A Forth Modules System with Name Reuse

Krishna Myneni and David N. Williams

14 February 2012

1a ⟨*origin file* 1a⟩≡ (25)

```
modular-forth.lyx
```

1b ⟨*version* 1b⟩≡ (25)

```
0.5.0
```

1c ⟨*license* 1c⟩≡ (25)

```
LGPL
```

1d ⟨*copyright* 1d⟩≡ (25)

```
Copyright (c) 2011-2012 Krishna Myneni and David N. Williams
```

1 Introduction

We provide a library to facilitate modular programming[1] in Forth. The primary rationale for the modules library is to,

1. Provide a framework for writing reusable code
2. Allow freedom to choose names appropriate to the task
3. Clearly define, and enforce (although not rigidly), an API
4. Enable writing code with inter-module dependencies

These features of the modules library are provided by *named modules*. Modules use two separate name spaces (wordlists), one for their *public* definitions and another for their *private* definitions. The public words of a module constitute its *application programming interface*, or API, while the private words are meant for internal use within the module, and typically are not visible outside of the module. In addition to named modules, another module type, *unnamed modules,* is provided as a simple modules facility for Forth systems which cannot support a large number of separate wordlists. Unnamed modules place their public words in the current compilation wordlist, and their private words into a single private wordlist. Our library uses design features from several sources [2, 3, 4, 5].

We will examine each of the features provided by the library. First we note that features 1), 3), and 4) provide a systematic method of organizing code for drop-in use within applications, similar to the organization of modular code in traditional structured programming languages used for modular programming, such as C. Furthermore, feature 2) of our modules system provides capabilities commonly associated with object-oriented programming (OOP), including

1. full name reuse for public API words by different modules, without name clash problems
2. ability to write modules for an abstract API specification
3. multiple instantiation of modules (similar to objects in OOP)

Our library provides a practical working solution to the name clash problem, encountered in large (or small) source development projects, using either centrally managed or distributed development models.

Although, by itself, our library does not provide a convenient object-oriented framework for Forth, it provides a bridge between traditional modular programming and full object-oriented methods. We also demonstrate how our modules system may be used to provide name space isolation[6] to a simple object-oriented Forth (OOF) implementation[7], which does not permit name reuse on its own.

1.1 The Case for Name Reuse

Bjarne Stroustrup, the inventor of the C++ language, commented that while the C language *enables* the paradigm of modular programming, it does not *support* modular programming[8]. Here, the term *support* means to go beyond just the theoretical possibility of such programming, but also to make such programming convenient. Similary, standard Forth and other Forth modular programming libraries may enable the reuse of API word names, but they do not support such reuse. What are the programming problems which will be alleviated by providing support for both modular programming and for reuse of word names in the APIs of modules? To answer this question, we consider two examples of well-known Forth libraries which are collections of source modules, the *Forth Scientific Library* (FSL)[9] and the *Forth Foundation Library* (FFL)[10].

1.1.1 Forth Scientific Library

The FSL is comprised of over 60 individual modules, featuring numeric algorithms for scientific and engineering use. It has been developed over more than a decade by various authors, working independently for the most part. The FSL provides a simple modules system in its auxiliary file[4], similar to the unnamed modules provision of our library. All private words of a module are compiled into a single wordlist. A notably inconvenient omission in the FSL modules facility is the ability to declare the beginning of a module. FSL modules which use the provided modules facility terminate the module with the word, `Reset-Search-Order`, which resets the search order to the Forth wordlist. This brute-force approach leads to problems for any application attempting to load FSL modules with other vocabularies or wordlists added previously to the search order. A cardinal rule emerges from this lesson learned from the FSL:

> *Loading of a module into the Forth environment should preserve the existing search order.*

All public words of an FSL module are compiled into the current compilation wordlist. Therefore, it is possible for public word names of one module to mask the public words of another module. An author of a new module could either devote extensive effort to studying the existing library contents and then attempt to avoid name clashes, or, more likely, leave it to the user of the library to resolve any such conflicts. Either way, the manual management of names is particularly problematic when the authors are independent, and the development is distributed.

Support for name reuse allows independent development of reusable code, without concern for name clashes. Consider the case where two module developers, Alice and Bob, are working independently on their respective modules, A and B. Neither developer is constrained in their choices for API word names, by the choices made by the other, nor do the two need to be directed by any central authority to use a particular naming convention. Alice and Bob use the names most appropriate for their particular module's API. A third developer,

Deepa, may write her module, D, which depends on both A and B, and refer to the names in A, B, and D, without ambiguity even if there are overlapping names. Furthermore, Deepa can even write her module in such a manner that if either Alice or Bob revises her/his module to provide additional API names, the code in D will not break due to name clashes.

Next, we consider the benefits of name reuse for module libraries developed through a centralized development model, which typically do not have the problem of name clashes since strict naming conventions are often defined and enforced. An example of this is the Forth Foundation Library, discussed below.

