maybe Forth programmers who look at OOP do so superficially, seeing the easy parallels but not the depth; or maybe we unconsciously remember our schoolteachers' prohibitions against passing messages in a class…

This issue shows object-oriented Forth from several angles: we are pleased to present the winners of *FD*'s object-oriented Forth contest. They are Rick Grehan, Roger Bicknell, and Clive Maynard. Their names are listed in order here, and our referees were hard pressed to determine the final standings. Different and more-or-less-complete approaches to implementing OOF are represented. There is some inevitable overlap in the tutorial sections, but each article contains its own particular insights.

How you view and use Forth will determine which of the code in this issue you will choose for experimentation. Look past the surface, into the deeper implications of object-oriented Forth, and let us know what you find there. We were unable to publish all of the good material submitted to this contest, so we hope to present more in the future, along with the results of your own OOF explorations!

Due to the amount of material generated by the above-mentioned articles, along with a lengthy and revealing excerpt from the on-line ANS Forth debate ("Best of GEnie"), our usual "reSource Listings" have been postponed. A few updates are included, though, and the entire listings will reappear soon.

Finally, welcome to the new year, traditionally a season of fresh beginnings. (Time to back up your data and reformat that hard disk.) *FD* is exploring upgrade options for coming issues, including more items about Forth-based solutions in action, Forth news, press releases and articles from vendors and developers, and a switch to wider text columns. Along with, of course, the fine technical fare *FD* readers expect.

But this magazine does not operate in a vacuum. (Do I repeat myself?) New articles and departments come when someone is inspired (or convinced) to write them. Press releases can get published only if businesses mail them. And developers' work gets known after they tell their peers about it. So take advantage of your citizenship in our virtual community. You might even give an *FD* subscription (i.e., membership in the Forth Interest Group) to your boss, company library, or co-worker. As one of our letter writers says this month, "We must not do nothing." That would, after all, be doubly negative.

—*Marlin Ouverson*
Editor

dpANS Forth Released for Public Review

The Draft Proposed ANS Programming Language Forth entered its official public review period in October. Copies of the proposed standard may differ from development versions (i.e., the "BASIS" documents), and can be purchased from Global Engineering Documents, Inc., 2805 McGaw Avenue, Irvine, California 92714. Ask for document #X3.215-199x. From within the United States and Canada, call 800-854-7179; from other countries, call 714-261-1455. The U.S. price was to be $50 per copy; for international orders, $65 per copy.

The public-review period extends from October 18, 1991 through February 25, 1992. Please send all comments to X3 Secretariat/CBEMA, Attention: Lynn Barra, 311 First Street N.W., Suite 500, Washington D.C. 20001-2178. Send a copy of your comments to American National Standards Institute, Attention: BSR Center, 11 West 42nd Street, New York, New York 10036.

Changes from Forth-83 include removal of ambiguities and restrictions, numerous optional language extensions, optional extensions for floating-point math, string handling, programming tools, additions to facilitate porting programs across disparate CPUs, and an optional interface between Forth and operating systems like UNIX, VMS, OS-2, and MS-DOS.

Forth Dimensions
Volume XIII, Number 5
January 1992 February

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Anna Brereton

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155. Administrative offices and advertising sales: 805-946-2272. Fax: 408-286-8988

The Forth Interest Group
The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

Letters

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

Marketing vs. Objectivity & Public-Domain Glut

I agree with many of the points mentioned in "Singapore Slingshot Targets FIG Issues" (letters, *FD* XIII/4). I do not see anything wrong with self-serving articles by Forth vendors. I certainly *hope* that *FD* readers are mature enough not to need to be "protected" from marketing hype. Many successful and well-respected trade journals are full of articles touting particular vendors' products. So long as the company affiliation of the author is identified, I have no problem distinguishing an "objective" article from a "marketing" article, and I read both kinds with interest.

I take issue with Mr. Tse's point number 19, in which he suggests another "model" system. In my opinion, the *last* thing the Forth community needs is Yet Another Public-Domain Forth. There are way too many public-domain Forths already, and creating another one will further erode the ability of the few Forth vendors that are left to make a living. The Forth community *needs* profitable vendors, because profits result in money that can be spent on advertising and marketing. Forth *desperately* needs visibility, and like it or not, visibility results from dollars spent on marketing.

Furthermore, it is extremely unlikely that the existing Forth vendors who have managed to stay in business for several years would switch to the proposed "super-duper" Forth system.

Forth to the Future,

Mitch Bradley
Bradley Forthware
P.O. Box 44
Mountain View, CA 94040

We Must Not Do Nothing

Dear Marlin,

Forth is the artist's language. It allows us to tap the computer's true potentials and create things of beauty—beauty in simplicity, conciseness, elegance, and speed. Is there anything more satisfying than creating a powerful algorithm, which does exactly what it is supposed to do, using about half-a-dozen words?

Forth has been around for two decades now, and many brilliant contributions have been made by the Forth community. Yet, despite the sustained efforts of many, there has been no widespread recognition and use of Forth.

Many explanations can be found for this. However, it does no good to speculate, feel sorry for ourselves, or be righteously indignant, if this doesn't lead to improved conditions.

Judging by recent *FD* articles, such as last issue's President's Letter ("I Have a Dream") and Mr. Tse's letter for the editor ("Singapore Slingshot..."), I believe that Forth is an idea whose time has come. These letters express a desire to change things for the better. I think that most of us share that sentiment.

The difference between Forth fading into obscurity and Forth becoming the foremost innovative force in the computer industry lies in what we, as individual FIG members, do about it. The one thing we must not do is nothing.

All of us have unique talents that we can contribute to help Forth expand. Let us, then—each of us—do something, no matter how small initially, to get the ball rolling. By doing so, we will eventually reap the rewards. Imagine the satisfaction of having our children or grandchildren say, "Wow, you were one of the guys who put Forth on the map!"

Peter Verhoeff
P.O. Box 10424
Glendale, California 91209

Forth on a Bathroom Scale—No Lightweight

Dr. Ting's letter to Mr. Koopman (*FD* XIII/3) inspired me to share my feelings about Forth and its future.

A few years ago, I started working at a thin-film circuit startup. The boss like Forth. Make that *loved* Forth. Therefore, I was dragged kicking and screaming into what turned out to be a really great language. My only instruction came from Mr. Brodie's *Starting Forth*. (I think Brodie deserves some kind of award for his contribution to Forth.) I had never taken a computer course.

In my four years with the company using Forth (first F83, and later LMI Forth), I managed to:

- From scratch, build an automated wafer-probing machine that tested hundreds of points on a semiconductor wafer, "learned" from the operator, and saved the test results to disk.
- Write the operating software, with a graphics interface, for a custom semi-automated, thin-film sputtering machine. Some amount of artificial intelligence was used in this project.
- Computerize a Dektak film-thickness profiler. Profiles were drawn in real time on a CRT, and could be zoomed in on, saved to disk, etc. The operator controlled the machine with a mouse.
- Starting with a bathroom scale, built an adhesion tester that measured film-to-wafer adhesion strength. A computer displayed the adhesion strengths in PSI and Pascals.
- Get a Harris RTX-2000 Forth engine interfaced to a liquid crystal display and drawing graphics.

Software development with Forth was fast. I usually had something coming to life in a matter of hours. Modifications were sometimes made practically in real time, while the machines were in production use.

Well, enough of the real-world applications of Forth. At issue is, why does its future look somewhat dismal, and what does it need to succeed?

1. It is reasonably well marketed.
2. It has a professional programming environment (vs. F83).
3. It is available in versions with a street price under $100.

When these requirements are met, people are more likely to spend the few dol-

It was a dream come true. It was a nightmare.

(Letters continue on page 13.)

Yerk Comes to the PC

Rick Grehan
Peterborough, New Hampshire

Some time ago, I discovered a Forth-based, object-oriented programming system for the Macintosh called Neon. For whatever reason, Neon was discontinued. Recently, however, the language has reappeared in the public domain as "Yerk," largely due to the work of Bob Loewenstein at the Yerkes Observatory (hence the new name).

I have always been impressed by Yerk's object-oriented abilities. Though I continue to work on the Macintosh, I do enough in MS-DOS that I began to wish for something like Yerk on the PC. So I wrote PCYerk, a moderately complete duplication of the Yerk syntax for the PC. I say "moderately" because there are some Yerk capabilities—not counting all the Mac toolbox routines Yerk has access to—that PCYerk does not support.

I've written PCYerk using Upper Deck Forth, a 16-bit multisegmented Forth for PCs running MS-DOS. It's likely that someone well-versed in F-PC could easily port Yerk to that system. I'll use this description of PCYerk as a vehicle for introducing you to some object-oriented concepts.

Objects and Classes

An *object* is a combination of code and data that your program can treat as an indivisible entity. The code associated with an object is really a collection of routines that manipulate that object's data and allow that object to interact with other objects in the program, I/O devices, and other parts of the system.

Guidelines discourage low-level manipulation of an object's interior, except via messages.

A *class* is a kind of template for an object. The class definition describes the mechanics of offspring objects, and consists of two main pieces: the *methods* (what objects know how to do) and the *instance variables* (an object's local data).

When you create an object, we say that the object has been *instantiated*: the class template has forged something real (as real as any piece of code can get). Once you have instantiated an object, you can make that object do things by sending it *messages*. Formally, a message is composed of two parts: a *selector* and some attendant data. The selector is an ID number that the object uses to determine which method to execute. In PCYerk, there's little difference between message and selector. The message is the selector; any data is passed on the parameter stack.

In PCYerk, the defining word for a class is `:class`. Listing One shows the source code for a simple class called `integer`. Notice that the class definition is bracketed by `:class` and `;class` (I'll explain the other parts of the definition later). These two words serve to encapsulate the class definition.

Instance Variables

As I mentioned, instance variables define the local storage associated with an object. If you look again at Listing One, you'll see that objects of the class `integer` possess a single instance variable: `localdata`. The word `ivar` expects a value on top of the stack that indicates the number of bytes to be allocated by the current instance variable—whose name `ivar` parses from the input stream. Whenever you create an object of type `integer`, the system knows to allocate two bytes of variable space.

Instance variable names last only as long as the class definition. In other words, upon execution of the code in Listing One, the symbolic name `localdata` is discarded (the word `;class` does this).

Messages and Methods

Objects of class `integer` understand two messages, `get:` and `put:`. Each of these messages corresponds to a method of the same name. I can create an object of type `integer` and store a 12 in it with:

```
integer myint
\ Create the object
12 put: myint
\ Send a put: message
```

As you've probably guessed, I can retrieve the contents of `myint` using the `get:` message. The only way a program can legitimately manipulate the instance data of `myint` is via `get:` and `put:`. I say "legitimately" here because a clever Forth programmer can manipulate anything. Object-oriented programming guidelines, however, discourage low-level manipulation of an object's interior except by that object's messages.

Sometimes you need an object to automatically execute a method when that object is instantiated. A good example would be an array class whose initialization method allocates space for the array, then stores in an instance variable the number of members allocated to that array.

The PCYerk syntax for setting an initialization method is shown in Listing Two, where I've defined a

Rick Grehan is a senior editor at *BYTE* magazine, where he is the technical director of BYTE Lab. He first encountered Forth over seven years ago when developing a music synthesizer control system built around a KIM-1. Since then, he has used Forth on 68000 systems (including the Macintosh), the Apple II, and the IBM PC. He has also done extensive work on the SC32 stack-based processor. Rick has a B.S. degree in physics and applied mathematics, and an M.S. degree in mathematics/computer science.

Listing One. Source code for a simple class.

```
:class integer
2 ivar localdata

:m get:    ( -- n )
   localdata @
;m

:m put:    ( n -- )
   localdata !
;m
;class
```

Listing Two. Initialization and self-referencing.

```
:m clear: ( -- )
   0 put: self
;m

clear <<init-method
```

method called clear:, which stores a zero in the local data. You must place the <<init-method word prior to ;class.

Listing Two also shows the use of the self word. This allows an object to reference itself in a method definition. It's as if the clear: method were saying: "Okay, object, here's a zero on the stack. Now send a put: message to yourself."

Inheritance

Inheritance is one of object-oriented programming's big buzzwords. The concept is simple enough: you start with simple classes and build on those to create classes that are incrementally more complex and specialized. Each new class within an inheritance chain carries all the knowledge (i.e., methods and instance variables) its ancestors carry.

For example, suppose I've defined a class called 1darray that allows you to create a one-dimensional array. Objects of this class would have methods to allocate array space, store a value to an index location, and retrieve a value from an index location.

Next, I define a class called $ptr_array that manipulates an array of string pointers. I want this new class to know how to manipulate a one-dimensional array (something 1darray can do), but it should also be able to print a string. In short, I want objects of the $ptr_array class to inherit the abilities of the 1darray class, then add some string-handling capabilities.

PCYerk provides inheritance via the <super word. Usage goes like this:

```
:class
$ptr_array
<super 1darray
```

which defines a new class $ptr_array whose superclass is the class 1darray.

The <super word does a number of things. First, it stores a link in the class-definition header. This link points to the class-definition header of the superclass. Whenever an object receives a message, that object first searches its class's methods list. If it can't find the method corresponding to the message, the system follows the superclass link and searches the superclass's methods list. This process continues until either the method is found or all classes on the chain have been searched. Such are the mechanics of methods inheritance. (Notice that, since the search takes place for local methods first, an object can redefine an inherited method.)

Next, <super initializes the class's instance variable space accumulator with the amount stored in the superclass's. In other words, a class inherits the instance variable space requirements of its superclass.

Finally, <super sets the class's initial method to the initial method of the superclass. The class inherits its superclass's initial method. As described above, you can override that initial method with the <<init-method word.

Inheritance is the basis for *polymorphism*, yet another piece of object-oriented jargon. Polymorphism means that different offspring of a given superclass respond to the same message selector differently. I might define a subclass of integer called integer_array. An integer object would respond to the get: message by retrieving the value in the single instance variable. Meanwhile, an object of class integer_array would respond to the same message by retrieving an index value from the stack and fetching the appropriate array element. Same message selector, different action.

Binding

Binding is the process of taking a message's selector and determining the executable address (i.e., the method) associated with that selector. Of course, the method depends on the object class you're sending the message to. There are two kinds of binding: early binding and late binding.

You've already seen early binding; it looks like this:

```
get: my_integer
```

which retrieves the contents of my_integer. The system knows the message (get:) and the object (my_integer) and can, therefore, locate the execution address of the method corresponding to the message at compile time.

(Internally, when the system encounters a message selector name, it places a 16-bit ID number on a special stack called the method stack, or mstack for short. When it encounters an object name, the system places the object's address on another special stack, the object stack—ostack—and transfers control to the word exec-obj, which pops the mstack and performs the binding. While the method executes, instance variables can resolve their addresses by calculating offsets from the object address on the ostack.)

Late binding is also referred to as deferred binding. In simple terms, it means the system doesn't know what object you're going to send a message to. Hence, the system can't bind the message to an execution address at compile time.

Here's an example of PCYerk's late binding:

```
get: { integerobj @ }
```

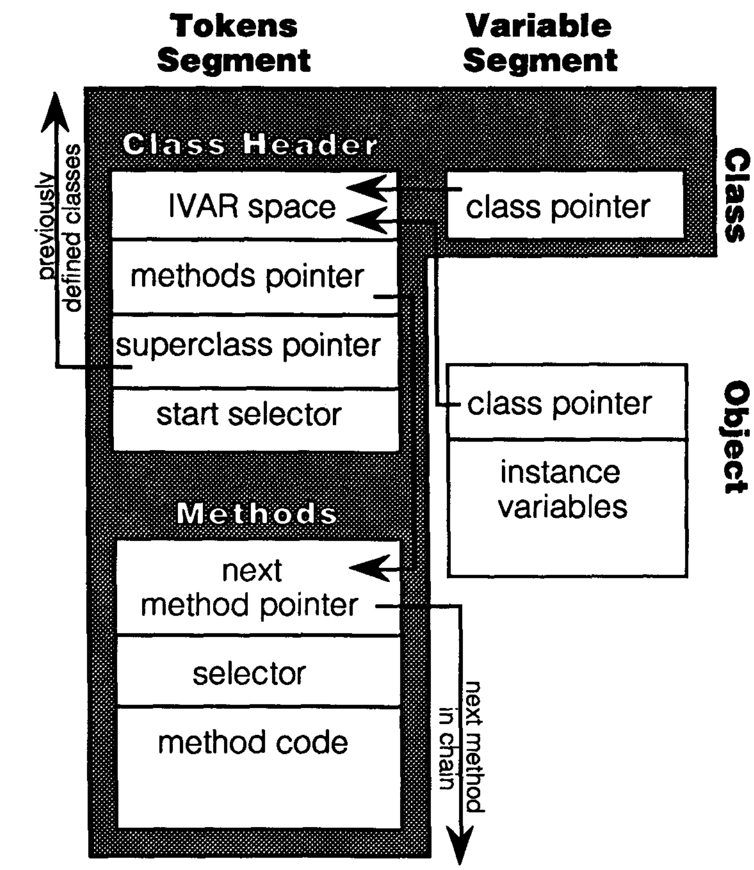
Here, I'm assuming that the variable `integerobj` holds an object's address. At compile time, there's no way the system can know what object will be in `integerobj` when the program executes. The system must, therefore, determine the execution address of `get:` at run time.