1.1.2 Forth Foundation Library

The FFL is another example of a library comprised of many individual modules. In this case, the library was written by a single author. In the FFL, no modules system is used, and there are no public or private wordlists. All definitions are compiled into the current compilation wordlist. Word names are prefixed to indicate the operand type, e.g. words for working with *singly-linked lists* are prefaced by "snl-", e.g. `snl-new, snl-init,` etc., while words for accessing or manipulating generic *binary trees* are prefixed with "bnt-", e.g. `bnt-new`, `bnt-init`, and so on.

Consider the following word definition from the FFL module, `snl.fs`.

```
: snl-remove-first   ( snl -- snn | nil )
  dup snl-first@
  dup nil<> IF               \ If first <> nil Then
    2dup snn-next@
    dup nil= IF              \   If first.next = nil Then
      over snl>last nil!     \     last.nil
    THEN
    swap snl>first !         \   first = first.next
    swap snl>length 1-!      \   length--
  ELSE
    nip
  THEN ;
```

The above word removes the first node of a singly-linked list. Using our modules system, this word would be a member of the named module, `ffl-snl`, and written as,

```
: remove-first   ( snl -- snn | nil )
  dup first@
  dup nil<> IF               \ If first <> nil Then
    2dup next@
    dup nil= IF              \   If first.next = nil Then
      over >last nil!        \     last.nil
    THEN
    swap >first !            \   first = first.next
```

```
    swap >length 1-!          \   length--
  ELSE
    nip
  THEN ;
```

All name prefixes, `snl-` in the above example, may be eliminated, leading to greater readability of the code, *and* without concern for name clashes! Furthermore, the non-prefixed words such as `NEW`, `INIT`, `INSERT`, `DELETE`, `GET`, `CLEAR`, `LENGTH@`, and `EMPTY?`, are common API words for both binary trees and singly-linked lists in the FFL. Thus, a *generic interface* may be provided for a set of modules requiring similar application interfaces.

In summary,

> *Support for name reuse allows independent development of reusable code, and permits writing more comprehensible code using generic interfaces.*

1.2 Search Order Utilities

We begin the development of our modules library by noting that public and private words of a module are grouped into different wordlists, and the switching among these wordlists in the search order requires convenient utilities: pushing and popping wordlists to/from the search order, obtaining and setting the search order depth, and saving and restoring the search order state. Additionally, saving and restoring the compilation wordlist is a necessary utility, and is included as part of the utilities.

5 ⟨*search order utilities* 5⟩≡ (25)

```
[UNDEFINED] drops [IF]
  : drops ( +n - ) 0 ?DO drop LOOP ;
[THEN]

[UNDEFINED] order-drops [IF]
  : order-drops  ( +n - ) ( o: wid_n ... wid_1 - )
      0 ?DO previous LOOP ;
[THEN]

[UNDEFINED] order-depth [IF]
  : order-depth  ( - n )  get-order dup >r drops r> ;
[THEN]

[UNDEFINED] >order [IF]
  : >order  ( wid - order: wid )
      >r get-order r> swap 1+ set-order ;
[THEN]

[UNDEFINED] order> [IF]
```

```
  : order>  ( o: wid - ) ( - wid )
       get-order swap >r 1- set-order r> ;
[THEN]

[UNDEFINED] order@ [IF]  \ not used here
  : order@  ( o: wid - ) ( - wid )
       get-order over >r drops r> ;
[THEN]
```

6 ⟨*search order state utils* 6⟩≡ (25)

```
VARIABLE initial-defs
VARIABLE initial-order-depth

: save-SO-state ( - )
        get-current  initial-defs !
        order-depth  initial-order-depth !
;

: restore-SO-state ( - )
        initial-defs @ set-current
        order-depth initial-order-depth @ - order-drops
;
```

2 Modules

The modules library provides the following elements for writing and working with Forth modules:

2.1 Types and Layout

The layout of a module is specified by the following words,

```
MODULE:  BEGIN-MODULE  PUBLIC:  PRIVATE:  END-MODULE
```

The word `MODULE:` declares the name of a module, and is usually the first statement in a Forth module source file. The `MODULE:` declaration is not used when an unnamed module is desired. `BEGIN-MODULE` is where the module begins, and `END-MODULE` is where it ends. The code area between the two is the *body* of the module. Within the body, the words `PUBLIC:` and `PRIVATE:` control the visibility of the module code to the module's user, the application programmer.

2.1.1 Unnamed Modules

The simplest type of module is an unnamed module. For this type of module, no module name is declared, i.e. the word `MODULE:` is absent.

```
BEGIN-MODULE
\ ...
Public:
\ ...
END-MODULE
```

The module's public words (API) are compiled into the current compilation wordlist, while the module's private words (internals) are placed into a separate `Private-Words` wordlist. Both the current compilation and `Private-Words` wordlists are pushed onto the search order by `BEGIN-MODULE`.

7 ⟨*unnamed private words* 7⟩≡ (25)

```
wordlist  CONSTANT  Private-Words
```

Since *all* unnamed modules place their private words into `Private-Words`, it is possible for private words from one module to redefine the names of private words from a previously loaded module. Name redefinitions pose no problem for proper execution of the API words. Indeed, it is common Forth practice to reuse names of words which are required only locally, e.g. temporary variables. However, redefinitions of words hide earlier definitions from inspection or calling by the user, and this may cause difficulty in debugging hidden words from within the Forth environment.

In practice, an issue of greater concern with unnamed modules is that the API words are placed in the current compilation wordlist. Thus,

> *Name reuse for API words in different modules is not guaranteed to be possible with unnamed modules.*

The latter problem leads to poor programming practices such as name prefixing, and the consequent drawbacks for applications programming. In general, unnamed modules are only recommended for use when the target Forth systems are severely constrained in the number of wordlists they can provide to the user. Named modules should be used where no such practical restrictions exist.

2.1.2 Named Modules

Named modules simply have the module name declaration prior to `BEGIN-MODULE`, e.g.

```
MODULE: Foo
BEGIN-MODULE
\ ...
Public:
\ ...
END-MODULE
```

Since each named module has its own distinct pair of wordlists for its API words and internal words, it is possible to

1. reuse API word names
2. inspect and call module-specific private words from the Forth environment.

The first possibility allows application programming with generic interfaces, thereby decoupling the interface to the module from its specific implementation. Several OOP-like features arise from this possibility, and these features will be discussed elsewhere. The second possibility is accomplished by retrieving the private wordlist id of the module and placing it on top of the search order, e.g.

```
' Foo >private >order
```

where the definition of `>PRIVATE` is given in section 2.3.

2.2 Named Module Declaration

Named modules are declared with,

```
MODULE: <module_name>
```

`MODULE:` also saves the search order depth and current compilation wordlist, for automatic restoration at the end of the module.

9 ⟨*module declaration* 9⟩≡ (25)

```
VARIABLE named-module   0 named-module !

: make-module ( "name" - )
        CREATE  here 0 , 0 , 0 , named-module !
        DOES>  ( o: wid' - wid )  @ >r get-order nip r> swap set-order ;

: MODULE:  ( "name" - )
        save-SO-state
        MODULES-WORDLIST set-current
        make-module

        initial-defs @ set-current
;
```

The word `MODULE:` creates the module name as a word in the reserved wordlist, `MODULES-WORDLIST`, and reserves space to store the public and private wordlist ids. Space is also reserved to store the address of a list of dependent modules, for future implementation.

> *An important feature of the execution behavior of the module name is that it has the same behavior as a Forth-83 vocabulary name.*

Executing the module name will replace the top wordlist in the search order with the wordlist containing the API words of the module. Hence, to append a module's API (public wordlist) to the current search order, one may simply write,

```
ALSO <module_name>
```

This isomorphism allows us to treat the Forth-94 standard words, `FORTH` and `ASSEMBLER` , in nearly the same way as a module name, a feature which will prove beneficial for word name reuse and resolving word references within named modules.

2.2.1 The Modules Wordlist

All module names are compiled into a separate wordlist, referenced by the constant, `MODULES-WORDLIST`, and added to the search order at the time that the modules library is loaded. `MODULES-WORDLIST` *is reserved for use by the modules library, and should not be used as a compilation wordlist for any other purpose.* In general, this wordlist should be transparent to the application programmer or user, and direct use of `MODULES-WORDLIST` is discouraged. However, in an interactive Forth environment, it may be necessary at times to manually push `MODULES-WORDLIST` onto the search order, e.g. after a search order reset such as the statement, `ONLY FORTH`. A Forth-83 vocabulary style word, `MODULES`, is provided for use with `ALSO` to accomplish this.

10 ⟨*modules wordlist* 10⟩≡ (25)

```
wordlist  constant  MODULES-WORDLIST
MODULES-WORDLIST >order

: MODULES ( - )  get-order nip MODULES-WORDLIST swap set-order ;
```

The following two statements are, therefore, equivalent.

```
MODULES-WORDLIST >order
ALSO MODULES  \ equivalent to above
```

Either syntax may be used – the latter is already familiar to Forth programmers.

Organization of the named modules into a separate wordlist makes it possible to provide a simple and convenient way for the programmer to view the modules which have been loaded into the Forth environment. This is done with SHOW-MODULES.

11a ⟨*show modules* 11a⟩≡ (24)

```
: show-modules ( - )  \ Display all loaded modules
        MODULES-WORDLIST >order  words  PREVIOUS ;
```

2.3 Public and Private Wordlists

For named modules, the module name word has a body which stores both the public and private wordlist ids (wids). These two wordlists are newly created when BEGIN-MODULE is encountered. The corresponding wids are also stored within the body of the module name word.