The words to handle late binding are `{` and `}`. (Yerk used `[` and `]`, but those words were already taken in UD Forth.) The curly brackets should immediately follow the message, and may enclose any Forth expression that yields an object's address.

Of course, late binding yields code that runs more slowly than code using early binding. This is because late binding defers until run time processing that would have been performed at compile time.

Late binding lets you—with only minor additional programming—bypass one of PCYerk's deficiencies: namely, that you define all instance variables using the `ivar` word. You cannot use an object as an instance variable. Suppose you've defined a class called `1darray` that lets you build one-dimensional array objects, and a class called `polygon` that builds polygon objects. It would be nice to have a `1darray` object as one of the instance variables of the

Figure One. Class header structure.



Listing Three. Building a headerless object.

```
:class polygon <super object
 2 ivar ^vertex_list

\ Allocate space for vertex list
:m init:  ( n -- )
     2*                  \ 2 coordinates per vertex entry
     here dup ^vertex_list !  ['] 1darray >body @
     instantiate
;m

init: <<init-method

:m ->vertex:    ( y x i -- )
     2*              \ Index * 2 for x and y coordinates
     tuck            \ Get a second copy of the index
     to: { ^vertex_list @ }    \ Store x
     1+              \ Advance index to next slot
     to: { ^vertex_list @ }    \ Store y
;m

;class
```

Listing Four. PCYerk source code.

```
\ PCYerk
\ Object-oriented extensions to Forth
\ a la Yerk (once, NEON)
\ Written for Upper Deck Forth, version 2.0
\ R. E. Grehan

\ Sorry, I can't stand "then"
: endif [compile] then ; immediate

\ *************
\ ** STORAGE **
\ *************
34 $variable tstring              \ Used in parsing names

\ Method stack
20 constant METH_STACK_SIZE
create mstack
      METH_STACK_SIZE allot

\ Objects stack
20 constant OBJ_STACK_SIZE
create ostack
      OBJ_STACK_SIZE allot

\ Instance variable names segment
variable ivar_seg                 \ Segment
20 constant IVAR_SEG_SIZE         \ Segment size in paragraphs
variable ivar_next                \ Offset to next free loc.

\ Methods names segment
variable methname_seg             \ Segment
100 constant METHNAME_SEG_SIZE    \ Segment size in paragraphs
variable methname_next            \ Offset to next free slot
variable curr_meth#               \ Current method #
variable my_meth#                 \ Method # about to be defined

\ Class definitions
variable   curr_class_off         \ Offset to current class
variable   curr_meth_tail         \ Current method tail

\ ******************************
\ ** METHOD AND OBJECT STACKS **
\ ******************************
\ NOTE: Neither stack do any bounds checking (for speed's
\ sake).  If bounds checking is added, the stack manipulation
\ words should be written in machine language.

\ Initialize the method stack...stores selector ids
: mstack-init     ( -- )
  mstack dup !
;

\ Push top word onto method stack
: mpush            ( n -- )
  mstack @ 2+ !                  \ Save item
  2 mstack +!                    \ Increment
;

\ Copy top of mstack to dstack
: mstack->dstack  ( -- n )
  mstack @ @
;
```

(Listing continues.)

polygon class. Then you could store the polygon's vertex list into the one-dimensional array object.

It turns out you can do this by defining one of the polygon's instance variables to be a pointer to an object of class `1darray`. Essentially, you build a headerless object; that is, one that does not have a head in the Forth dictionary. The word `instantiate` will build such a headerless object. All `instantiate` needs on the stack is a starting address in the variable region (where the new object will go) and a pointer to the class definition. The code showing this technique is in Listing Three (which presumes that you've defined the `1darray` class already).

Notice that I've defined `init:` to multiply the number of vertex entries by two, since the vertex list will carry an x and y coordinate for each vertex. The next line stores the address of the next free location in the variable `segment` into `^vertex_list`, which becomes a pointer to our one-dimensional array. I then use `[']` to retrieve the code address of the `1darray` class, then fetch the address of that class definition. The `instantiate` word actually creates the array. Keep in mind that, when `instantiate` executes, the `1darray` object's initial method—which allocates the memory space—will be executed.

Now we can store an x and y coordinate into the `1darray` object using a method called `to:`, which we've presumably already defined for the `1darray` class. You can extend this idea as far as you'd like to go, creating pointers to objects with pointers to objects, and so on.

Nuts and Bolts

Refer to Listing Four, the complete source for PCYerk. The heart of PCYerk is

:class. The :class word uses a nested create ... does> structure—the kind that makes my head hurt whenever I have to think about it. At compile time, :class builds a class header structure as shown in Figure One.

I have to take a moment here to describe something of the structure of UD Forth. Being on the IBM PC, UD Forth has a segmented architecture. Executable code resides in the code segment, variables are stored in the variables segment, threading pointers are kept in the tokens segment, and names are kept in the headers segment. Names built using Forth's create word return a pointer to a parameter field in the variables segment, hence the class pointer in that segment for a class definition.

The first field in the header—ivar space—tells the system how much space to set aside for instance variables when an object is created. The second field is the head of a linked list that connects all the methods for a particular class. Next comes the superclass pointer field, which is set by the <super word, and which provides the means by which a class inherits methods from its superclass. The last word of the class header is the start selector; it identifies which method will be automatically executed when you create an object.

Objects carry execution addresses defined by the code following the second does> in :class. When you send an object a message, the system follows the pointer to the class header, then searches down the methods list chain (as described above) to determine what code to execute. Notice that the code for a method is absolutely headerless; a method doesn't even possess code field addresses. A special word—(domethod)—executes a

(PCYerk Listing Four, continued.)

```
\ Pop top word from method stack
: mpop              ( -- n )
  mstack->dstack                          \ Fetch
  -2 mstack +!                            \ Decrement
;

\ Initialize the object stack
: ostack-init       ( -- )
  ostack dup !
;

\ Push top word onto object stack
: opush             ( n -- )
  ostack @ 2+ !                           \ Save item
  2 ostack +!                             \ Increment
;

\ Copy top of ostack to dstack
: ostack->dstack    ( -- n )
  ostack @ @
;

\ Pop top word from object stack
: opop              ( -- n )
  ostack->dstack                          \ Fetch
  -2 ostack +!                            \ Decrement
;

\ Dup top of object stack
: odup              ( -- )
  ostack->dstack                          \ Fetch top
  opush                                   \ and push
;

\ Drop top of object stack
: odrop             ( -- )
  opop drop
;

\ Clear both stacks.
\ Use this if something aborts and you don't want the
\ stacks growing forever.
: clear-o&mstacks
  ostack-init
  mstack-init
;

\ ***********************
\ ** TEMPORARY SEGMENTS **
\ ***********************
\ The method names and instance variable names are kept in
\ temporary segments.  These segments are allocated
\ from DOS.  When you're done defining things and
\ its time to make an executable, just free those
\ segments. (The word 'end-objects', defined later,
\ does all that.

\ Compare two counted strings. seg1 addr1, seg2 addr2 point to
\ segment and addresses of two strings with preceding count
\ bytes. Returns 0 if equal, else nonzero
: ccompl     ( seg1 addr1 seg2 addr2 -- n )
  \ First check byte counts
  count1 >r 2swap count1 r@ =
```

```
  if        r> compl            \ Lengths match...try comparison
  else      4drop r>drop 1      \ Show mismatch
  endif
;

\ Advance to next item past the current counted string.
\ seg:addr points to counted string.  Takes into account
\ trailing integer.
: nextstr   ( seg addr -- seg addr' )
  countl + 2+
;

\ Search one of the temporary segments.
\ seg1:addr1 points to string to search for
\ seg2 is temporary segment to search
\ max is maximum current offset in segment.
\ n is returned associated integer; -1 means
\ the string was not located.
: search-tseg     ( seg1 addr1 seg2 max -- n )
  dup                             \ Anything to look for?
  if
      >r                          \ Save max
      0                           \ Start search at zero
      begin
            2over 2over           \ Dup seg/address
            ccompl                \ Look for match
            0=
            if     2swap r> 3drop \ Clear stack
                   countl + @l    \ Fetch value
                   exit
            endif
            nextstr               \ Advance to next string
            dup r@ >=             \ Topped out?
        until
        r>drop                    \ Clear return stack
  endif
  4drop                           \ Clear stack
  -1                              \ Show error
;

\ **
\ ** Instance variable segment handling
\ **
\ The ivar segment is a temporary region where the system
\ keeps a list of the current class definition's instance
\ variables.  Each entry is composed of a length byte, the
\ name, and a 2-byte value that indicates that instance
\ variable's offset into an instance of the class

\ Allocate space for the IVAR segment.  Place the segment
\ in global variable ivar-seg
: alloc-ivar_seg   ( -- )
  IVAR_SEG_SIZE alloc
  error                                  \ Fetch error
  if  abort" Ivar allocation error"
  endif
  ivar_seg !                             \ Save pointer
;

\ Clear the ivar segment
: clear-ivar_seg   ( -- )
  0 ivar_next !
;
```

(Listing continues.)

method by mimicking the run-time action of the colon word. You pass `(dometh-od)` the starting address of the method code and it handles the rest.

PCYerk does not create standard Forth headers (á la `create`) for instance variables and methods. In the case of instance variables, you want their names to disappear after the class definition. For method names, you don't want their names taking up header space, since they are instantly resolved to two-byte selector ID numbers.

PCYerk allocates two memory blocks (using the UD Forth word `alloc`, which provides access to the DOS function for allocating a memory segment): one to hold instance variable names, the other to hold method names. Each name stored in one of these blocks is associated with an integer.

In the case of instance variables, the associated integer carries that instance variable's offset into the object's local data space. At compile time, when the system encounters an instance variable, it looks up the variable's offset and compiles that as a literal, followed by the word `(ivar)`. At run time, `(ivar)` takes the offset from the stack and resolves that offset to an address.

In the case of methods, the associated integer is the selector ID number. When you define a new method's name, the system increments an internal counter and the incremented value becomes that method's ID number. This ensures a unique ID number for each method. (Yerk used a hashing method to generate such ID numbers. I chose a separate route, since the code for handling instance variable names and method names was so similar.)

But wait. If the system puts method names and instance variable names in these alternate segments, how does Forth find those names during compilation? You have to patch `interpret`.

In UD Forth, `interpret` first tries to find the word in the dictionary. If that fails, `interpret` tries to parse the word as a number. If the number conversion routine can't digest it, `interpret` executes `do-undefined` (which prints out the offending word and executes `quit`.) I overwrite the call to `do-undefined` to point to `<interp-patch>`. The `<interp-patch>` word (see Listing Four) looks first in the method segment, then in the instance variable segment. If `<interp-patch>` finds the word in either segment, it takes appropriate action. Of course, `<interp-patch>` ultimately falls through into `do-undefined`.

You have to execute `start-objects` before you begin defining any classes. The `start-objects` word allocates and initializes the instance variable and method names segments, then patches `interpret` and clears the methods and objects stacks. Finally, when you're ready to create a standalone application, execute `end-objects`. This repatches `interpret` to put it back the way it was, and releases the allocated memory blocks (which are unnecessary in the run-time code.)

(PCYerk Listing Four, continued.)

```
\ Given that addr points to a counted string that represents
\ an instance variable name, return the associated offset.
\ If you can't find that variable, return a -1.
: search-ivar      ( addr -- n )
  vars swap                 \ String is in vars segment
  ivar_seg @                \ Search through ivars segment
  ivar_next @               \ Max. to look for in ivar segment
  search-tseg               \ Search a temp. segment
;

\ addr points to a counted string that represents an instance
\ variable name.  n is the offset to attach to that instance
\ variable.  Add this name to the list.
: add-ivar   ( n addr -- )
  dup c@ >r                           \ Save byte count
  vars swap                           \ Source address
  ivar_seg @ ivar_next @              \ Destination
  $!l                                 \ Copy the string in
  ivar_seg @ ivar_next @ r@ + 1+ !l \ Store associated value
  r> 3 + ivar_next +!                 \ Advance next
;

\ **
\ ** Methods name segment handling
\ **
\ The methods segment looks a lot like the IVARS segment.
\ it holds the list of methods defined within the system.
\ Associated with each method name is a unique 2-byte
\ id.


\ Allocate space for the method name segment
: alloc-methname_seg     ( -- )
  METHNAME_SEG_SIZE alloc
  error                                  \ Fetch error
  if  abort" Methname allocation error"
  endif
  methname_seg !                         \ Save pointer
;

\ Clear the method segment
: clear-methname_seg     ( -- )
  0 methname_next  !
;

\ Search for a method in the methods segment.  Return -1 if
\ not found.  Else return method #
: methname-find    ( addr -- n )
  vars swap                 \ String is in vars segment
  methname_seg @            \ Search through ivars segment
  methname_next @           \ Max. to look for in ivar segment
  search-tseg               \ Search a temp. segment
;

\ Add a new method to methods segment.  Associate n with that
\ method as the method's id
: add-methname     ( n addr -- )
  dup c@ >r                              \ Save byte count
  vars swap                              \ Source address
  methname_seg @ methname_next @         \ Destination
  $!l                                    \ Copy the string in
  methname_seg @ methname_next @ r@ + 1+ !l
                                    \ Store associated value
```

```
  r> 3 + methname_next +!          \ Advance next
;

\ ***************
\ **  METHODS  **
\ ***************
\ Methods are kept on singly-linked list. That list is anchored
\ in the class definition structure defined below.  Each entry
\ on a method list looks like this:
\     Token segment
\     [  link to next  ]
\     [  Method id #   ]
\     [ ...tokens      ]

\ Attach a new method to tail. addr on top of stack is assumed
\ to be pointer into token segment
: new-method-tail ( addr -- )
  \ See if we are first method added. If so, attach to parent.
  curr_meth_tail @ ?dup 0=
  if  dup                                  \ Copy ourselves
            curr_class_off @ 2+
            !t
  else      over swap !t                   \ Fix link
  endif
  curr_meth_tail !                         \ We are new tail
;

\ (>super)
\ This routine looks 'up the chain' to an object's super object.
\ Used when searching for methods to execute.
\ addr1 is the current object's address in the token seg.
\ addr2 is the super object's address or 0 if none found
: (>super)  ( addr1 -- addr2 )
  4 + @t                       \ Fetch the super object address
;

\ (domethod)
\ Following code word vectors execution to a method.
\ Assumes that the value on top of the stack is offset
\ into token space for the method.
code (domethod)          ( off -- )
      bp    dec                \ Make room on return stack
      bp    dec
      si 0 [bp] mov            \ Push IP
      bx si mov                \ Get method address in IP
      bx    pop                \ Pop stack
      next                     \ Take off!
end-code

\ domethod
\ Calls (domethod) and clears the object stack.
\ Off is the address of the method code.
: domethod         ( off -- )
  (domethod)
  odrop
;

\ (methid->addr)
\ Given a method id, this finds that method's address in the
\ token segment. addr1 is the address of the object (in the
\  token segment) whose method
\ list we'll search.  addr2 is the method address, or 0 if the
\ method wasn't found.
```

(Listing continues.)

(Letters, from page 5.)

lars to at least give it a spin. Hobbyists will buy it. Small companies will try it. The word gets around.

At this time, the low-cost versions of Forth are not at all well marketed—if you could say they're marketed at all. I have not used F-PC yet, but I assume it fixes the awful programming environment of F83. How can we expect people accustomed to the pleasant Microsoft Quick-x environments to accept F83? The slick, professional Forth versions are just too expensive to become popular.

The same two solutions keep popping into mind: either Microsoft QuickForth or Borlund TurboForth. The successes of the Quick/Turbo BASIC, C, and Pascal packages speak for themselves.

At this time, I am learning C. I don't know if there's something wrong with me, or if it's due to my Forth experience, but I'm certainly *very* disappointed with the C language. Compared to Forth, C seems like a giant step backward. So why bother? To the "professionals" in the personnel departments, C is a recognizable "hot" buzzword on a resume, while Forth just looks like you can't spell.

Finally, I hope the FIG dues increase to $40 will not be the death blow—but it just might. It shouldn't cost $6.67 per issue to publish *Forth Dimensions*. Maybe it's time for a major re-thinking of the structure of the group.

Sincerely,
Steve J. Noll
1288 Winford Avenue
Ventura, California 93004

The Atari Lesson

Dear Editor:

I notice two alarming trends in my latest issue of *FD*: 1) circulation has dropped from an average 1934 to 1750, and 2) a vendor is complaining about lack of support/hostile publication

(Letters continue on next page.)

rules. (Jim Callahan, Harvard Softworks; *FD* XIII/4).