11b ⟨*new wordlists* 11b⟩≡ (12b)

```
wordlist    \ new private wordlist
wordlist    \ new public wordlist
2dup named-module @ 2!
```

Direct access to a named module's wids may be needed to support debugging and testing. For example, we may wish to add a module's private wordlist, which is not accessible directly, to the search order. It is possible to retrieve both public and private wids simply from the execution token (xt) of the module name, e.g., for the module named Foo,

```
' Foo >public     \ obtain Foo's public wid
' Foo >private    \ obtain Foo's private wid
```

The words >PUBLIC and >PRIVATE have simple definitions:

11c ⟨*module wordlist primitives* 11c⟩≡ (24)

```
: >public  ( xtmodule - wid )  >body @ ;
: >private ( xtmodule - wid )  >body cell+ @ ;
```

All unnamed modules use a single separate wordlist for their private definitions, referenced by the constant, Private-Words. The public wordlist used by an unnamed module is simply the current compilation wordlist. Therefore, retrieval of the pair of existing public and private wordlists for an unnamed module is accomplished by,

11d ⟨*existing wordlists* 11d⟩≡ (12b)

```
Private-Words
get-current
```

2.4 Start of the Module

The start of the body of a module is declared by `BEGIN-MODULE`. Subsequently defined words are compiled into either the module's private or public wordlists. Prior to modifying the search order and compilation wordlist, `BEGIN-MODULE` also saves information which allows `END-MODULE` to restore the initial search order depth and initial compilation wordlist.

12a ⟨*set public private* 12a⟩≡ (12b)

```
( wid_private wid_public - )  public-defs !  private-defs !
```

12b ⟨*begin module* 12b⟩≡ (25)

```
VARIABLE private-defs
VARIABLE public-defs

: BEGIN-MODULE  ( - ) ( o: - public private )
    named-module @ if
      ⟨new wordlists 11b⟩
    else
      save-SO-state
      ⟨existing wordlists 11d⟩
    then
    2dup ⟨set public private 12a⟩
    2dup  >order  >order
    drop set-current    \ default section is private
;
```

The public and private wordlists are pushed onto the search order, and the private wordlist is set to be the current compilation wordlist.

2.5 Body of the Module

The body of the module is the code section between `BEGIN-MODULE` and `END-MODULE`. Defined words in the body of a module are compiled into either the module's public or private wordlists. The word `PUBLIC:` declares that subsequent definitions will be placed in the public wordlist, while the word `PRIVATE:` declares subsequent definitions will be placed into the private wordlist.

> *The public words of a module constitute its API to the user or application programmer. Private words are for internal use within the module, and are not intended to be used by external code.*

Private words may be internal data manipulated by the module code, along with helper words which are useful factors of the public words. Following `BEGIN-MODULE`, the compilation wordlist is set to the private wordlist.

The declarations, `PRIVATE:` and `PUBLIC:`, should occur only within the body of a module, and may be invoked any number of times there, and in any order.

13a ⟨*body declarations* 13a⟩≡ (25)

```
: not-in-module ( - ) ."  Not in Module Body!" cr ABORT ;
: safe-set-current ( wid - )
        ?dup IF set-current ELSE not-in-module THEN ;
: PRIVATE:  ( - )  private-defs @  safe-set-current ;
: PUBLIC:   ( - )  public-defs  @  safe-set-current ;
```

2.6 End of the Module

The task of `END-MODULE` is relatively simple: restore the initial search order depth and the initial compilation wordlist, and reset the public and private wordlist variables to a known state. It also resets the variable, `named-module`, the contents of which are used to determine which type of module is being used for the next use of `BEGIN-MODULE`.

13b ⟨*end module* 13b⟩≡ (25)

```
: END-MODULE  ( o: <extras> - )  restore-SO-state
        0 named-module !
        0 private-defs !
        0 public-defs  ! ;
```

If search order manipulations performed after the module name declaration, `MODULE:` , or between `BEGIN-MODULE` and `END-MODULE`, only have the effect of adding additional wordlists to the search order, the initial search order will be restored by `END-MODULE`. An ambiguous search order state exists if the initial search order has been modified, instead of just added to.

2.7 Restrictions

A few restrictions of the present modules library are important to note:

1. The module layout is not nestable.
2. The part of the search order present immediately before the module declaration may not be changed inside the module.

While restriction 1) is not inherent to the modules system, but rather a restriction based on our current implementation, there is presently no compelling reason to allow modules to be nested. Restriction 2) is intended to avoid any side effects on the state of the system due to the module code. Since the search order depth and compilation wordlist just before `MODULE:` (or just before `BEGIN-MODULE` for unnamed modules) are restored by `END-MODULE`, this restriction allows for the initial search order to be extended within the module body, and restored to the initial state after `END-MODULE`. The integrity of the system search order state could be made safer by actually saving and restoring the initial search order. However, our implementation does not do that.