I believe 1) and 2) above are related. Without aggressive marketing (of both Forth products and *FD* itself), all previous creative efforts will wither on the vine. The 240 million people in this country have a lot of demands on their time. If Forth doesn't *appear* worth the effort, people won't invest the time to find out it is. Look at the success of the C programming community. As a dabbler in both languages, I can vouch that there is no shortage of public relations on the C side.

Therefore, I implore you and your staff to listen to these vendor complaints and take action. How about some articles comparing the various hardware Forths (Silicon Composers, etc.). How about articles comparing the advantages and disadvantages of the various software Forths (polyFORTH, HS/Forth, MMS-Forth, etc.). How about inviting the vendors to declare the advantages of their systems in article form (they would probably be willing to pay for the opportunity...).

Failure to take action will lead to suffering what I call the Atari lesson. In 1985, when I bought the computer I'm typing this letter on, the personal computer industry was just taking off. The Apple II was showing its age, the overpriced/underpowered IBM XT was carving a large market share, and the AT had just appeared. Probably the most popular computer was the very limited Commodore 64. The most intriguing computer out was the Apple Macintosh—a user-friendly machine which cost over $2000 (with student discount) for the 256 Kbyte standard. Into this maelstrom jumped a recently reorganized Atari with the ST. A window/mouse-driven machine with a big *color* screen and 512 Kbyte, all for less than $1000. It was a dream

(PCYerk Listing Four, continued.)

```
: (methid->addr)   ( addr1 n -- addr2 )
  swap 2+                               \ Advance to method pointer
  begin
      @t                                \ Fetch pointer
      dup
  while
      dup>r                             \ Save copy
      2+ @t                             \ Fetch id number
      over =                            \ Match?
      if    drop                        \ Clear method id #
            r> 4+                       \ Point to code
            exit
      endif
      r>                                \ Ready for next loop
  repeat
  2drop 0                               \ Show failure
;

\ find-method-code
\ Expects a method id # atop the method stack and an object
\ pointer atop ostack. Locates the method code and
\ leaves it on dstack. In so doing, the method stack is popped.
: find-method-code       ( -- code )
  mpop                                  \ Get method id
  ostack->dstack                        \ Fetch the object
  @                                     \ Address in token seg
  begin
      2dup swap (methid->addr)          \ Get address
      ?dup                              \ Didja find it?
      if    -rot 2drop                  \ Clear the stack
            exit                              \ Bug out
      endif
      (>super)                    \ Not found...go to super object
      ?dup                        \ Any super object??
    0=
  until
  clear-o&mstacks                             \ Clear the stacks
  abort" Method not found"
;

\ Define a method.  This word doesn't do a create...it
\ loads the method name in the method segment (unless
\ its already there), then compiles the code at the
\ end of the object definition.  The code is linked to
\ the preceding method for that object.
: :m
  \ We must be defining a class
  curr_class_off @ 0=
  if  clear-o&mstacks
            abort" Method def. ouside class"
  endif

  \ We are the new method tail...so fix the link
  \ code.
  here-t new-method-tail
  0 ,t

  \ See if the method is in the method seg.  If it is,
  \ return the method #...if not, add this method in and
  \ assign a number.
  blword                                \ Parse the name
  tstring $!                            \ Put it in tstring
  tstring methname-find                 \ Look for the method
  dup -1 =                              \ Found?
```

```
  if  drop                                \ Clear stack
        curr_meth# @ dup                  \ Fetch current method ID #
        tstring add-methname              \ Add method to the class
        1 curr_meth# +!                   \ Bump current method ID #
  endif

  \ Store method # for ;m and set aside space in token seg
  my_meth# !
  0 ,t

  \ Now go ahead and compile the method code.
  [compile] ]
;

\ End of method definition
: ;m
  \ Store the method # so the system can find it
  my_meth# @
  curr_meth_tail @ 2+ !t

  compile unnest                          \ Do a semicolon
  [compile] [                             \ Set interpret state
; immediate

\ *************************
\ **  INSTANCE VARIABLES  **
\ *************************
\ Define an instance variable.
\ Used in the form:
\  n ivar <name>
\ n indicates # of bytes for this instance variable.
: ivar      ( n -- )
  blword                                        \ Parse the name
  tstring $!                                    \ Put it in tstring

  \ See if ivar already exists
  tstring search-ivar -1 <>
  if  abort" Ivar already defined"
  endif

  \ Fetch current offset--add it and ivar to ivar space
  curr_class_off @ dup @t dup
  tstring add-ivar

  \ Update ivar space for next offset
  rot +
  swap !t
;

\ This code does the actual instance variable processing.
\ When he executes, he expects the offset of an instance
\ variable on the data stack.  He also expects an object
\ address (in variable segment) on the ostack.
\ The returned addr is the offset to the instance variable.
: (do-ivar)        ( off -- addr )
  ostack->dstack                          \ Get object address
  2+                                      \ Skip pointer to token seg
  +                                       \ Add offset
;

\ ***************
\ **  CLASSES  **
\ ***************
```

(Listing continues.)

come true. It was a nightmare. The aggressive marketing of IBM and Apple soon gobbled up the whole market. Software sources dried up. With money from huge sales, Apple and IBM improved their machines, leaving Atari in a non-competitive position. Now they make PC clones to stay alive.

Does this sound familiar, Forth programmers? How many of you use C professionally and Forth on the side? In 1985, the Atari ST delivered not only the best bang for the buck, but (to me) it was the all-around best computer available—speed, memory, display, interface, etc. Now I envy '486 EISA machines with MS-DOS 5.0 and Windows. There is a deadly parallel here to what has happened in the Forth community. The secret: to cut costs, Atari didn't invest in marketing, resulting in a product nobody heard of.

In last issue's editorial, guest Horace O. Simmons recommended that Forth users promote Forth in non-Forth journals. That's an excellent idea. How about promoting it in our own?

John H. Lee, Lt. USN

QuikFind Addendum

Dear Editor,

While browsing through my article ("QuikFind String Search," *FD* XIII/4), I noticed a couple of errors and (heaven forbid) an error in the code listing. Here is a list of corrections:

1. Page 21, hash algorithms figure. Captions reads, "...After each XOR, the bits in e index," which should be ""...After each XOR, the bits in the index...."
2. Page 23, definition of HASH reads:

```
D177 ( magic seed )
SWAP
COUNT 1F AND
```

but should read as:

```
COUNT 1F AND
```

```
D177 ( magic seed )
SWAP
```

3. Page 24, fig-Forth to botForth definitions, reads:

```
: ENDIF ( sys -- )
  0 \ LITERAL
  \ DO ; IMMEDIATE
```

but should read:

```
: ENDIF ( sys -- )
  \ THEN ; IMMEDIATE
: FOR (sys -- )
  0 \ LITERAL
  \ DO ; IMMEDIATE
```

Also, there is no definition for @+. It can be extracted from the definition for C@+. In a fig-Forth system, I believe the equivalent for RECURSIVE would be to SMUDGE the latest definition, since SMUDGE merely toggles a bit. In my definition of : (colon), I preceded RECURSIVE with a \ since it is an immediate word in botForth.

Since writing this article, I have another another word, DICTIONARY, which creates an instance of a hash table. This allows multiple hash tables to be created. Also, the hash tables are dynamic. They initially occupy no RAM but, as entries are added to them, they grow geometrically to accommodate the number of entries. EMPTY empties the table and returns the used memory.

Another addition I have found very useful is the word ADJUNCT. This works just like QUIKFIND except it returns an entry in a parallel table where additional information may be stored about the string. Thus, you can associate a string with a block of code, another string, or whatever.

If there is interest, I could publish the updated version of QuikFind. Right now, I am using it to build a translator which allows phrase definitions in Forth, instead of just words. Hopefully, more on that later.

Rob Chapman

(PCYerk Listing Four, continued.)

```
\ exec-obj
\ This fellow expects an object pointer (in vars segment) atop
\ the object stack and a method # atop the methods stack.
\ Executes an object's method
: exec-obj  ( -- )
  \ Find the method's executable code
  find-method-code
  domethod
;

\ instantiate
\ addr1 is current pointer in var seg
\ addr2 is object's token pointer
\ Stores that pointer in the
\ variable segment, then allocates ivars space.
: instantiate     ( addr1 addr2 -- )
  dup                       \ Make copies of token pointer
  dup ,                         \ Store token pointer in var seg
  @t                            \ Fetch ivar space
  allot                         \ Allocate variable storage
  6 + @t                        \ Fetch startup method
  dup -1 <>                     \ Anything there?
  if  mpush                     \ Push method
            opush               \ Push object
            exec-obj            \ Execute stuff
  else
            2drop               \ Drop -1 and object pointer
  endif
;

\ **
\ ** Class definition
\ **
\ The contents of a defined class are:
\    Token segment:       Vars segment:
\    [  Ivars space ]<---[ token ptr  ]
\    [  Meth list   ]
\    [  Super ptr   ]
\      [  start meth  ]
\    [ ..tokens     ]
\ Note that the code following does> can do a @ and
\ retrieve the offset into token space for the class
\ definition structure.
\
\ Once instantiated, an object looks like this:
\     Token segment:     Vars segment:
\     [  (;code)    ]    [  token ptr  ]   <<<< To parent class
\     [  here-c     ]    [ ...ivars    ]
\     [  ..tokens   ]
: :class
  0 curr_meth_tail !                \ No methods yet
  clear-ivar_seg                    \ No instance variables
  create                            \ Build the name field
      here-t dup curr_class_off !   \ Set current class
      ,                             \ Build pointer in vars seg.
      0 ,t                          \ Size of ivars region
      0 ,t                          \ Pointer to list of methods
      0 ,t                          \ Pointer to superclass
      -1 ,t                         \ Initial method
  does>
      @                             \ Fetch token pointer
      here swap                     \ Get current object pointer
      create                        \ Make a header
            instantiate             \ Instantiate the object
```

```
          immediate              \ Make the object immediate
      does>
          opush                  \ Get object ptr. on ostack
          state@
          if                     \ We are compiling
               compile (lit)     \ Compile obj ptr. as literal
               ostack->dstack    \ Get object pointer
               ,t                \ There's the pointer
               compile opush     \ Compile an object push
               compile (lit)     \ Another literal is
                                 \ method code pointer
                find-method-code  \ Get method's code pointer
                ,t                \ Compile that
                compile domethod  \ Code to execute method
                odrop             \ Don't need object anymore
          else                    \ We are interpreting
                exec-obj          \ Execute the object
          endif
;

\ Complete a class definition
: ;class
  clear-ivar_seg                  \ No ivars segment
  0 curr_class_off !              \ No current class
;

\ Special word that returns current object so object
\ can send a message to itself.  Use 'self' inside
\ the methods definitions to refer to the current object.
: self      ( -- )
  compile (lit)
  curr_class_off opush                  \ Get current object
  find-method-code                      \ Locate method code
  ,t                                    \ Store as literal
  compile odup                          \ Dup object
  compile domethod                      \ Execute method
  odrop                                 \ Clear object stack
; immediate

\ Define a class's super class.
\ A class will inherit instance variable space, methods, and
\ startup methods from the super class.  A class can override
\ methods and startup methods.
: <super
  \ Find the object and resolve code address to token address
  blword find 0=
  if  abort" Super object not found"
  endif
  >body @ dup
  \ Store token address into super pointer of current class
  curr_class_off @ 4 +
  !t
  \ Copy ivars into local ivars
  dup @t curr_class_off @ !t
  \ Copy initial method
  6 + @t curr_class_off @ 6 + !t
;

\ Define initialization method.
\ This routine expects a method id on the top of the method
\ stack.  It stores that method id as the object's startup
\ method.
```

(Listing continues.)

Attention Forth Authors!

Author Recognition Program

To recognize and reward authors of Forth-related articles, the Forth Interest Group (FIG) has adopted the following Author Recognition Program.

Articles

The author of any Forth-related article published in a periodical or in the proceedings of a non-Forth conference is awarded one year's membership in the Forth Interest Group, subject to these conditions:

a. The membership awarded is for the membership year following the one during which the article was published.
b. Only one membership per person is awarded in any year, regardless of the number of articles the person published in that year.
c. The article's length must be one page or more in the magazine in which it appeared.
d. The author must submit the printed article (photocopies are accepted) to the Forth Interest Group, including identification of the magazine and issue in which it appeared, within sixty days of publication. In return, the author will be sent a coupon good for the following year's membership.
e. If the original article was published in a language other than English, the article must be accompanied

(Continues on next page.)

by an Engish translation or summary.

Letters to the Editor

Letters to the editor are, in effect, short articles, and so deserve recognition. The author of a Forth-related letter to an editor published in any magazine except *Forth Dimensions* is awarded $10 credit toward FIG membership dues, subject to these conditions:

a. The credit applies only to membership dues for the membership year following the one in which the letter was published.
b. The maximum award in any year to one person will not exceed the full cost of the FIG membership dues for the following year.
c. The author must submit to the Forth Interest Group a photocopy of the printed letter, including identification of the magazine and issue in which it appeared, within sixty days of publication. A coupon worth $10 toward the following year's membership will then be sent to the author.
d. If the original letter was published in a language other than English, the letter must be accompanied by an English translation or summary.

(PCYerk Listing Four, continued.)

```
: <<init-method         ( -- )
  mpop
  curr_class_off @ 6 +
  !t
;

\ **********************
\ ** DEFERRED BINDING **
\ **********************
\ Deferred binding allow you to specify the object at runtime,
\ rather than at compile time.

\ { Starts deferred binding.  He assumes there's a method # on
\ top of the method stack.  He copies that as a literal into
\ inline code (along with an mpush).
: {
   state@
   if                                   \ We are compiling
           compile (lit)                \ Compile literal
           mpop ,t                      \ Get method #
           compile mpush                \ Compile mpush code
     endif                           \ Interpreting--do nothing
; immediate

\ } Concludes a deferred method.  He assumes there will be
\ (at runtime) a method # on top of the method stack and an
\ object pointer atop the data stack.  He pushes the object
\ pointer onto the object stack, finds the method, and executes
\ it.
: }
  opush                                 \ Push object pointer
  exec-obj                              \ Execute it
;

\ ***********************
\ ** PATCHES AND MISC. **
\ ***********************
\ Following code is the patch to interpret.
\ Allows system to recognize methods and instance variables.
\ NOTE: When we get here, literal? has left 2 zeros on stack.
\ For uniformity's sake...we pass them on along.
: <interp-patch>
  2drop                                 \ Clear stack
  \ See if the item in question is a method.  If so, leave the
  \ method id # on the method stack
  here methname-find dup
  -1 <>
  if  mpush exit                        \ Push the method #
  else      drop
  endif

  \ Not a method -- see if it's an ivar
  here search-ivar dup
  -1 <>
  if  ?comp                          \ GOTTA be compiling
            compile (lit)            \ Compile ivar value
            ,t
            compile (do-ivar)        \ Compile ivar handler
            exit
  else      drop
  endif
```

(Listing continues on page 21.)

News from the Forth RoundTable

Best of GEnie

What is this language, Forth?

Gary Smith
Little Rock, Arkansas

Yes, there is an ANS Forth in the process of being drafted. Yes, the Technical Committee has labored long and hard in its collective attempt to meet the conflicting demands of minimalist versus maximalist, desktop user versus embedded-system implementor. Yes, many compromises have been arrived at and many ambiguities removed from the BASIS as it winds ever closer to becoming not only X3J14 BASIS.xxx, but the final draft proposal manifest we all look forward to. *[See dpANS Forth announcement, page 4.]*

As was pointed out in my last column—via exchanges gleaned from GEnie Forth RoundTable Category 10, Topic 25—several questions are still being debated. In this issue, we examine discussions in Category 10, Topic 12, "X3J14 Holding Pattern," to discover that even the question, "What exactly is this language, Forth?" is subject to heated discussion. Maybe, when the dust has settled, we will discover the ultimate truth that Forth is an attitude and has nothing to do with standardization.

Read on...

Category 10: Forth Standards

From: Doug Philips
Re: Architecture and Implementation

John Wavrik writes:

"The ANSI team has apparently not only invented a new language, but also a new concept in computer science: a language that manipulates data structures in a functional way but does not allow us to know what the data structures are. Sure doesn't sound like a good idea, does it? Certainly isn't a tested idea, is it?"

Oh, come on now, X3J14 didn't do this first, X3J11 did it, and they probably weren't even the first! How big is an integer (cell)? Implementation defined, guaranteed to be at least n bits. How big is a long (2cell)? Implementation defined, guaranteed to be at least m bits and $m \geq n$. I will admit that one needs to know something about the size of things (not structure!), so that, say, `' foobar !` will work (or not). Do I need to know anything about what a `'`-execution-token really is? No. All I need to know is the set of operators that take one (or more) as arguments and the set that can produce them. I believe the technical term is "abstract data type." Can you do arithmetic on a `'`-execution-token? Yes, but it will not be portable. As the standard is concerned with portability, it will not allow such action in a conforming program.

"If one is to limit the extensibility of Forth and rely upon vendor-supplied standard operators, then a great number of them must be supplied in the hopes of meeting as many needs as possible. Words like `COMPILE`, and `START:` become extremely important as an attempt to rescue some of the functionality of classical Forth. Even then, one typically finds that the supplied operators do not do exactly what is needed. Sounds exactly like the trap that most conventional languages have fallen into, doesn't it? And Forth did have a viable solution, didn't it? And the ANSI team is proposing a language that ignores this solution, isn't it?"

Straw argument. If Forth had already had *viable* and *portable* solutions, there would be no "hard work" to doing an ANSI standard. (Nor, perhaps, a need to do one at all.)

"Not only is it not easy to tell, without extensive testing, whether sufficiently many operators have been added—but there is the very real problem of making sure that they have been specified clearly."

This is very important, I will agree.

"I mention this word because it is one in which deviant implementations have already appeared. There have been a host of messages in this newsgroup pointing out that some of my examples using `START:` do not work on other trial implementations. All I can say is that I consulted the author before implementing mine. Incidentally, I don't think this will be unusual—I think that as more implementations of the proposed ANSI Forth appear, more deviations will appear. It is almost an inevitable consequence of trying to specify operators while being fuzzy about what they operate on."

Funny, I thought that was just the natural result of using English. And of the fact that any group, having concentrated on something for as long as any of the ANSI Technical Committees [TCs] do, will come to an understanding that is not always transcribed in the first pass or two. In fact, ANSI takes into account that it may not get completely clarified until after the standard is adopted. At that point, an official "request for interpretation" can be submitted. I'm not totally up on my procedure here, but the answer is probably binding on the standard (could someone from the TC spell this out in painstaking detail for me, please?). Yes, it would be better if that never had to be done. Better still is a plan to handle corrections.

Useful things that can't be done demonstrate weakness in the standard...

"It's a bit like a car trip: if a wrong turn was taken somewhere, should we just say 'It's history, we can't change it'; or do we do what most sensible people do: get back on the right road?"

"I think that it is a truly unwise strategy for the ANSI team to propose a new language and then use strong arm tactics to get its acceptance rushed through. It will do a great deal of harm for the survival of Forth to accept a bad standard—and I don't think anyone should regard it as 'fate' that we must do so."

Indeed. Make up your mind. How can ANSI "get back on the right road" if it is charged with codifying existing practice (wrong turns)? As soon as it does, it takes a turn never before taken. As far as "rushing," they haven't even gotten to the first public review yet! What we've seen so far is a rather open window into what has before been a closed process. (It is said that those who like sausages and politics should not watch either being made. The same could be said for standards.)

—Doug

From: John Wavrik
Re: X3J14 Holding Pattern Here

Greg Bailey writes,

"With all due respect, I find myself disappointed with Dr. Wavrik's posting of 19 Aug. 91 entitled, 'General Response to E. Rather and G. Bailey.' As carefully as I read it, I do not see that it is germane to most of the points in my posting of 16 Aug."

I hope that by now Mr. Bailey has had a chance to read the more specific response to his Aug. 16 posting, which I posted a few days ago. It does take some time for messages to travel from UseNet to GEnie—and I think we'd all benefit by having a chance to read, think about, and make careful responses. Generally, I find that it isn't a good idea for me to post an immediate response to a controversial topic—it seems better to think things over and edit my first draft. I apologize for the delay of a day or two in responding.

"I was hoping that Dr. Wavrik would admit to the existence of tradeoffs and to the fact that the work of X3J14 has economic implications beyond the performance of Forth in popularity contests."

Here, as in other places in his messages, Mr. Bailey has a tendency to put words in my mouth which I have never spoken (and which correspond to thoughts I am not thinking). There is no "popularity contest" involved here—just hard, economic reality. My living for the past ten years, at least, has been directly connected with my use of Forth as a tool. I intend to keep using Forth to make my living. My interest in a good standard for Forth is very definitely connected with my livelihood. Acceptance of Forth in universities, colleges, and many parts of industry will depend on whether a good standard—guaranteeing both power and portability—is produced.

"Therefore, I ask again. If some particular single one of Chuck's implementations is 'brilliance' and 'genius,' then what of all his others that differed, most notably the Novix chip (stacks not part of addressable memory, memory cell-addressed, 'reducing architectural features to the lowest common denominator' (Chuck has, in my experience, always advocated assumption of only positive divisors in signed division), and so on."

I did answer comments along this line in my specific response to Mr. Bailey. In sum, Charles Moore is working in a special environment. His interest is in hardware applications, and his work does not require portability. He can assume only positive divisors in addition, for example, because that is all that occur in his work. Others of us work in environments in which portability is very important (and in which negative divisors do occur).

"I submit that Chuck's particular genius has always lain in his uniquely clear insight about the simplest solution to the most challenging part of any problem. I further submit that I've never seen any evidence that a single architecture/implementation frozen for all time was anywhere on Chuck's agenda. Is this 'dissonance' so disturbing to Dr. Wavrik that he feels he must 'correct' it by attributing to Chuck the notion that the immutability of the architecture is more important than the solution of problems?"

Here again, Mr. Bailey seems intent on putting words in my mouth. I perceive no dissonance, nor am I correcting Charles Moore. I agree with Greg Bailey that Chuck's gift is coming up with good simple solutions to problems. Given the nature of Chuck's work, he would want to experiment with very low-level changes to his systems. Others of us are solving very different types of problems.

Suppose the problem is to produce a language that is tremendously powerful and flexible, yet will allow code to run correctly on many platforms. What would be the simplest possible solution to that problem?

I think if you will look back at all I have written, you will find that I have only (and consistently for the past several years) made the following observation: It could well be that the simplest solution is to agree on the architecture of an abstract machine (perhaps, if necessary, making separate but overlapping standards for a few different types of architecture).

"Dr. Wavrik, I ask that you re-read my earlier posting, compare it with your reply, and see if you don't agree with me that your posting has frustratingly little to do with the issues raised."

No, I think the specific reply I gave you addresses the issues quite well. The reposting of the "Architecture vs. Implementation" paper was only intended to eliminate some apparent confusion.

"Forth is not the result of slavish pursuit of 'symmetry,' and portable power of the sort your paper seems to assert is essential."

Again, words are being put in my mouth. I said nothing about symmetry—although I do think that slavish pursuit of simplicity might be worth trying.

"Do you seriously propose that your definition of power (portable hacking) be given absolute precedence over other definitions of power (practical usefulness for demanding applications, for example) that have characterized most of the dramatic successes of Forth that I am aware of?"

Again, words are being put in my mouth. Portable hacking is not my definition of power. Power, for me, is the ability to accomplish difficult things without fighting the language. Forth is the only language I've ever used where I feel that I can conceive of what needs to be done, and Forth will allow me to do it. Most languages

(PCYerk Listing Four, continued from page 18.)

```
  0 0                                \ Look like literal?
  \ Let do-undefined handle things
  do-undefined
;

\ Following code patches interpret.  Do it AFTER you've
\ allocated methods and variable segments
: patch-interpret
  ['] <interp-patch>
  ['] interpret >body 40 + !t
;

\ Put interpret back the way it was.
: unpatch-interpret
  ['] do-undefined
  ['] interpret >body 40 + !t
;

\ Initialize the system.
\ One you've included [i.e., loaded] this code, you must
\ execute "start-objects" before you can begin defining
\ any objects.  When you're done defining and calling all
\ your objects [i.e., you're about to make an executable],
\ execute "end-objects".
: start-objects
  alloc-ivar_seg              \ Allocate ivars segment
  clear-ivar_seg
  alloc-methname_seg          \ Allocate method name segment
  clear-methname_seg
  1 curr_meth# !              \ Start method #'s
  patch-interpret             \ Fix interpret
  clear-o&mstacks             \ Initialize the stacks
;

\ Clean things up
: end-objects
  unpatch-interpret                 \ Put interpret back
  ivar_seg @ free                   \ Ditch ivars segment
  methname_seg @ free               \ Ditch method name segment
;
```

(End listing. Next issue contains code for basic & storage classes, byte & word arrays, strings, & string arrays.)

require me to fight them to shape their rigid features to match the problem (and sometimes they are so unsuitable that I can't realistically do the task).

Power in Forth comes, in great measure, from the user's ability to understand how the system works—and being able to harness that understanding.

We are both in agreement that power has something to do with practical usefulness for demanding applications—my demanding applications as well as your demanding applications.

"You may feel, for example, that performance is no longer relevant, as you have posted."

Again, words are being put in my mouth. What I said is that language performance is no longer measured entirely in terms of execution speed.

I regard Forth as a "high performance language" in my area because it facilitates the development and modification of programs. I can still get close to the speed of compiled code by heavy use of assembly language (which I do when a system has become stabilized)—but, really, high-level Forth running on a microcomputer is no match in speed for the output of a good C compiler. It would be foolish to give up the attributes which make Forth a high performance language (in terms of ease of development, power, flexibility) to achieve marginal gains in execution speed.

Hang around a university for a while—people don't talk about how to write clever, tight code these days. The problem is writing and maintaining large programs that do powerful things and run correctly.

John J Wavrik
jjwavrik@ucsd.edu
Dept of Math C-012
University of California, San Diego
La Jolla, CA 92093

From: Elizabeth Rather

J. Wavrik writes:

> "Both Greg Bailey's and Elizabeth Rather's comments illustrate the fact that there are also people in the Forth community for whom reusability of code is not important—and who alter their systems down to the lowest level for each new application."

John, you're seriously distorting the point of Greg's and my remarks. We are challenging your continuing assertion that there is such a thing as "traditional" Forth from which the world has been deviating and which ANS Forth is deprecating. Our discussion of deviations from the earliest days to the present is intended to point out that there has never been such a golden age, and that your nostalgia for it is, therefore, inappropriate.

Greg and I and the entire committee are extremely concerned with portability of application code, as well as "programmer portability" (the ability of programmers to move from one system to another, preserving both sanity and competence without massive new learning curves). Why else do you think we have invested so heavily in the standards effort?

We hope and believe that the steps we are taking will improve Forth in both these respects.

> "Production of code has become an extremely expensive affair—I think it is more typical these days to find people who can't afford to throw away the kind of time and effort needed just for a marginal gain in execution speed—and I think you can find as many of them in industry as in academia."

Once again, you are mistaken if you think we disagree. Our objective in describing Forth behaviorally rather than by constraining implementation choices is to permit implementors to provide an internally optimized (and hence fast) system whose surface, as presented to the application program, offers a very high degree of portability due to its conformance to rigorously defined behaviors.

> "A major factor, however, is that people who do not need portability also do not need a standard."

How do you reconcile this with your continuing assertions that the members of the TC don't care about portability? Do you contend that these people have spent tens of thousands of dollars and a lot of their billable hours over a period of years to do something they don't need or want?

The disagreement between you and the committee is not over who wants portability, but *how portability is achieved.* We believe it can most usefully be achieved by defining the behavior of Forth words, and you'd prefer to see their implementation standardized. This is a simple disagreement, which is okay, but the discussion will be advanced most usefully if you direct your comments to that rather than spurious assertions about mythical traditions and the motives of the TC members.

> "Simplicity, comprehensibility, being supplied with source code, ability to reproduce the system are among the things I lump under the heading 'glass box.' If anyone undertakes to write a standard for Forth, these are exactly the qualities which need to be made portable."

Simplicity and comprehensibility sound great. No argument.

(GEnie continued on page 28.)

Object-Oriented Forth

Roger Bicknell
New Westminster, British Columbia, Canada

In the beginning, there was programming. Now, the programming was formless and unstructured—darkness was over the surface of the design problem.

Then came structured programming—a disciplined coding style and a logical analysis technique which emphasized the nature of the coupling between code modules and the design reasons behind the creation of the code modules. Creating a routine out of code that just happened to be performed at the same time (temporal cohesion) was out. Creating one routine out of code that did a couple of closely related things (logical cohesion) was out—it necessitated passing control flags between routines (called *control coupling*, oddly enough). There was (once again?) the dawning realization that a routine should do one thing and only one thing (and do it well). However, there was more to be discovered.

If a careful study of procedures proved beneficial, then what about a close look at the nature and form of the data being operated upon by those newly-structured routines? Thus, someone coined the phrase *object oriented*, probably in contrast to the prevailing procedure-oriented programming of the time.

Enter Analysis of Data

Data can be kept in atomic types like variables, constants, and literals; or in molecular groupings, such as named records and indexed arrays. Many early computer languages provided a small selection of atomic data types and expected you to use records, for example, to simulate any different types of data you might fancy. The ability to abstract data, or to create a new data type, is an essential feature of an object-oriented language implementation.

Information hiding is the backbone of code security, reliable re-entrancy, and data abstraction

Roger Bicknell is an electrical engineer who has programmed in Forth as a hobby since 1982. He enjoys the simplicity and interactivity of the language, and uses it to experiment with language design. He says, "Some people gamble, I program... Yes, I know it's 4:00 a.m., but I've just got to tweak this one last word..." Roger welcomes feedback at 315 Devoy Street, New Westminster, BC, Canada V3L 4E8 or at R.BICKNELL2 on GEnie.

But it is not simply for the prettiness of being able to refer to a new data type like `COMPLEX` (say, made up of two float-type variables called `REAL` and `IMAGINARY`) that data abstraction is important. A procedure's code is necessarily specific about what type of data it operates upon. The code of + (plus) assumes an integer data type, and `F+` (f-plus) assumes a floating-point data type. These cannot be swapped (or even duped—else your employment may be over). Thus, it only makes sense to group the definition of a data type with each and every procedure that will operate upon it—making for easier maintenance. When a data type's definition and its associated procedures are grouped into one module, further de-coupling between modules can be realized. For example, a stack can be implemented using an indexed array, or a buffer and an offset. Allowing any other module to know the implementation details is not necessary and could allow direct access to elements within the stack structure. This direct access is a potential problem for the future, if ever the implementation of stack structure is changed. "What they don't know can't hurt you." Of course, it is important to keep in mind that considerations like maintenance are far more complex and important for a large multi-programmer project than for a lone-wolf, one-nighter program. You needn't hide anything from yourself, but you should protect your code from what those other crazy, reckless, undisciplined hacks might do.

It is possible to de-couple to the extent of not even allowing direct reference to a data type's routines. Instead, a message is sent over to that module *asking* for a procedure to be enacted. Thus, even the data type's procedures could be modified (which might be necessary if the details of the data change), and still it would not affect any other module. The intent here is to provide robust code by limiting side effects of changes, and to provide reusable code by de-coupling it from as much as possible.

The intent of object-oriented programming is to extend the ideas of structured programming to include techniques dealing with data which enhance reliability and which minimize the maintenance of a software project by reducing the amount of modifications necessitated by a change.

Glossary

In the object-oriented vernacular, a data structure is called an *object* or an *instance* of a *class*. The class contains the information necessary to construct an instance, as well as all the routines that operate on its data type. The internal objects that make up an instance (like fields within a record) are called *instance variables*. An object is sent a *message*, which specifies what to do, but not how to do it. The object then finds the *method*, the word that performs the correct action; this is called *binding* the message to the object. There are two forms: *early* binding and *late* (or *dynamic*) binding. Early binding is done at compile time and dynamic binding is done at run time. The ability of one class to gain access to another class's methods is called *inheritance.*

Defining an Object-Oriented Language

An object-oriented programming language can be described as having at least four features: data abstraction, information hiding, dynamic binding, and inheritance.

Data abstraction is the ability to create new data types. Of course, common Forth already has this. The `CREATE DOES>` team is all that is required to invent any new data type. For example, while standard Forth does not specify array-type data, it is a simple task to implement such.

Information hiding is the backbone behind code security, reliable re-entrancy, and data abstraction. Information hiding is necessary in order to provide more than one context in which to interpret a name. Local variables are an example of information hiding, because they are unavailable to any routines other than the one in which they are defined. Of course, one way to hide things in Forth is to put them into a separate vocabulary. The most obvious reason to hide a word in Forth is to be able to have more than one word with the same name, but the concept of information hiding goes a little deeper than that. One important tool in providing code security and reducing interdependencies between software modules is hiding details of implementation. As in the example given above, whether a stack uses an offset to point into a buffer or uses an index into an array should not be known or exploited by other software modules—because if this implementation should ever change, the exploitive code will probably break. Thus, information hiding is required if one is to provide reliable, modifiable code modules.