3 Programming with Modules

An important feature of our modules system, and one which distinguishes our system from prior systems, is support for name reuse of API words. As a general rule,

> W*e advise against using the names of standard Forth words as module member word names.*

For example, it is not advisable to use the words `CREATE` and `FREE` as API names. In contrast, generic words such as those shown in Table 1 often provide the clearest API names in various contexts, and may be considered for reuse in different modules, where such names are sensible choices. When a class of modules requires the same high-level API, a generic module may provide an *abstract interface* layer, consisting of such generic words, for use by the application. An example is an application which communicates with devices across different hardware interfaces, but using a common software interface, e.g. a graphics program for generating output on various graphics devices.

When a word name appears in multiple wordlists, there exists a dependency on the current search order for references to the word. Manipulation of the search order is necessary to ensure that the desired word is compiled. Such manipulation can be accomplished with standard Forth, but it may be cumbersome to do so. A similar issue also arises for conditional definitions of words. Our modules system provides operators to directly reference a word within a named module, and to test for membership of a word within a module. Use of these reference operators effectively bypasses the problem of search order dependency and improves the readability of the code.

Get	New	Open	Commit	Setup	Show	Solve
Insert	Empty	Close	Verify	Home	Display	Calc
Delete	Equals	Read	Lock	Track	Draw	Step
Copy	Set	Write	Unlock	Start	Refresh	Integrate
Append	Index	Clear	Enable	Stop	Rotate	Reduce
Put	Position	Init	Disable	Acquire	Flush	Interpolate
Length	Print	Reset	Assign	Sample	Sync	Fit
First	Sort	Send	Detach	Accumulate	Query	Tolerance
Last	Remove	Receive	Select	Store	Clip	Iterate
Push	Modify	Connect	Listen	Retrieve	Offset	Use
Pop	Exchange	Status	Talk	Configure	Mask	Zero

Table 1: Generic API word name examples.

Modular programming must also take into account dependencies between modules. The dependencies among modules may be depicted using a *dependency graph*, as shown in Figure 1. Module dependencies affect both the *load sequence* of modules, and search order setup required for the successful loading of a module. We discuss the implication of word name reuse for effective search order setup needed by a module.

3.1 Module Dependencies and the Search Order

Modules, vocabularies, or ordinary wordlists required by a named module should be brought into the search order after the name declaration and before the start of the module, i.e. between `MODULE:` and `BEGIN-MODULE`. Our cardinal rule, that loading of a module should preserve the search order, is provided for by `END-MODULE`, which automatically removes the wordlists added to the search order after the name declaration. For the dependency graph shown in fig. 1, the recommended placement of the declaration of dependencies for module E is illustrated below.

```
Module: E
Also D  Also B  Also C   \ Dependencies for Module E
Begin-Module
( ... )
End-Module
```

Note that BEGIN-MODULE will push the module's own public and private wordlists onto the search order, so that a module's words are always found first within the body of the module.

It is not recommended that wordlists be added to the search order within the module body, i.e. after `BEGIN-MODULE`. Since `BEGIN-MODULE` pushes the current module's public and private wordlists on top of the search order, adding to the search order after `BEGIN-MODULE` can potentially mask the module's own word names within its body. Such a practice can cause problems for the module's

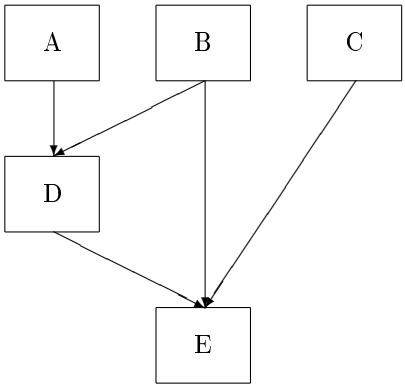

Figure 1: Dependency Graph for a Set of Modules. All modules also have a dependency on the Forth wordlist (not shown).

author, since he/she has to be aware of any potential name clashes from the dependencies. Also, future versions of the dependent modules could cause the module code to break due to new name clashes being introduced.

> *Adding to the search order within the body of the module is strongly discouraged, with the exception that local modification and restoration of the search order is permissible.*

In the presence of name reuse, it appears that one must also be careful with the ordering of the dependencies. However, as mentioned earlier, our modules library provides reference operators, the proper use of which can make irrelevant the ordering of dependencies. These operators are discussed in the next section.