Figure One. Extending common Forth to OOF.

```
\ oofinitl.fth   910808 rwb

only forth also definitions

4 constant cell  \ cell is #bytes in stack width

: cell+         cell + ;
: cell-         cell - ;
: cells         cell * ;

: struct:       \ ( -- offset )
        0 ;

: :field        \ ( offset /field -- offset' )
    create
        over , +
    does>       \ ( 'struct pfa -- 'field )
        @ + ;

: ;struct       \ ( /struct -- )
        constant ;

cr .( oofinitl loaded ) cr
```

Dynamic binding is simply the ability to decide upon the appropriate method of implementing an action at run time rather than at compile time. This can become important when an object's class is unknown at compile time. This feature has been exploited in the definitions of `PUSH` and `POP` in the `STACK` class code. (See Figure Eight, page 30.) Also, a stack can be made out of any new class of object, and `PUSH` cannot know ahead of time the new method's CFA for storing the object in the stack. Dynamic binding is not always necessary, but when it is needed, it is indispensable.

If a new object class is very similar to an existing class, it may be economical for the new class to use some of the other's methods, rather than rewriting them. This is the concept of inheritance—allowing a newly defined class to inherit some (or all) of the methods of an existing class. Inheritance allows the creation of a new class by merely defining the difference between the new class and a previously defined class—thus, a lot of code can be reused and time saved.

File-loading sequence.

```
\ oof.fth        910729 rwb

: task  ;

fload oofinitl.fth
fload oofclass.fth
fload oofmssag.fth
fload oofobjec.fth
fload oofprima.fth
fload oofcmplx.fth
fload oofarray.fth
fload oofstac2.fth

cr .( oof loaded ) cr
```

Commenting Style

I should first comment on my commenting and naming style. Consider the comment for the word `CLASS:` in Figure Two. The <name> token appearing before the stack comment refers to the input stream argument that `CLASS:` requires. Also, I like to preface a stack comment entry with `'` (tick), `/` (forward-slash), or `^` (caret). I use tick to mean "address of," forward-slash to mean "size of," and caret to mean "contains the address of." Thus `'/body` in the stack comment means "the address of the size of body." Also note that I like to preface `STRUCT:` fields with + (plus). I just do. Forgive me.

Code Description

Figure One defines some words, additions to common Forth, that are used in the definition of object-oriented Forth (OOF). `CELL` is set to the width of Forth's stack—

Figure Two. Object class words.

```
\ oofclass.fth          910729 rwb

only forth also definitions

#vocs cells constant /context

struct:
                #threads cells  :field +threads
                /context        :field +context
                cell            :field +/body
;struct /class

: >context    \ ( 'class -- ) MACRO to select object's context.
        +context context /context cmove ;

: class:        \ <name> ( -- '/body offset )
    vocabulary
        last @ name> >body >user        \ ( -- 'body )
        /context cell+ ualloc drop      \ allocate space for
                                        \ rest of body
        ( 'body ) dup context !         \ setup class to be
                                        \ both context..
        context over +context /context
                                \ ( -- 'body 'cxt 'c.cxt /cxt )
        cmove definitions       \ ( -- 'body ) ..and current vocabs.
        +/body dup off 0        \ ( -- '/body offset )
    does>   ( pfa -- )
        >user >context ;

: END           \ ( -- )
        only forth also definitions ;

: ;class        \ ( '/body offset -- )
        END  swap ! ;

: METHODS       definitions ;

cr .( oofclass.fth loaded ) cr
```

my Forthmacs (by Mitch Bradley) for the Atari ST is a 32-bit Forth, so my CELL is set to four bytes. Adjust yours accordingly. A simple method for grouping data into named records is provided by the three words STRUCT:, :FIELD, and ;STRUCT.

Common Forth contains the seeds of an object-oriented language. CREATE DOES> provides the ability to abstract data and create new data types. Vocabularies can provide privacy, as well as inheritance. Thus, only ten new words need be used to program in OOF: CLASS:, ;CLASS, INHERIT, METHODS, END, MESSAGE, OBJECT, OBJECTS, :VAR, and :VARS.

Figure Two provides definitions of object class words. I have implemented a CLASS as a hybrid of STRUCT: and VOCABULARY—which reflects the twofold nature of a class: to provide the internal data structure of its type, and to house the data's routines. See Figure Seven (page 29) for a good example of how these words are used. (In Forthmacs, the parameter field of a vocabulary is kept in the user area. So, while the general idea is to build a vocabulary with two extra fields—the +context field and the +/body field—these must be allocated in the user area for Forthmacs.)

MESSAGE is defined in Figure Three. A message merely records its name as a string within its body, and then tries to find it in the context vocabulary (CLASS) at run time. (This is an example of late binding.) Note that all messages are prefaced with a < (less-than) character. This is done so that the message and the method will not be confused. The < is stripped off before being compiled within the message with the SWAP 1+ SWAP 1- code.

Figure Four contains words which construct instance OBJECTs and instance VARiables. An instance contains two fields: its class pointer and its body. An instance OBJECT's body contains its instance variables. An instance VARiable's body contains its offset within its parent OBJECT.

Figure Five (pg. 28) contains the initial bootstrapped object class, called PRIMARY. I decided that PRIMARY methods should just go in the FORTH vocabulary, so that PRIMARY would not have to be INHERITed by each class; thus, the phrase PRIMARY METHODS has been commented out. The only real need for a PRIMARY class is for indirect reference to the object on top of the stack. This data must be a declared class. PRIMARY is declared just for this situation. Other than that, its "object-ness" may be ignored and it can be treated just like a Forth variable or structure field.

Figure Six (page 28) is a simple example of building new classes. Note that the methods' code is very Forth-ish: the implicit stack operators (DUP, SWAP, TUCK, etc.) are still needed, parameters are still passed on the stack, and RPN syntax is still used—thus, OOF blends well with common Forth. It is not necessary for 2COMPLEX to explicitly INHERIT from the COMPLEX class in order for the methods (like @@) to pass the <@@ message along. In this case, the instance variable X will accept the <@@ message and interpret it correctly. *Inheritance* becomes an issue when a class that is a specialization of another class, wishes to use some of the other class's methods. For example, if one had both AUTOMOBILE class and CHEVY class, there may be AUTOMOBILE methods that are applicable to the CHEVY class objects. By simply stating AUTOMOBILE INHERIT before defining CHEVY class, it inherits all the methods of AUTOMOBILE. I used this feature to provide STACK objects with the <LENGTH operator, by inheriting it from the ARRAY class.

Figure Seven (page 29) is an example of defining an ARRAY class of objects. The #EL instance variable contains the number of elements in the array. The ^EL-CLASS variable points to the class of the array's elements. Consider the INDEX method: the last thing that must be done, after deriving the address of the indexed element within the array, is to switch the class context to that of the elements so subsequent messages will be bound to the correct instance type (and, thus, the correct method will be executed). The words OBJECTS and :VARS in Figure Four assume that the first two instance variables within a group-type object (like ARRAYs, STACKs, MATRICEs, QUEUEs, etc.) will be #EL and ^EL-CLASS, respectively.

Figure Eight (page 30) gives an example of INHERITing a class—as STACK is a specialization (and superset) of ARRAY class. Note that the class context must be switched to that of the stack's elements just before the message is sent to fetch or store the element object in both PUSH and POP.

One kludge I wanted to avoid in defining grouped objects (like ARRAYs and STACKs) was the need to predetermine the size of the body in the class definition. This would either cause all the ARRAYs (or STACKs) to have the same number of elements, or else necessitate a new defining word for creating objects for each new group-type class. Consider the following instantiation of a COMPLEX STACK.

```
STACK OBJECT FRED
23 COMPLEX OBJECTS
```

Thus, FRED is defined as the object at the head of the group of elements, with the defining word OBJECT. Then 23 complex-type objects were allotted. OBJECTS is capable of patching the previously created word (in this case, FRED) with the number of elements allotted into FRED's #EL instance variable.

The benefit of an object-oriented implementation's security can be seen by comparing the code of Figure Eight with that of Figure Nine (page 31). Both are implementations of stack-type data, but are radically different. Note that code using STACK would not break if STACK were changed—due to being forced to use *only* procedures within the STACK class's code-definition module.

Figure Three. Defining MESSAGE.

```
\ oofmssag.fth          910729 rwb

: >in@          ( -- ) \ macro to save input stream pointer.
        in-file @ ftell ;

: >in!          ( -- ) \ macro to set input stream pointer.
        in-file @ fseek ;

: MESSAGE       \ <name> ( -- )
        >in@                 \ remember input stream location.
    create                   \ create message
        >in!                 \ put input stream pointer back.
      bl word count          \ grab <name> and compile string in
        swap 1+ swap 1- ",   \ pfa of message.
    does>       ( pfa -- ? )
        find
        if
                execute         \ effect method of action.
        else
                abort" method unknown"
        then
;

cr .( oofmssag.fth loaded ) cr
```

Conclusion

If an object-oriented programming language is defined by the characteristics of data abstraction, information hiding, dynamic binding, and inheritance, only a little needs to be added to Forth to make it so. In keeping with an RPN syntax, the process of binding a message is shared between the object and the message words. It may appear that this responsibility could have been wholly shifted to the message word, but only because this is a late-binding example. Because late binding has an associated run-time penalty (finding the correct method to execute), early binding is usually used except when late binding is required. It would take about two more words, and about five minutes of coding, to convert the given code to early binding (really!). This will be left as an exercise to the reader. (Oh! I've always wanted to say that!) Maybe I will update the code in an upcoming article... Also, one could optimize the code associated with the first instance variable reference. Because it has an offset of zero within the parent object, it is wasteful at run time to add the offset.

Object-oriented Forth, as presented, is upwardly compatible with common Forth (Forth-83, specifically); thus, they can be intermixed at will. Programming in OOF promotes grouping all of a data class's routines, just below the declaration of the inner layout of the class. Also, it allows one to focus on objects and actions without constant regard to internal implementation details (one definition of PUSH works for any kind of STACK).

An object orientation is simply the coding disciplines specific to the expression of data, and which are complementary to those for procedures. The intent is to reduce maintenance by minimizing modifications caused by a change, and to increase productivity by enhancing reusability of existing code.

Figure Four. Words to create objects and instance variables.

```
\ oofobjec.fth          910730 rwb

: +objects       \ ( 'body -- 'objects ) MACRO
        dup cell- @              \ ( -- 'body 'el-class )
        +/body @ + ;

: /body          \ ( -- /body )
        context @ +/body @ ;

: do-object      \  ( /body -- )
        allot
    does>
        dup @ >context           \ select object's context.
        cell+                    \ ( -- 'body )
;

: object         \  <name> ( -- ) Create instance of class.
    create  context @ ,  /body do-object ;

: objects        \ ( n -- ) Create n instances of class.
        /body over               \ ( -- n /body n )
        last @ name> >body cell+ \ ( -- n /body n 'body )
        context @ over cell+     \ ( -- n /body n '#el 'elclass ^elclass )
        ! !                      \ ( -- n /body )
        * do-object ;

: do-var         \ ( offset /body -- offset' )
        over , +
    does>    ( pfa -- 'body )
        dup @ >context
        cell+ @ +        \ ( -- 'var )
;

: :var           \ <name> ( offset -- offset' ) Create instance variable.
    create  context @ ,  /body do-var ;

: :vars          \ ( offset n -- offset' ) Create n instance variables.
        /body over               \ ( -- offset n /body n )
        last @ name> >body cell+ \ ( -- offset n /body n 'body )
        context @ over cell+     \ ( -- offset n /body n '#el 'elclas ^elclass )
        ! !                      \ ( -- offset n /body )
        * do-var ;

cr .( oofobjec.fth loaded ) cr
```

Advertisers Index

On-Line Resource Updates

Two out-of-date items mistakenly crept into last issue's "reSource Listings." These are the Wetware Forth conference (under Unix BBS's) and the Cave (under non-Forth-specific BBS's with extensive Forth libraries). Please disregard both.

In France, the Forth BBS JEDI has ceased operating. Try Serveur Forth, which claims news, services for new programmers, Forth teaching material, and a file library. It supports up to 19200 baud, depending on access method (for full details about high-speed, Minitel, or alternate-carrier access, contact sysop Marc Petremann, REM Corp., 17 rue de la Lancette, F-75012 Paris, France). From within France via modem, call the following. (From Germany, add the telephone prefix 00 33. From other countries, use the prefix 33.)

(1) 41 08 11 75
300 baud (8N1) or
1200/75 E71
or:
(1) 41 08 11 11
1200 to 9600 baud (8N1)

The Programmer's Corner BBS in Maryland has a Forth message area and a Forth file area. Call 401-596-1180 or 401-995-3744.

Additional non-Forth-specific BBS's with extensive Forth libraries:

- PDS*SIG
  San Jose, CA
  408-270-0250
  SprintNet node casjo
  StarLink node 6450
- Programmer's Corner
  Baltimore/Columbia, MD
  301-596-1180 or
  301-995-3744
  SprintNet node dcwas
  StarLink node 2262

(Note: PC-Pursuit is now SprintNet.)

Figure Five. The initial, "bootstrapped" class.

```
\       oofprima.fth    910723 rwb
  MESSAGE <@@
  MESSAGE <!!
  MESSAGE <??
  MESSAGE <init

class: PRIMARY
        cell+ \ bootstrap size of class
;class

\ PRIMARY METHODS
: @@            \ ( 'body -- primary )
        @ ;

: !!            \ ( primary 'body -- )
        ! ;

: ??            \ ( 'body -- )
        ? ;

: init          \ ( 'body -- )
        off ;
END

cr .( oofprima.fth loaded ) cr
```

Figure Six. Building new classes.

```
\ oofcmplx.fth          910728 rwb

only forth also definitions

\ MESSAGE <@@
\ MESSAGE <!!
\ MESSAGE <??
\ MESSAGE <init

class: COMPLEX
        PRIMARY :var real
        PRIMARY :var imag
;class

COMPLEX METHODS

: @@            \ ( 'body -- real imag )
        dup real @  swap imag @ ;

: !!            \ ( real imag 'body -- )
        tuck imag !  real ! ;

: ??            \ ( 'body -- )
        dup imag ?  real ? ;

: init          \ ( 'body -- )
        dup imag off  real off ;
END
class: 2COMPLEX
        COMPLEX :var x
        COMPLEX :var y
;class

2COMPLEX METHODS
```

(Figure continues.)

(GEnie from page 22.)

Regarding "being supplied with source code," two comments:

(a) Forth, Inc. supplies complete source code under license with all polyFORTHs, along with the ability to reproduce the system, as we believe these are important entitlements to those of our customers who do want to optimize their applications in the knowledge that they will be fairly transportable across polyFORTHs on other platforms, but harder to port to other Forths. Making this choice is their prerogative.

However, as you yourself point out, there are other people for whom the need for portability is paramount. The standard, also as you point out, is for those people. If the TC mandates that all conforming implementations not only follow a particular model but supply source and regeneration capability, the result will be *few* conforming implementations, and mediocre performance on those that do conform. It's hard to see how this benefits anyone.

This is why the TC believes the better way to facilitate portability is by standardizing behavior.

(b) The reality of the marketplace is that most of Forth, Inc.'s competitors do *not* supply source and regeneration capability, and they are nonetheless successful in their respective markets. This supports the conclusion that there are very many Forth programmers who don't find these things essential to their work.

In summary, I personally agree with you as to the value of source and regeneration capability, but emphatically do not agree that they should be mandated in a standard.

> "Since when are the two previous standards for Forth 'some particular model'?

Greg was only trying desperately to understand what on earth you do mean in invoking "traditional Forth," as you keep doing.

"I use 'Forth' to refer to the language as described in the books cited most often as references: *Starting Forth* and *Thinking Forth* by Leo Brodie, and *Forth: A Text and Reference* by Kelly and Spies."

At last, a workable definition! However, these fine books all make it very clear that, although they discuss such things as dictionary structure for pedagogic purposes, implementations do vary. Primarily, they define Forth behaviorally, just as ANS Forth does. I quote from Kelly & Spies (pg. 305–6):

"The Forth standards wisely make no attempt to define how the language works internally. The point of the standards is to promote a functional compatibility of programs, not to stifle original ways of adapting Forth to new hardware."

Couldn't have said it better myself.

"Several of the languages I have used... are described as 'functional' languages... Each of these languages is described in terms of a set of operators. In each case, however, the operators act on a specific data type or types... It is meaningless to have operators that do not operate on anything!

"The ANSI Team has apparently not only invented a new language, but also a new concept in computer science: a language that manipulates data structures in a functional way, but does not allow us to know what the data structures are. Sure doesn't sound like a good idea, does it? Certainly isn't a tested idea, is it?"

Can't offhand think of *any* languages that describe how their data structures are arranged in memory, let alone how their *code* is arranged in memory, which is what you seem to expect of Forth. ANS Forth pays a great deal of attention to describing data types, at least as clearly as C, etc. It also explicitly describes (Section 5.4 in BASIS, 3.4 in dpANS-2) the regions of memory that are addressable by a standard program. Most high-level languages don't let you address memory at all. C sort of does, via 'pointers,' but pointers are still a lot more abstract than Forth's addresses.

"...an attempt to rescue some of the functionality

(Figure Six, continued.)

```
: @@               \ ( 'body -- x y )
        dup >r  x <@@
            r>  y <@@ ;

: !!               \ ( x y 'body -- )
        dup >r  y <!!
            r>  x <!! ;

: ??               \ ( 'body -- )
        dup >r  y <??
            r>  x <?? ;

: init             \ ( 'body -- )
        dup >r  y <init
            r>  x <init ;
END

\ Example: 2COMPLEX OBJECT 2understand?
\ .. oh gosh NO! :^)

cr .( oofcmplx.fth loaded ) cr
```

Figure Seven. Defining an array class.

```
\ oofarray.fth    910723 rwb

  MESSAGE <index
  MESSAGE <length

class: ARRAY
        PRIMARY :var #el
        PRIMARY :var ^el-class
;class

ARRAY METHODS

: '/el-body     \ ( 'body -- '/el-body ) MACRO
        ^el-class @ +/body ;

: index          \ ( i 'body -- 'objects[i] )
        tuck ^el-class @ dup >r       \ ( -- 'body i 'el-class )
        +/body @ *                    \ ( -- 'body offset )
        swap +objects +               \ ( -- 'body[i] )
        r> >context ;

: length         \ ( 'body -- n )
        dup '/el-body @
        swap #el @ * ;
END

cr .( oofarray.fth loaded ) cr
```

of classical Forth... And Forth did have a viable solution, didn't it? And the ANSI team is proposing a language that ignores this solution, isn't it?"

We'd sure appreciate it if you'd share this "viable solution" with us, John. And please be specific, rather than vaguely alluding to "classical Forth," so we can consider your proposed language for incorporation.

We believe ANS Forth is extensible, and would very much like to know exactly what you feel is compromised. As you seem to have a high regard for precise language, we'll be grateful if you'd offer us some as an example.

Your discussion of START: would be helpful, except that Mitch has already told you that we agreed there was a problem with BASIS15's definition and fixed it. We'll look forward to seeing whether you agree that it is fixed in dpANS-2. There are probably a lot more areas in which clarity can be improved, and appreciate people pointing out other specific instances.

"I think it is a truly unwise strategy for the ANSI team to propose a new language and then use strong arm tactics to get its acceptance rushed through."

We have no intention of doing so, and couldn't if we did. The public review process is deliberately lengthy, in order to ensure as much feedback as possible.

"It will do a great deal of harm for the survival of Forth to accept a bad standard—and I don't think anyone should regard it as fate that we must do so."

We heartily agree. We look forward to hearing from lots of people in the public review process.

Figure Eight. Code for the stack class.

```
\ oofstack.fth          910723 rwb

\ MESSAGE <@@
\ MESSAGE <init
  MESSAGE <push
  MESSAGE <pop

ARRAY also

class: STACK
        PRIMARY :var #el
        PRIMARY :var ^el-class
        PRIMARY :var ^tos
;class

STACK METHODS

: push          \ ( obj 'body -- )
        dup ^tos @ swap                 \ ( -- obj 'tos 'body )
        '/el-body @ negate  \ ( -- obj 'tos 'body -/body )
        over ^tos           \ ( -- obj 'tos 'body -/body ^tos )
        +!                              \ ( -- obj 'tos 'body )
        ^el-class @ >context <!! ;

: pop           \ ( 'body -- obj )
        dup ^tos >r                     \ ( -- 'body )
        dup '/el-body @ >r              \ ( -- 'body )
        dup ^tos @                      \ ( -- 'body 'tos )
        swap ^el-class @ >context <@@   \ ( -- obj )
        r> r> +! ;

: init          \ ( 'body -- )
        dup ^tos >r                     \ ( -- 'body )
        dup +objects >r                 \ ( -- 'body )
        dup '/el-body @ >r              \ ( -- 'body )
        #el @ r> *                      \ ( -- length )
        r> +                            \ ( -- 'bos )
        r> ! ;

: @@            \ ( 'body -- obj )
        dup ^tos @                      \ ( -- 'body 'tos )
        swap ^el-class @ >context <@@ ; \ ( -- obj )
END

cr .( oofstack.fth loaded ) cr
```

From: Steve Geller

I used to use Forth, but got tired of my boss blaming all the software bugs on Forth (he's a Fortran and BASIC enthusiast). I now write most of my software in C and assembler.

The non-portability of Forth has annoyed me, because I work on a variety of environments: PC, Unix, VAX, Mac, and some embedded stuff. The chief annoyance was when a word with the same name did different things depending on the implementation. This is the main reason for standardization, in my view.

I think much of the squabble I read here will fade away once a standard is clearly defined—and widely implemented. I may well take another look at Forth when ANS Forth appears. I sure do like the consistency of C implementations; most of the problems I've hit were with small differences (or just plain bugs) in run-time library implementations.

Some argument centers on whether ANS Forth should codify existing practice or define a better language. The first idea seems rather reactionary. The present implementations are not going to disappear when ANS Forth appears; there will be a period of transition. If the standard is well defined, it will be accepted in the marketplace and everyone will be better for it. Variant Forths will be around forever. There are always "extensions" to any standard. F83 was full of extensions to the '83 stan-

Figure Nine.

```
\ OOFNEWST.FTH         910818 rwb

\ MESSAGE <@@
\ MESSAGE <init
  MESSAGE <push
  MESSAGE <pop

ARRAY also

class: STACK
        PRIMARY :var #el
        PRIMARY :var ^el-class
        PRIMARY :var head
;class

STACK METHODS

: element-context       \ ( 'body -- )
        ^el-class @ >context ;

: 'obj          \ ( 'body -- 'obj )
        dup +objects swap      \ ( -- 'objects 'body )
        dup #el @ over head @  \ ( -- 'objects 'body #el head )
        - swap '/el-body @     \ ( -- 'objects head' el-/body )
        * + ;

: full?         \ ( 'body -- f )
        dup head @ swap #el @ = ;

: empty?        \ ( 'body -- f )
        head @ 0= ;

: push          \ ( obj 'body -- )
        \ check not full
        \ incr head
        \ set element context
        \ store obj
        dup full? not                  \ ( -- obj 'body ~f )
        if
                1 over head +!         \ ( -- obj 'body )
                dup 'obj
            swap element-context       \ ( -- obj 'body )
                <!!
        else
                abort" Stack Full!"
        then ;

: pop           \ ( 'body -- obj )
        \ check not empty
        \ set element context
        \ fetch obj
        \ decr head ptr
        dup empty? not           \ ( -- 'body ~f )
        if
                dup >r  -1 >r    \ defer til after obj fetch.
                dup 'obj
            swap element-context
                <@@                    \ ( -- obj )
                r> r> head +!
        else
                abort" Stack Empty!"
        then ;
```

(Figure continues.)

dard, and became a *de facto* standard itself.

The question is really whether the ANS standard will be an attractive proposition to users and implementors. I should think it might be, given the background and caliber of people working on the committee. I am going to try to obtain a dpANS document whenever it becomes available to the general public.

From: John Wavrik
Subject: Traditional Forth

Elizabeth Rather writes,

> "Traditional Forth, for example, allows the user to know and make use of knowledge of what is 'compiled' (or, more accurately, assembled)—and to exercise total control over the process.
>
> "Hogwash! What on earth is this 'traditional Forth,' and what *did* it 'compile or assemble'? Did it assemble the same thing on a 6502 as it did on a PDP-11? If so, how did it run? And if not, how could the user 'know and make use of' that knowledge in a transportable fashion?"

To describe what I call traditional Forth, perhaps it would be wise to repeat the major texts I have used in teaching Forth (I am not going back to Kitt Peak Primer and the various manuals, *Forth Dimensions* articles, etc. that I used to actually learn the language—just the printed works that I feel describe what I am calling traditional Forth):

1. *Starting Forth* by Leo Brodie (published by Forth, Inc!!!)
2. *Thinking Forth* by Leo Brodie
3. *Forth: A Text and Reference* by Kelly & Spies

I should also list the systems I have used over the years, all of which have been reasonably consistent with the description of Forth given

(Figure Nine, continued.)

```
: @@            \ ( 'body -- obj )
        \ check not empty
        \ set element context
        \ fetch obj
        dup empty? not                  \ ( -- 'body ~f )
        if
                dup 'obj
                swap element-context
                <@@                     \ ( -- obj )
        else
                abort" Stack Empty!"
        then ;

: init          \ ( 'body -- )
        head off ;
END
```

in these books:

MMS-Forth for TRS-80
 Model I (two versions)
MMS-Forth for IBM-AT
MVP-Forth for DEC Rainbow
MVP-Forth for IBM-AT
MVP-Forth for Apple II
Kitt Peak VAX-Forth
F83 for IBM-AT
Guy Kelly Forth for IBM-AT

I also use F-PC, which is moderately consistent.

I should mention that I have found it not too difficult to interchange code between these systems—so my own experience has been with Forth as a fairly portable language.

As for what these systems assembled, and how use is made of it:

In each of these systems (see also the texts), the body of a dictionary entry consists generally of a sequence of addresses of component words. Embedded data is preceded by a "handler" word. Control flow is achieved by the inclusion of branching words (only a conditional "branch on zero," traditionally called ?BRANCH or 0BRANCH, and an unconditional BRANCH are needed) and special words to handle the DO ... LOOP construct.

This information constitutes the machine language for the abstract processor on which all these versions of Forth are built. As it turns out, knowledge of the exact addresses is not needed to exercise control. Only the fact that the components are of the form described above (together with a few extra details about how the processor acts when executing the code).

Let's examine how this knowledge is used to solve a simple (but somewhat amazing) problem: the introduction of a new data type into the Forth system. Forth is remarkable in that new data types can be introduced seamlessly. One aspect of this is the production of appropriate handlers for a new data type.

Traditional Forth comes with only one data type: the integer (possibly also double-precision integers). The handler embedded in code for the integer data type is traditionally called `LIT`. When `LIT` executes, it puts on the stack the integer immediately following it in the dictionary body, and then it moves the instruction pointer past that integer. Here is the definition that works on all the systems mentioned above:

```
: LIT
  R>  DUP CELL
  + >R  @ ;
```

We are using here the fact that all of these systems increment the instruction pointer and store it on the return stack when a new word executes. We can easily imitate this guide to skip over embedded data of any size, and put any information about it on the stack—perhaps just the starting address.

(I should mention that an important aspect of Forth in my work is the ability to seamlessly integrate into a Forth system new and unusual data types—some systems have as many as seven new types, each with appropriate mechanisms for storage management, appropriate handlers, operators, etc.)

The basic control structures are defined in the same way in all of these systems. For example:

```
: IF
  COMPILE ?BRANCH
  HERE 0 ,
  ; IMMEDIATE

: THEN
  HERE SWAP !
  ; IMMEDIATE
```

(Compiler security has been ignored. I believe all the above systems use the absolute address rather than a displacement—but the change is not a major one.)

With this information, one can produce any conceivable control structure on any of these systems by laying down and resolving the appropriate branch instructions. (To be sure, some such structures, like the Eaker case statement, can be synthesized using standard control constructs—although with reduced efficiency.)

In brief, the user has both knowledge of and control over what is assembled. The standard language provides words (like the control flow words) that introduce variants into the normal succession of addresses constituting the machine language of the abstract machine—but access is there for the user to do something different. In effect, the user has as much control over the process of translating a high-level language into "object code" as does the writer of a compiler for a conventional language. The user has the tools to make a high-level language look like anything he wishes—because he has complete control over the process of compilation. And he can do it portably if he uses "traditional Forth."

This is a remarkable and somewhat subversive idea: that a user should have power normally reserved to specialists. I wouldn't dismiss it as hogwash if I were you!

From: John Wavrik
Re: Disenfranchised

Mitch Bradley writes,

> "Where Dr. Wavrik has been specific rather than philosophical (e.g., user-defined control structures), the committee has attempted to deal with the issues. It would have saved me a lot of time if the specific issues had been presented in the form of proposals; then I wouldn't have had to do the work of writing the proposals."

In the interest of historical accuracy, Mitch Bradley had a proposal he wanted to submit in this area. He consulted me and a few other people. I gave him my impression of his proposal, but he submitted it anyway. I do

(GEnie continued on page 38.)

Simple Object-Oriented Forth

C.A. Maynard
Wilson, West Australia

Forth is a minimalist language, by which we mean that the core of the language provides facilities from which the user/programmer can build his own working environment. It has also been described as a syntax-directed language because, if you can define a syntax which will best express your needs, Forth will allow you to create a functional equivalent suitable for programmer use.

The latest thing in programming tools is the use of the object-oriented approach, where data and operations upon that data are all part of the same "object." This is often refered to as *encapsulation* and results in improved data security, as only those operations which have been designed to work with the object's data structure will be executed, and there is no direct access to the data itself. The user of an object need have no knowledge of the details of data storage or even details of the applicable methods. He/she just needs to know the valid operations and any parameter-passing requirements which may be necessary to operate upon the object.

The user also needs to know the terminology which fits the use of these techniques. There are several variants on the object-oriented theme, but the following is adequate for our needs.

To hide complex methods, we can set up a new vocabulary.

A *class* definition specifies the data structures needed, any initialization, and the operations which may be performed upon the class.

An *instance* is a particular object of a specific class, and has built the data structures defined in the class definition.

A *method* is an operation which can be performed upon an object.

Inheritance allows the user to minimize the additional programming necessary to handle a new variation on an object. If the methods defined for the ancestor object are valid, the object will simply obtain them via the inheritance chain. If there is the need for a different definition of a method for a new class, that may be included in the new class and will only work for the new class of object and its decendents. This ability is called *polymorphism.*

Overloading is the ability to have more than one definition of the same method, and ensuring that the correct one is applied to a particular object.

Forth has the facilities to create a very simple but effective object-oriented programming environment. The following discussion and development will not produce the fastest object-oriented implementation in Forth, but will introduce the direct application of defining and compiling words to establish an appropriate syntax.

The first requirement, then, is to propose an appropriate syntax to represent the object class. (See Figure One.)

By analogy with cooking, one should consider that this syntax provides the recipe to create new objects but that one must use the recipe to make an object (i.e., create an instance) before one can use it.

This syntax must provide the compiler with all the information needed to construct the object. The method names must be available for use by any class definition, and so will have to be defined before use.

It will also be necessary to provide syntactic delimiters between the method name and the method, and also to separate this from the following method name. These methods may result from short in-line definitions or may need to access predefined and hidden method definitions. Appropriate delimiter pairs which have been selected are: `::` and `;;` for in-line definitions, `M:` and `M;` for predefined methods.

Note for understanding the following code

1. The operations are made independent of 16-bit or 32-bit Forth implementations by using the constant `WSIZE` which returns the number of bytes assigned to storage of an integer variable. This technique reduces the efficiency of definitions but ensures portability, which is a reasonable compromise.
2. Hidden methods must be passed the address of the beginning of data within the object itself, and this is shown in the stack comments as "addr" on top of the stack.
3. The stack comments for the methods indicate the parameters needed when applying the method and/or the results produced by the method.
4. The class must inherit from another class or `NULL`.

Putting this together for an example class which consists of a point defined by its x,y coordinates and a variety of useful (and not-so-useful) methods, [refer to Figure Two].

To create an instance of this class called `PT` we simply execute:

```
10 15 POINT PT
```

Clive Maynard is a senior lecturer in the Department of Computer Engineering at Curtin University of Technology, Western Australia, where this article's contents are used as part of students' Forth instruction. He also runs Wave-onic Associates, a systems-design consultancy. He teaches real-time systems using Forth on PCs and on the Motorola 68HC11, and is a co-author of *The Art of Lisp Programming* (Springer-Verlag, 1990). Clive has developed a number of embedded systems for industrial application. Current interests include the development of practical, analytical tools for predicting real-time scheduler performance in embedded systems; as well as real-time systems and AI.

This particular example has been designed to initialize the x and y coordinates to 10 and 15, respectively, so the most likely operations to follow are similar to those given in Figure Three.

The syntax is, of course, typical RPN where the parameters are put on the stack first, then the message or method selector. Finally the operation is performed by the object itself, which is just what object-oriented programming is all about.

Examples of inheritance, polymorphism, and overloading can easily be developed by extending from a Point class to a Rectangle class, and further to a Square class. To simplify the discussion, we will use a rectangle oriented parallel to the x and y axes, which can be defined by its two opposite corners: upper left (ul) and lower right (lr). (See Figure Four.)

Creating the object becomes simply:

```
40 18 6 10 RECTANGLE
RECT
```

The rectangle object contains two points, and it is appropriate that the designer of the new class can operate on these hidden or anonymous objects as if they were separate. To obtain the upper left and lower right corner values for the rectangle, we have used the structure `GETXY IN PT`, giving us access to the point methods needed. We could have used the `GETXYM` word from the `(METHODS)` vocabulary, because it is designed to work with a `POINT` object. This is only possible because we have not made our Point methods totally hidden to other classes. If we took advantage of the `EXCISE` facility included with UR/FORTH, we could not cheat by accessing the hidden methods.