3.2 Referencing Module Member Words

Support for name reuse in modules requires convenient methods for referencing the desired word from a specific module. It is not always convenient to manipulate the search order manually to place a particular module's word(s) on top of the search order. This is particularly true when defining a new word. As an illustration, consider the following case. An existing module named `Bar` provides a public word named `Get`. While coding a new module, `Foo`, we wish to write a word, `Insert`, which references `Get` from the module `Bar`. The code is sketched below.

```
Module: Foo
Also Bar
```

```
Begin-Module
Public:
: Get ( ... ) ;  \ member of Foo

: Insert ( ... )
    [ ALSO Bar ]  Get  [ PREVIOUS ]
    ( ... )
```

The definition of `Foo`'s member named `Get` masks the definition of `Bar`'s `Get`, because `Bar`'s API words are lower in the search order than those of the current module. To avoid this masking, the search order is locally modified to reference `Get` from the module `Bar` and then restored. Manipulation of the search order inside the definition of a word is messy, and results in code which is hard to read.

Two operators are introduced to simplify referencing of a word in a specific module, independent of the current search order. These are the module *member reference operator*, ∋, and the module *self-reference operator*, ∃. We use the Unicode math symbols, ∋, meaning "contains as member" (U+0x220b) and ∃, meaning "there exists" (U+0x2203); however, corresponding plain text versions of these operators are also provided: `[m]` and `[this]`, respectively. Both ∋ and ∃ are state smart.

3.2.1 Module Reference Primitives

For proper error reporting by the operators, ∋ and ∃, the word name to be searched for as a member of the specific module is stored. The word `member-find` is a parsing word which checks for the presence of the named module, and looks up the member word in the public wordlist of the module. The word `module-do-ref` performs either compilation or execution of the referenced word based on the current state *and* the precedence of the word.

17 ⟨*module reference primitives* 17⟩≡ (24)

```
create member_name 128 allot

: !member-name ( c-addr1 - c-addr2 u)
       dup c@ 127 min 1+ member_name swap move ;

: member-find ( "module-name" "word-name" - 0 | xt 1 | xt -1 )
       bl word find 0= ABORT" Unknown Module!"
       also execute order> >r ( "word_name" )
       bl word !member-name
       member_name count r> search-wordlist ;

: member-not-found ( - )
       member_name count type
       ."  : Member Not Found!"  cr ABORT ;
```

```
: module-do-ref ( ... xt 1 | xt -1 - ...)
        -1 = state @ and IF  compile, ELSE execute THEN ;
```

Analogous words to the Forth standard words, "tick" (') and "bracket-tick" ([']), may now be defined to return or compile execution tokens referenced by module name.

18a ⟨*module tick operators* 18a⟩≡ (24)

```
: m' ( "module-name" "word-name" - xt )
        member-find 0= IF member-not-found THEN ;

: [m'] ( "module-name" "word-name" - )  m' postpone literal ; immediate
```

3.2.2 The Member Reference Operator

The *member reference operator*, ∋ , allows direct referencing of a word which is a public member of a specific named module, using the syntax,

```
∋ <module_name> <word_name>
```

The reference operator ∋ may be used either inside or outside of a module, and in either compilation or interpretation state.

If the specified word is not found in the public wordlist of the specified module, compilation is aborted with an error message:

```
<word_name> : Member Not Found!
```

A plain text representation of the ∋ operator is [m].

18b ⟨*member reference operator* 18b⟩≡ (24)

```
: [m] ( ... "module-name"  "word-name" - ? )
        member-find
        ?dup IF  module-do-ref  ELSE  member-not-found  THEN
; immediate

true [IF] : ∋ postpone [m] ; immediate [THEN]
```

Our previous example in section 3.2 for the definition of `Foo`'s member `Insert` now may be written as,

```
Public:
: Get ( ... ) ;  \ member of Foo
: Insert ( ... )
    ∋ Bar Get
    ( ... )
```

Furthermore, it is possible that an API word of the current module may hide an intrinsic word in the Forth wordlist. Here, the design feature of having the execution behavior of a module name be the same as the intrinsic Forth-94 word, `FORTH`, means that we can apply the member reference operator to the word `FORTH`. For example, if the Forth wordlist provides the word `SCAN`, we may write,

```
∋ Forth scan
```

to reference the word `SCAN` from the Forth wordlist. Such a reference will never be masked by name reuse in other wordlists, which are higher in the search order than the Forth wordlist. While we advised against the reuse of *standard* Forth names, a typical system's Forth wordlist provides numerous additional words, beyond just the standard words.

3.2.3 The Self-Reference Operator

The *self-reference operator*, ∃, is used only within the body of a module, with the syntax,

```
∃ <word_name>
```

This operator denotes that the specified word name refers to a word in the *current* module's private *or* public wordlist. It may be used in either interpretation or compilation state. If the specified word is not found in *either* the private or public wordlists of the current module, compilation is aborted with the error message:

```
<word_name> : Member Not Found!
```

The plain text representation of the ∃ operator is `[this]`.