The programmer may want to use a particular class method which is appropriate for his anonymous (i.e., unnamed) objects within a new class, and there has to be a valid syntax to access the method. In this case, the data structure address is available and only the methods need to be obtained. The use of `IN` is to bypass the local data structure address inside `PT` in favor of that provided on the stack. Syntactically, this has exactly the same effect as `INHERIT` but may be used to access any object matching the hidden data structure. *This*

Figure One. Syntax to represent object class.

```
CLASS <name>
DATA
  ... \ Data structure and
      \ initialization
METHODS
  <methodname1> .. <method1>
  <methodname2> .. <method2>
INHERIT
  <ancestor> or NULL
    \ Where NULL means
    \ no ancestor class
ENDCLASS
```

Figure Two. Defining the point class.

```
METHODNAME GETX         METHODNAME GETY
METHODNAME PUTX         METHODNAME PUTY
METHODNAME GETXY        METHODNAME PUTXY
METHODNAME SWAPXY       METHODNAME GRIPE \ Used as failing method

\ The following are useful general method names providing
\ for instance initialization and class recognition for the user.

METHODNAME BUILD        METHODNAME ASTEXT

\  The following are hidden methods.
\  These words will not be visible within the Forth dictionary
METHOD: GETXYM ( addr -- x  y )
      DUP  WSIZE  +  @   SWAP  @          METHOD;

METHOD:  PUTXYM ( x y addr -- )
      DUP  >R   !  R>  WSIZE  +  !        METHOD;

METHOD: SWAPXYM   ( addr -- )
      DUP  @  OVER  WSIZE  +  @
      2  PICK  !  SWAP  WSIZE  +  !       METHOD;


CLASS POINT ( x y -- )
DATA
      ,   , \ Initialize x y to the values on the stack
METHODS
\ Inline functions
PUTX ( x -- )          ::  WSIZE  +  !  ;;
GETX ( -- x )          ::  WSIZE  +  @  ;;
PUTY ( y -- )          ::  !  ;;
GETY ( -- y )          ::  @  ;;
ASTEXT  ( -- string^ )  :: " Point" ;;
\ Hidden functions defined earlier
GETXY ( -- x y )  M: GETXYM  M;
PUTXY ( x y -- )  M: PUTXYM  M;
SWAPXY ( -- )     M: SWAPXYM  M;
BUILD ( x y -- )  M: PUTXYM  M;  \ An alias for PUTXY here.
INHERIT
      NULL
ENDCLASS
```

Figure Three. Using the point class.

```
GETXY  PT  .  .  15  10  ok  \ Note the order printed!
SWAPXY  PT  ok
GETXY  PT  .  .  10  15  ok
ASTEXT PT COUNT TYPE   Point  ok
22 7 BUILD PT ok
GETXY PT  .  .  7  22  ok
```

Figure Four. Defining a rectangle class.

```
METHODNAME PUTHEIGHT         METHODNAME GETHEIGHT
METHODNAME PUTWIDTH          METHODNAME GETWIDTH
METHODNAME UPPERLEFT         METHODNAME LOWERRIGHT


METHOD: PUTHEIGHTM ( h addr -- )
    DUP @ ROT + \ The new lower right -> ylr
    SWAP WSIZE 2* + !
METHOD;

METHOD: PUTWIDTHM ( w addr -- )
    DUP WSIZE + @ ROT + \ The new xlr
    SWAP WSIZE 3 * + !
METHOD;

METHOD: GETHEIGHTM ( addr --  h )
    DUP  WSIZE  2*  +  @  SWAP  @  -
METHOD;

METHOD: GETWIDTHM ( addr --  w )
    DUP  WSIZE  3  *  +  @  SWAP  WSIZE  +  @  -
METHOD;

CLASS RECTANGLE ( xlr ylr xul yul -- )
\  Rectangle aligned to the x y axes

DATA
    ,   ,  \ Upper left     POINT yul xul
    ,   ,  \ Lower right    POINT ylr xlr

METHODS
    PUTWIDTH ( w -- )             M: PUTWIDTHM M;
    GETWIDTH ( -- w )             M: GETWIDTHM M;
    PUTHEIGHT ( h -- )            M:  PUTHEIGHTM M;
    GETHEIGHT ( -- h )            M: GETHEIGHTM M;
    UPPERLEFT ( -- x y )          :: GETXY IN PT ;;
    LOWERRIGHT ( -- x y )         :: WSIZE 2* + GETXY IN PT ;;
    ASTEXT ( -- string^ )         :: " Rectangle" ;;
    BUILD ( xlr ylr xul yul -- ) ::  DUP >R PUTXY IN PT
                                     R> WSIZE  2*  +
                                     PUTXY  IN PT  ;;
INHERIT
    PT \ NOTE: Instance of POINT needed, not the class.
ENDCLASS
```

may only be used within a class definition, as it is only within the class definition that there is knowledge of the internal data structures to allow the programmer access.

Now for the SQUARE class, defined in Figure Five. Producing a new square:

```
5 12 13 SQUARE FRED
```

We now have a single inheritance chain of classes to experiment with! [See Figure Six.]

Creating the Object Syntax

The use of defining and compiling words in Forth provides the programmer with the ability to produce new language constructs, and is the core of the syntax-generation process.

To generate the syntax, some preliminary functions will prove useful later. During creation of a specific syntax, these will be specified as the need arises; but here we will separate them from the detailed discussion of the syntax itself.

To generate distinct method names, it is only a matter of making a functional equivalent to a variable without using the storage. In many systems where the dictionary and vocabulary coexist, simple name creation would be enough; but to be completely general, we will define them as follows.

```
: METHODNAME
  CREATE  0  ,  ;
```

To hide the more complex methods from the normal programming environment, we can set up a new vocabulary into which all of these definitions can be placed.

```
VOCABULARY (METHODS)
```

If we wished to interactively open this vocabulary for the storage of a new definition and then return to the normal FORTH vocabulary, we would enter the sequence:

```
(METHODS) DEFINITIONS
: ...content of new word...;
FORTH DEFINITIONS
```

This simple sequence may be directly converted to compiler words as follows.

```
: METHOD:
  [COMPILE] (METHODS)
  [COMPILE] DEFINITIONS
  [COMPILE] :
; IMMEDIATE

: METHOD;
  [COMPILE] ;
  [COMPILE] FORTH
  [COMPILE] DEFINITIONS
; IMMEDIATE
```

where each immediate word used for the interactive sequence is effectively deferred by the use of `[COMPILE]` until the execution of `METHOD:` and `METHOD;`. Normal colon definitions may be deferred by using `COMPILE`.

`NULL` is a word to indicate that a search through the inheritance chain has been unsuccessful, and should simply return a message to this effect and stop execution.

```
: NULL ( flag -- )
  ABORT"
  No method available "
;
```

The structure of defining words in Forth provides the basis for combining data and execution functions within one object. The basic form for a defining word is:

```
: <object>
  CREATE
  ... storage set up ...
  DOES>
  ... run-time operations
;
```

To match our object syntax to the appropriate Forth structure, the programmer builds a definition—based on the core words of the language—which will do what is needed, as in Figure Seven.

This may be demonstrated by expanding on the `POINT` example. To provide the facility for anonymous access, we need the additional concept: a method having a `TRUE` value indicates that its execution has been entered through inheritance or deferral, and that only its methods are required, not the data structure. [See Figure Eight.]

The process of configuring our object-oriented syntax is simply matching the two forms and defining the necessary compiling words to handle the operations. [See Figure Nine.]

The above constructs take less than one page of code, yet provide all the functionality discussed at the beginning of this document.

Figure Five. Defining a square class.

```
METHOD:  SIDEM ( s addr -- ) \ Store in both height and width
    2DUP  PUTHEIGHT IN RECT PUTWIDTH IN RECT
METHOD;


CLASS SQUARE ( side x y -- )
\ A square is a rectangle with height = width
DATA
    2DUP  ,  ,         \ Upper left
    2  PICK  +  ,  +  ,     \ Lower right
                       \ Using same data structure as before
METHODS
    PUTWIDTH ( w -- )          M:  SIDEM  M;
    PUTHEIGHT ( h -- )         M:  SIDEM  M;
    ASTEXT ( -- string^ )      :: " Square" ;;
    BUILD ( side x y -- )      ::  DUP  >R  PUTXY IN PT
                                   R>  SIDEM  ;;
INHERIT
    RECT
ENDCLASS
```

Figure Six. Playing with classes.

```
ASTEXT FRED COUNT TYPE Square ok
GRIPE FRED  "FRED" No method available ok
GETX FRED  .  12 OK
LOWERRIGHT FRED  .  .  18 17 ok
6 9 10 BUILD FRED ok
UPPERLEFT FRED  .  .  10  9  ok
GETWIDTH FRED  .  6
```

Figure Seven. An object as a Forth defining word.

```
: <object>
  CREATE
  ( build the instance data structure )
  DOES>
    CASE
    method1 OF do.method1 ENDOF
    method2 OF do.method2 ENDOF
    ...
    inheritance
    ENDCASE
;
```

Extending the Example: Text Window on a PC

We have developed our syntax and object creation methods, through an example sequence, from a Point to a Rectangle to a Square. A text window is an example of a rectangle with additional attributes:

Figure Eight. Dropping data address during method inheritance.

```
: POINT ( x y -- )
CREATE
 ,    ,
DOES>( method dataaddr | method dataaddr true dataaddr2 -- )
 \ Check if execution is entered through inheritance process
 \ and drop the address provided by DOES> if it is.
 OVER  TRUE  =  IF  2DROP  THEN  SWAP
 CASE
    GETY  OF  @  ENDOF      \  etc.
    ...
    SWAPXY  OF  (METHODS)  SWAPXYM  FORTH  ENDOF
    \ Switch vocabularies to find the right word
    SWAP TRUE NULL TRUE
    \ Deal with inheritance and stack requirements of ENDCASE
 ENDCASE
;
```

Figure Nine. Defining the required compiling words.

```
: CLASS   [COMPILE]  :       ; IMMEDIATE
: DATA    COMPILE  CREATE    ; IMMEDIATE

: METHODS
  COMPILE DOES>        COMPILE OVER          COMPILE TRUE
  COMPILE =            [COMPILE] IF          COMPILE 2DROP
  [COMPILE] THEN       COMPILE SWAP          [COMPILE] CASE
; IMMEDIATE

\ The following two are really deferred aliases
:  ::           [COMPILE] OF      ; IMMEDIATE
:  ;;           [COMPILE] ENDOF   ; IMMEDIATE

:  M:           [COMPILE] OF        [COMPILE] (METHODS)
  ; IMMEDIATE

:  M;           [COMPILE] FORTH     [COMPILE] ENDOF
  ; IMMEDIATE

\ IN is required to access an anonymous object within a new class
\ which, in practice, operates exactly the same as inheritance.
: IN  COMPILE SWAP          COMPILE TRUE       ;  IMMEDIATE
: INHERIT  COMPILE SWAP     COMPILE TRUE       ; IMMEDIATE
: ENDCLASS
    COMPILE TRUE       [COMPILE] ENDCASE [COMPILE] ;
; IMMEDIATE
```

1. The rectangle may be displayed.
2. The contents may be cleared.
3. Text may be placed anywhere within the window.
4. Current text should be scrollable in the window.

To create such a window, we would expect to have to use the operating system commands of the IBM-PC, but these commands should not be visible to the user of the window. All such detailed operations should be confined to the hidden vocabulary. The user should expect to see a text window object characterized by the code in Figure Ten.

The user will still be able to apply any methods associated with a rectangle object to the text window, as well as the new methods specific to the text window itself. The following code is derived from the UR-FORTH access to IBM-PC internals, and demonstrates what is needed to create the new facilities for our object, and also the ability to keep such details from the normal programmer.

Most of the code in Figure Eleven is derived from a demonstration example by Ray Duncan of LMI, but takes advantage of our predefined objects by building on the POINT facilities.

We may now complete our definition of a text window object (see Figure Twelve, page 40).

Efficiency & Generality

The approach we have used above to create an object-oriented syntax leads to a direct implementation of the requirements, but does not lead to fast execution. By eliminating the use of in-line definitions and by completing all definitions within the hidden vocabulary, it is possible to use vectored execution techniques for method access, which results in very fast chaining through the inheritance list.

The remaining limitation of this implementation is that it only supports single inheritance, by which we mean that there is a path of inheritance from any particular class to a class which inherits NULL, and failure to find the method within this search halts the process. A more general solution would be to have multiple inheritance for a class and allow the search to try to find the requested method by searching through a specified set of class chains until it finds the appropriate method. From the user's syntax requirements, this can be accomplished by simply introducing a list of inheritances to replace the single instance discussed above. From an implementation viewpoint, this is not such a simple task—but a very good analysis-and-programming exercise.

(GEnie from page 32.)
not endorse, and never have endorsed, the approach that has been taken in this area by the ANSI team. I felt that, in this case, an attempt was made, *pro forma*, to consult me. I thank Mitch Bradley for at least making an effort to hear different opinions before taking ... action.

From: Greg Bailey

In reply to John Wavrik's recent postings regarding the discussion that has followed his "disenfranchised" posting:

First, I should like to apologize to Dr. Wavrik for having misunderstood his intentions in re-posting his architecture article. It was dated 19 Aug., appeared on GEnie 20 Aug., and, given its wording ("this may be the best general response"), it seemed to me that this was the totality of his response. Since a more specific response appeared on GEnie five days later, I clearly misunderstood his intent.

Second, I should like to apologize to Dr. Wavrik if I have put any words into his mouth. On the other hand, it is difficult to discuss the positions taken by another without restating them somewhere along the line; and since obviously such restatements are not in the other party's words, it would seem that the same could be said of any rebuttal delivered by anyone. However, if my restatement of what John appears to be saying is grossly at conflict with his meaning, I am glad to be shown what the meaning really is. In fairness, however, one major reason for replying to John's postings is that he is articulate and seems to me to have put *many* words into the mouths of the TC.

For example, John has drawn the following erroneous interpretations of just several recently made points:

> "GB's... comments illustrate the fact that there are also people in the Forth

Figure Ten. User-level view of window code.

```
METHODNAME CLEAR           METHODNAME DISPLAY
METHODNAME SCROLLUP        METHODNAME SCROLLDOWN
METHODNAME >XY

CLASS WINDOW ( xlr ylr xul yul -- )
DATA
  , , , ,       \ Rectangle
\ Plus additional attributes internal to window operations
METHODS
  CLEAR         M: ... M;
  DRAW          M: ... M;
  SCROLLUP      M: ... M;
  SCROLLDOWN    M: ... M;
  >XY ( x y -- ) \ Move cursor to x y
                 \ within the window
                M: ... M;
  ASTEXT        M: ... M;
  BUILD         M: ... M;
INHERIT
  RECT
ENDCLASS
```

Figure Eleven. Details derived from LMI demo.

```
HEX
METHOD: WPAR@ ( addr -- dx cx bx )
\ Fetch parameters for an IBM-PC video I/O call
    DUP  >R LOWERRIGHT IN RECT  100  *  +
    \   dx from ul
    R@ UPPERLEFT IN RECT  100  *  +
    \   cx from lr
    R>  WSIZE  4  *  +  @
    \   bx from the attribute variable
METHOD;

METHOD: W-ATTRIB ( attrib addr -- )
\  Change the initializing attribute
    SWAP  100  *  SWAP WSIZE  4  *  +  !
METHOD;

METHOD: W-EXEC ( dx cx bx ax -- )
\ Execute the window function
    regAX  !  regBX  !  regCX  !  regDX  !  10  INT86
METHOD;


METHOD: W-CLEAR  \  Initialize the window
    WPAR@   0600   W-EXEC
METHOD;

METHOD: W-UP \ Scroll the window up
    WPAR@    0601  W-EXEC
METHOD;

METHOD:  W-DOWN  \  Scroll the window down
    WPAR@  0701  W-EXEC
METHOD;
```

(Figure continues.)

(Figure Eleven, continued.)

```
METHOD:  W-GOTOXY  (  x  y  addr -- )
\ Cursor addressing within the window
    UPPERLEFT IN RECT  D+  GOTOXY
METHOD;

METHOD:  W-HOME  (  addr -- )
\ Move cursor to the window home position, upper left
    UPPERLEFT IN RECT  GOTOXY
METHOD;

METHOD:  W-LLC  (  addr -- )
\ Move the cursor to the lower left corner of window
    WSIZE  +  DUP  @  SWAP  WSIZE  +  @  GOTOXY
METHOD;

METHOD:  W-BORDER  (  addr -- )
\ Draw a border around the window using IBM character set
    DUP  >R UPPERLEFT IN RECT  R> LOWERRIGHT IN RECT
    \  The window parameters are now on the stack
    OVER  1+  4  PICK
    DO \ Do two sides
          I  3  PICK  1-  GOTOXY  0C4  EMIT
          I  OVER     1+  GOTOXY  0C4  EMIT
    LOOP
    DUP  1+  3  PICK
    DO  \ Do the other sides
          OVER  1+  I  GOTOXY  0B3  EMIT
          3  PICK     1-  I  GOTOXY  0B3  EMIT
    LOOP
    OVER  1+  3 PICK  1-  GOTOXY  0BF  EMIT  (  urc)
    3  PICK  1-  OVER 1+  GOTOXY  0C0  EMIT  (  llc )
    1+  SWAP          1+  SWAP  GOTOXY  0D9  EMIT  (  lrc )
    1-  SWAP          1-  SWAP  GOTOXY  0DA  EMIT  (  ulc )
METHOD;
DECIMAL
```

community for whom reusability of code is not important...." "Forth has acquired an unfortunate reputation as being highly non-portable, and GB's comments serve to reinforce this impression." "...throw away time and effort needed just for a marginal gain in execution speed...." "His [GB's] work does not require portability..." "No standard is needed for people who plan to ignore it anyhow..." "...ER and GB's responses add unfortunate confirmation to the suspicion that the ANSI team is writing a new language which they plan to pass off as Forth." "The ANSI team is dominated by people who do not place much value on portability—and Greg Bailey says as much."

These and many similar passages from recent postings of John's serve to create, by repetition, the erroneous impression that members of the TC, including myself, have little or no interest in portability or reusability of code and are doing grievous harm to what John sees as Forth. In fact, this is an erroneous interpretation of at least my position, and I presume that the root of the problem is that at least until the semantic issue I mentioned on 16 August is clarified, John will continue to misunderstand the motives and actions of the TC.

Simply stated, *again*, my understanding of John Wavrik's position is that to him Forth means (and I presume he believes it was *intended* to mean) a static, open implementation model. For example, he considers that Forth includes a word spelled `DOCOL` that, when executed, returns a value that can be passed to `,` (comma) with specific and well-defined meaning having to do with the creation of a body of executable code. He also believes that Forth includes words spelled `?BRANCH` and `0BRANCH` that are, and I gather must be, used in implementing control-flow words. He feels likewise about the existence, and likewise about a method of implementation that should be guaranteed to work, for `LIT`. John, am I misstating your position here at all? I don't think I misunderstand you. What I have heard you say before is that you don't really care what it is, but whatever it turns out to be you want it all (i.e., you really strongly desire a standard that prescribes an implementation—whether you draw the "architectural" boundary there or not—at least completely enough that you *know* and *can manipulate* the executable text of a colon definition; that you *know* and *can manipulate* the structure of the dictionary; and so on). My understanding of your position is that a laudable standard could be formed by taking virtually *any* good implementation of Forth, documenting the whole thing, and saying that standard Forth must be implemented in this way on all computers.

Before I reply in detail to your postings, I think it would be useful to refine with you the above paragraph as needed, so that what we have is a concise but accurate statement of where you draw the line.

At the same time, so that we can all calibrate your sensitivity to the performance one may expect of an *application* written in Forth, I would like to know what you mean by "marginal gains in speed." For example, is a 10x performance improvement on a given CPU marginal to you? Readers of these postings might erroneously conceive, for example, that the architecture-independent definition of Forth we have tried to write in the dpANS was undertaken for no other reason than to permit implementations that shave a few percent off execution speed. Nothing could be farther from the truth.

I still feel that we are debating semantics and would like, if possible, to partition the argument into two issues: (1) the merits of architecture independence vs. prescribed implementation methods, and (2) specific things you would like to do in a portable way but feel it is impossible to do in terms of the dpANS. If possible, it would also help if items in this latter category were identified as to their portable feasibility in terms of Forth-79 or Forth-83.

As a final point for this posting, my several anecdotes about Chuck Moore were not intended to devalue portability or reusability of code. I was instead tossing them out because it seemed to me that John considered it self-evident that Forth was conceived to be what he wants it to be. This struck me as curious since, for as long as I have been participating (since the end of 1975), the inventor of Forth and those who have worked with him have *continually* been developing its architecture to increase the breadth of its applicability. Obviously, this development could have been arrested at any point to produce a frozen model that I believe would have the properties John seeks. This does not mean that our applications lack *practical* portability or reusability. It does, however, mean that, to the extent that those applications exploited the processor or the characteristics of the implementation, they would need attention when dusted off.

Figure Twelve. Completed definition of text window object.

```
HEX
CLASS WINDOW ( xlr ylr xul yul -- )
DATA
    ,   ,    \  ul  POINT
    ,   ,    \  lr  POINT  or RECTANGLE
    700 ,  \ Additional attribute internal to window operations
METHODS
    CLEAR        M:  DUP  W-CLEAR   W-HOME   M;
    DRAW         M:  DUP  W-BORDER  DUP  W-CLEAR  W-HOME  M;
    SCROLLUP     M:  DUP   W-UP   W-LLC   M;
    SCROLLDOWN   M:  DUP   W-DOWN   W-HOME  M;
    >XY  ( x y -- ) \ Move cursor to position x y in window
          M:  W-GOTOXY   M;
    ATTRIBUTE  ( attr  --  )  \  Change window attribute value
          M:  W-ATTRIB  M;
    ASTEXT        ::  " Text Window" ;;
    BUILD  ( xlr ylr xul yul  -- )
          ::  DUP  >R  BUILD IN RECT
          \  Reuse the previous definition
          700  R>  WSIZE  4  *  +  !  ;;  \ Add default attribute
INHERIT
    RECT
ENDCLASS
DECIMAL
```

The following demonstrates use of the text window objects:

```
30   10   5   5          WINDOW  W1
70   15   40   7   WINDOW  W2
70   23   10   21  WINDOW  W3

:  WDEMO
    CLS  DRAW W1  DRAW  W2  DRAW  W3
    \  Window W3 now active for text entry.
    ."  We will scroll the left window up"
    0  1  >XY  W3  ." and the right window down."
    0 BEGIN 1+
          SCROLLUP  W1  ." Line # "  DUP  .
          SCROLLDOWN W2  ."  Line  #  "  DUP  .
    ?TERMINAL UNTIL DROP
    CLEAR  W3  ." The demonstration is finished."
    0 0  GOTOXY
;
```

From: Greg Bailey

John Wavrik writes on 25 Aug. 91 that Mitch's account of events with user-defined control structures failed to mention that John does not endorse, nor has he ever endorsed, the approach that has been taken in this area by the ANSI team.

It would be enlightening for John to amplify on this negative opinion by stating his reasons. It would also be useful if John were to illustrate these reasons with some examples of things that can't be done portably in terms of the operators included in the dpANS.

Useful things that can't be done are valid demonstrations of weakness in the standard, and will always be interesting to the TC. However, the general methods documented in the dpANS (specifically of postponing members of the basic control-flow wordset) were chosen because they *do* work on the majority of systems; indeed, the major differences between these systems had to do with manipulation of items on the compile-time control-flow stack, and these differences have been addressed with operators to manipulate them. Conversely, "just using ?BRANCH and 0BRANCH" will *not* work on many systems, because many systems lack these words. Indeed, some, such as the Novix and Harris chips, and microcoded or native code implementations, have no place for those words. On the other hand, [COMPILE] IF or POSTPONE IF does in fact cover the bases in such cases.

The TC believes that the ability of a Forth programmer to compose control

(Continues on page 42.)

(GEnie, from page 40.)

structures in terms of the dpANS is vastly superior to that provided by either Forth-79 or Forth-83. Please note that the xBRANCH words were not required and, indeed, were not particularly encouraged, nor were they anywhere near universally supported; and that there was no practically portable way for users to implement control structures without depending on intimate knowledge of the intermediate database used by each system. Anyone with evidence to contradict this belief is encouraged to demonstrate problems during the review period.

From: Elizabeth Rather
To: John Wavrik
Re: "Traditional Forth"

Thank you for your very clear discourse defining what you mean by that term. I would like to urge you, however, to try to find a better adjective than "traditional," because that implies a heritage, ancestry, and universality that really isn't justified. For example, the xBRANCH words you mention were introduced in Forth-83 as an experimental wordset (by Kim Harris, I believe), and systems that maintained an allegiance to Forth-79 would not have used them. So you might say that "some" or even "many" implementations work that way, but prior to Forth-83, *no* systems worked that way that I am aware of; and it was not, by any means, universally adopted afterwards. You may feel that this is unnecessary quibbling over an adjective, but it is an adjective that has value judgements associated with it, too, and inappropriate use of it introduces heat into what should be a logical discussion.

Along the same lines, use of "assembling" to describe laying down material for the Forth engine to process obfuscates more than it enlightens, because it directs the reader's thoughts to machine code. That was what I was "hogwash-ing" at.

Now, I'll leave it to Mitch to tell you how to write portable literals in ANS Forth, because he does that sort of thing so well, and concentrate on the principles.

The TC considered including the xBRANCH words, but left them out because those of us who were familiar with a lot of systems (Martin Tracy, in particular) were able to show that, in fact, they had *not* been implemented widely, for some pretty good technical reasons. Instead, we provided POSTPONE and liberalized the use of structure words, and finally introduced some lower-level words (SO, STILL, etc.) in the TOOLKIT wordset. Wil Baden was the principal architect of our approach to handling this, and although we've fine-tuned his work somewhat, we think he did a great job. The result is that you have a great deal more power and flexibility by using phrases such as POSTPONE ELSE (for an unconditional forward branch) than with the other words, because it is required and simple to implement, whereas the BRANCH tools were in violation of so many implementations that there is no general expectation that it can be there.

In fact, a number of us on the TC like to use such techniques as you describe, and believe that ANS Forth offers greatly improved power and flexibility in these areas while additionally taking steps to improve portability of these techniques onto direct-compilation systems, Forth chips, and 32-bit systems. I guarantee you that your strategies wouldn't have worked on any of these! So the net result is not only more programmer power, but greater portability.

From: L. Zettel

Pardon me while I pick a few nits. Now that we are agreeing, for the time being, that "traditional Forth" is the Forth described by Brodie and by Kelly & Spies, I thought it would be enlightening to look up LIT in the indices of these books. Very interesting. Kelly & Spies (p. 320) give the definition:

```
: LITERAL
  STATE @
  IF COMPILE LIT
  , THEN ; IMMEDIATE
```

Brodie, second edition offers

```
: LITERAL ( n --- )
  COMPILE (LITERAL)
  , ; IMMEDIATE
```

Significantly (to my mind), *neither* offers a definition of LIT or (LITERAL).

From: John Wavrik
Re: X3J14 Holding Pattern Here

Elizabeth Rather writes,

> "The disagreement between you and the committee is not 'who wants portability' but *how portability is achieved.* We believe it can most usefully be achieved by defining the behavior of Forth words, and you'd prefer to see their implementation standardized."

Actually, the disagreement hinges more on what Forth is capable of doing—or how powerful and flexible the language should be.

This is probably the main source of disagreement. It might stem from a difference in view of what the Forth language is, has been, or could become. It might stem from a willingness to trade away capabilities of Forth to achieve harmony among vendors. It might stem from a disagreement about what it should be possible to do portably.

My claim is that Forth has traditionally been a language which allows the user to build major language features. (There is a Forth literature discussing variant methods for doing local variables, exception handling, adding object orientation, etc.) Forth has been a toolkit for building application-oriented languages. The ANSI team is heading in the direction of *including* some important features (local variables, exception handling, etc.) but *removing* the ability to build such things.

There are several other points of disagreement—most notably those having to do with clarity of definitions and simplicity of action. Words whose meanings can be interpreted differently by different implementors are useless for portable programming. The best tools available should be used to make the actions clear. Empty abstraction should be avoided—the actions of words should be as simple as possible. There are important aspects of the character of traditional Forth (simplicity, access, comprehensibility, etc.) that should be preserved.

There is no disagreement at all about describing Forth words in terms of their behavior. This is how Forth words have always been described. (On most systems, the lowest-level words have always been implemented in machine language, so it has never been possible to standardize their implementation.)

In this regard, I should mention that clarity of a description of behavior is improved immensely if a glossary entry is accompanied by a sample definition. In the Golden Days of Forth, this was a way we old-timers found helpful to convey the intended behavior of a word. I realize that the young folk have extreme prejudices against doing sensible things like this, so I'll just keep my mouth shut and rock on the porch here, looking through my old copies of *BYTE* magazine and generally basking in nostalgia!

> "Can't offhand think of *any* languages that describe how their data structures are arranged in memory, let along how their *code* is arranged in memory, which is what you seem to expect of Forth. ANS Forth pays a great deal of attention to describing data types, at least as clearly as C, etc. It also explicitly describes (Section 5.4 in BASIS, 3.4 in dpANS-2) the regions of memory that are addressable by a standard program. Most high-level languages don't let you address memory at all. C sort of does, via 'pointers,' but pointers are still a lot more abstract than Forth's addresses."

Conventional languages allow data structures only to be created by a limited set of mechanisms built into the language—and then impose further limitations on the status of these structures (how they can be passed to functions, how operators may act on them, etc.). This is one of the reasons for using Forth.

Obviously, someone must decide how a data structure is arranged in memory, how it is accessed, etc. In conventional languages, it is the designer of the language. In Forth, it can be the user (who is, in a real sense, the designer of languages).

I really have never understood arguments which pick some limitations that make other languages inflexible and use that to suggest that Forth should be equally inflexible.

From: John Wavrik
Subject: Nostalgia???!!???

Elizabeth Rather writes,

> "Our discussion of deviations from the earliest days to the present is intended to point out that there has never been such a golden age, and that your nostalgia for it is, therefore, inappropriate."

Somehow, I feel like I am in the middle of the novel *1984*, in which the establishment had newspapers rewritten to show that certain events never happened. Here is what I remember:

When I became involved with Forth, most computer magazines had regular articles on the language. *BYTE* magazine devoted at least one full issue to Forth (perhaps more). Some magazines had a Forth column. My first course on Forth was taught (by request) to 30 faculty and staff members—including representatives from the computer center, who wanted to be able to support the hot new language. Forth was the official language of astronomy, and the Center for Astrophysics and Space Studies (CASS) was one of the main groups using it at UCSD. Several people at Scripps Institute of Oceanography also used the language. I regularly received requests about where to obtain an implementation of the language. Applications written for one platform seemed magically to run on others—and there was a healthy exchange of applications and ideas. Magazine ads offered a variety of utilities (good editors, decompilers, etc.). You didn't have to justify your choice of Forth.

I am really trying to be a good citizen—so I am trying to believe with all my might that this never happened (but if it didn't, then why do I have on the wall of my office a poster of the *BYTE* magazine cover featuring Forth?).

We are losing sight of the purpose of introducing this. The way Forth is described in the most popular texts was quite common—which is why the texts described it as they did. One must remember that, if one is writing a general textbook for a language (rather than a manual for a particular dialect), it is best to stick to common practice. I have chosen the name Traditional Forth for this language because it is the form in which Forth was realized in a great many systems, from the earliest times to the present.

Please note that there is nothing in the previous paragraphs that says there were no variant systems. There is nothing in the previous paragraphs that casts aspersions on the use of a non-standard system for certain applications. There is nothing in the previous paragraphs that says that everything that has been done in the past in an attempt to standardize Forth was done perfectly.

I don't regard as nostalgia an effort to call attention to some extremely strong and positive things that were going on with the Forth language at that time.

ANSI is headed toward including some important features, but removing the ability to build such things...

Contributions from the Forth Community

We are beginning to assemble a great collection of Forth code in machine-readable form.
If you need a good Forth, it is probably here.

Minimum-requirement Forths:	**PocketForth, PYGMY, eForth**
The kitchen-sink Forths:	**F-PC, BBL**
Complete starters:	**F83, Kforth, ForST**
Object-oriented Forths:	**Yerkes, MOPS**
Macintosh Forths:	**Yerkes, MOPS, PocketForth**
IBM Forths:	**PYGMY, F-PC, BBL, F83, Kforth, eForth**
Atari Forth:	**ForST**
8051 Forths:	**8051 ROMmable Forth, eForth**
Graphic and floating-point Forths:	**Yerkes, MOPS, F-PC, Kforth**
Forth tutorials:	**The Forth Course, F-PC Teach**
Applications:	**Forth List Handler, Forth Spreadsheet, Automatic Structure Charts, A Simple Inference Engine, The Math Toolbox**
Great demos from St. Petersburg:	**AstroForth and AstroOKO**

(See the Mail Order Form inside for more complete descriptions)

Yet to come:
- Collections of tools and techniques are being assembled that cover communications, hardware drivers, data analysis, and more math and numerical recipes.

Things we need or which are not currently available in machine-readable form:
- Original listings of fig-Forth for any machine on disk. We do not currently have them.
- We can use many more applications and application ideas that include source code.
- Code from the authors of FORML papers and past *Forth Dimensions* articles.

Send submissions to: **FIG, c/o Publications Committee, P.O Box 8231, San Jose, CA 95155**

Forth Interest Group
P.O.Box 8231
San Jose, CA 95155

Second Class
Postage Paid at
San Jose, CA