19 ⟨*self reference operator* 19⟩≡ (24)

```
: [this] ( "word-name" - )
    bl word !member-name
        private-defs @ 0= IF not-in-module THEN
        public-defs  @ 0= IF not-in-module THEN
        member_name count private-defs @ search-wordlist
        ?dup IF
```

```
                module-do-ref
            ELSE
              member_name count public-defs @ search-wordlist
              ?dup IF  module-do-ref  ELSE  member-not-found  THEN
            THEN ; immediate

true [IF]  : ∃ postpone [this] ; immediate  [THEN]
```

3.3 Conditional Definitions of API words

It is common practice in Forth source to avoid duplicate definitions of common usage words through conditional definitions, using `[UNDEFINED]` or `[DEFINED]`. A module, however, must guarantee the definition of its member API words within the module body. Therefore,

> *Conditional definitions of public words in a module must ensure that a new definition always exists.*

For example, consider a strings module which provides the commonly used, although not standard, word named `SCAN`. Since some Forth implementations may provide a built-in `SCAN`, the following type of conditional definition is often used in Forth source,

```
[undefined] scan [if]  : scan ... ;  [then]
```

Such a conditional definition of `SCAN`, made within the public section of a module, poses a problem. If `SCAN` is found in the current search order, the module will provide no definition of an API word named `SCAN`. Then, an attempt to use the module's member `SCAN`, via the member reference operator, for example, would fail!

Within the module, a definition of the module's API word, `SCAN`, may be ensured by writing,

```
[undefined] scan [if]  : scan ... ;
[else]  : scan scan ;  [then]
```

The above approach is dependent on the current search order. More than one module or F83 vocabulary in the current search order may provide `SCAN`. We provide the word `[MEMBER]` to perform a targeted membership test, thereby avoiding the search order dependency. Thus, if the Forth system provides `SCAN`, we may write,

```
[member] Forth scan [if]
  : scan ∋ Forth scan ;
[else]
  : scan ... ;  \ new source definition of SCAN
[then]
```

21 ⟨*member test* 21⟩≡ (24)

```
: [member] ( "module" "name" - flag )
    member-find if drop true else false then ; immediate
```

3.4 Loading Modules

A module may either choose to load its dependent modules, to ensure they are accessible in the Forth environment, before adding them to the search order, or may leave the task of loading dependencies to the higher level application code. The latter approach is generally more easy to correct whenloading problems arise, as may happen when using a later version of a module, for which the dependencies have changed.

4 Future Directions

The practical application of any framework is essential to closing the feedback loop necessary to craft a useful and usable library. We have put into practice the modules system described here, presently with applications each using on the order of 10 modules. Indeed the exercise of converting existing Forth files to Forth modules has led to iterative enhancements, refinements, and bug fixes with the modules library. The exercise has also demonstrated to us the benefits of name reuse which we discussed in this literate program. At this early stage, it is likely that we may have missed an essential feature (or few), needed to support the conversion of existing code to a modular format. As we, and, hopefully others in the Forth community, gain experience with our modules library, such deficiencies will be made apparent and remedied in future versions of the library.

One area of planned development is to provide additional tools to Forth authors, to aid with development of modules supporting name reuse. Already, the simple tool `SHOW-MODULES` is very useful for a programmer to viewthe available modules loaded in a Forth system. Other envisioned tools will support the development and use of generic interfaces by allowing the programmer to query for a list of name overlaps within modules. Such tools include:

- display the names of all modules which provide a particular name in their API,

  ```
  s" <word_name>" MEMBER-OF
  ```

- display all API names which overlap between two given modules,

  ```
  ' <mod_A>  ' <mod_B>  NAMES-OVERLAP
  ```

- display a given module's dependencies,

  ```
  ' <mod_name>  SHOW-DEPENDENCIES
  ```

Some of the above tools, such as `SHOW-DEPENDENCIES`, simply require additional bookkeeping by the modules library – we have already reserved a cell in the body of a module name word to be able to save a list of dependencies associated with a module. Other tools such as `MEMBER-OF` and `NAMES-OVERLAP` currently cannot be implemented in standard Forth, due to the lack of a standard way

of traversing wordlists and obtaining word names. However, we expect such features may be standardized in the near future since a number of Forth systems already provide such features. A Request for Discussion (RfD) has already been posted to the comp.lang.forth newsgroup for the standardization of a word named `TRAVERSE-WORDLIST`. We are optimistic that standardized infrastructure will be adopted in Forth systems to support the development of such tools.

Finally, as Bernd Paysan and others have noted, it is desirable to have a modules system which is a subset of a fully-implemented object-oriented programming framework in Forth (OOF). Since our modules library provides some of the features of OOF, further experience with the library may lead to the development of such a framework, usable both as a modules system or for OOF programming. Whether or not such a fusion will be possible, or desirable, we believe that our current modules framework provides a number of useful benefits for the development of large, robust, and comprehensible Forth programs under diverse development environments.

5 The Modules Library

5.1 Forth System Requirements

In addition to system requirements listed below, the modules library assumes that `HERE` never returns zero.

5.1.1 Maximum Number of Wordlists

The Forth 200x requirements that at least eight wordlists be allowed in the search order and that it must be possible to create at least eight new wordlists, when the Search-Order wordset is present, may be adequate for many uses of modules. The file `modules.fs` adds `MODULES-WORDLIST` when loaded. During the loading of a module, a minimum of two wordlists is temporarily added. A minimum of one wordlist is needed for each named module on which a module depends.

But having to worry about how many wordlists are allowed is not in the spirit of the freedom of name reusage afforded by named modules. Fortunately some common systems do not have an eight wordlist limitation. For the search order, gforth and iForth allow 16, and pfe allows 64, and kForth permits an arbitrary number of wordlists. At present we have not yet determined whether or not practical limits exist in various Forth systems for the number of new wordlists which may be created. An application using 50 named modules will require 100 new wordlists to be created. However, only a few of these wordlists will need to be present in the search order at any given time.

5.1.2 Required Words From Optional Wordsets

The modules library requires words from the following optional wordsets from the Forth 200x standard:

Core Extension: `?DO COMPILE, NIP STATE TRUE`

Programming-Tools: `WORDS [IF] [THEN]`

Search-Order: `GET-CURRENT GET-ORDER SEARCH-WORDLIST SET-CURRENT SET-ORDER WORDLIST`

Search-Order Extension: `ALSO PREVIOUS`

5.1.3 Unicode Support

Unicode support is not a system requirement. Optional names for two of the words in `modules.fs` use unicode, namely, ∋ and ∃. Many modern Forth systems support the use of UTF-8 unicode characters, even when they do not include the optional Extended Characters word set, especially when loaded from a source text file. To properly display such characters in a terminal window or enter them from the keyboard, the host terminal application must have them enabled.

The unicode names may be deselected — the ASCII names `[M]` and `[THIS]`, corresponding to ∋ and ∃, are always present.

5.2 modules.fs

The chunks of the modules library are now assembled.

24 ⟨*module utilities* 24⟩≡ (25)
⟨*show modules* 11a⟩
⟨*module wordlist primitives* 11c⟩
⟨*module reference primitives* 17⟩
⟨*module tick operators* 18a⟩
⟨*member reference operator* 18b⟩
⟨*self reference operator* 19⟩
⟨*member test* 21⟩

25 ⟨*modules.fs* 25⟩≡

```
\ Modules library, version ⟨version 1b⟩
\ License: ⟨license 1c⟩
\ ⟨copyright 1d⟩
\ This file is automatically generated using LyX and noweb.
\ Changes should be made to the original file, ⟨origin file 1a⟩
```

⟨*search order utilities* 5⟩
⟨*search order state utils* 6⟩
⟨*unnamed private words* 7⟩
⟨*modules wordlist* 10⟩
⟨*module declaration* 9⟩
⟨*begin module* 12b⟩
⟨*body declarations* 13a⟩
⟨*end module* 13b⟩
⟨*module utilities* 24⟩

Acknowledgements

Both public discussions on the usenet newsgroup, comp.lang.forth, and private discussions by e-mail with several members of the Forth community have been very influential in the design and coding of the modules library presented here. In particular, we would like to acknowledge Bruce McFarling, with whom we have had extensive public and private discussions. A significant portion of the code, including search order utilities, are due to these discussions. Numerous other members of the Forth community were involved in the comp.lang.forth discussions of modular programming in Forth during October 2011, and this discourse has also significantly influenced the design and consideration of the modules system features.

References

[1] http://en.wikipedia.org/wiki/Modular_programming

[2] D. N. Williams, `root-module.fs`, v 0.8.2 (2011).

[3] B. McFarling, suggested module facility for the Forth Scientific Library, `comp.lang.forth` (2011).

[4] E. F. Carter, et. al., Forth Scientific Library Auxiliary File, `fsl-util.x`, `http://www.taygeta.com/fsl/library/fsl-util.fs (2008).`

[5] N. Bridges, *Named and Anonymous Modules for Standard Forth*, http://qualdan.com/forth/modules.fs (2006).

[6] K. Myneni, example use of modules with mini-oof, ftp://ccreweb.org/software/gforth/experimental/modules/textbox/ (2011).

[7] B. Paysan, *Detailed Description of Mini-OOF*, `http://bernd-paysan.de/mini-oof.html` (2008).

[8] B. Stroustrup, *The C++ Programming Language, 2nd ed.*, Addison-Wesley (1993); see section 1.2.

[9] C. G. Montgomery, ed., *Forth Scientific Library*, http://www.taygeta.com/fsl/sciforth.html(2011).

[10] D. van Oudheusden, *Forth Foundation Library*, http://soton.mpeforth.com/flag/ffl/index.html(2010).