Some comments on the proposed Forth Standard

Ian van Breda

Hailsham, East Sussex, UK

Abstract

This paper discusses the current proposals for a new Forth standard following on from ANS Forth. Some changes of name are suggested. It is shown that the proposals for the Extended Characters wordset can either be subsumed into existing definitions for fixed-character sets or be added in a way that make the two schemes compatible. Proposals are also made for record structures coupled with dot notation, that fit well with existing Forth practice and would allow a natural extension to object-oriented Forth.

1 Philosophical

I feel that I am something of an interloper at this conference, as my background in astronomy is rather different from most attendees. I am fully aware that my comments may not always be welcomed by those who have spent an enormous amount of work putting together the proposals for a new Forth Standard [1]. This is even before considering the considerable effort required to reconcile differing opinions, often strongly held and to which I cannot claim to be an exception. My comments are not intended as personal criticism in any way but are based purely on technical considerations. It is to be hoped too that any misconceptions in my comments will be forgiven.

With thousands of Forth users worldwide, establishing a standard is very important. The Holy Grail for desktop computing must be for the computer to run a pre-emptive multi-tasking Forth system on top of which other operating systems can be built, particularly Unix, Mac OS X and Windows.

Providing a standard for Forth presents unique problems due to its extensibility. It must be able to provide a platform for a variety of uses, including robotic instrument control (the original use of Forth), embedded systems and use with modern desktop operating systems.

As operating systems get further and further apart from the hardware, which therefore becomes progressively more difficult to program (such as USB), the plea by the inventor of Forth, Charles Moore, for simplicity in computing becomes ever more valid. Development environments have become very complex and documentation is no longer available in book form; on-line documentation does not have the disciplined structure required of paper manuals and is awash with unexplained acronyms and buzz words, each requiring a link to find out what it means, only to be confronted by further links.

When it comes to hardware, Forth provides a unique capability for accessing hardware directly. In astronomy, this direct access to hardware is especially important, as recent spectacular scientific advances have been almost totally dependent on new technology, both engineering and applied physics. Currently, developments are eagerly awaited in superconducting detector arrays that will revolutionise the observation of the most distant galaxies.

Despite the fact that Forth provides a unique perspective on computing, it is ignored in university courses: there even exists a book, purporting to compare computer languages, that ignores Forth completely (Cezzar [2]). There appear also to be university courses which contain no practical electronics component. We see the results in a myriad of ways in poor programming of a variety domestic devices.

In what follows, where examples are given, they use the truncated form of stack comment used in Forth / 68 [3], which is used as a comparison in a number of places. Readers should find no difficulty, however, in interpreting their format.

1.1 Some basic principles

Common practice is not necessarily a good argument for including features in a standard. For example, in the first Forth systems, `LOAD` was used to interpret Forth blocks when compiling programs; the natural extension was to use `LOAD" ..."` for loading named files, yet `INCLUDE` was used instead. Likewise, `S"` was adopted in the ANS Standard for defining a string, when the argument for current practice was extremely weak, stating that some users used `"` and some used `S"`. It is not even clear why some users had departed from using `"` which had been common practice at that point, going right back to the original systems of Charles Moore. What is worse, it even looks rather fussy and ungainly.

The problem with using common practice as an argument is that there is a danger of 'standardisation by stealth'. Whose common practice is it anyway?

Similarly, having always done something wrong is not a good argument for continuing to do so if it is technically incorrect (Section 3.1).

It is also important to take note of developments outside Forth so that, where appropriate, similar techniques can be adopted within the Forth framework.

Giving preference to avoiding breaking of existing code over technical soundness is also not a good argument. For Forth/68, ANS Forth was the first standard Forth considered good enough to move from polyFORTH, which had been used by many programmers up to that point. To make the transition, much code had to be rewritten, i.e. was 'broken' [3]. Nevertheless, the changes were not too difficult to make, as Forth is relatively easy to change and, in particular, to debug.

2 Some general points

2.1 A name for the Standard

The Standard needs a name. The reasons are:

(i) ANS has been deleted in the text for the proposed Standard but programs conforming to the Standard cannot claim to be the only valid Forth programs;

(ii) A name is needed to identify which Standard applies, i.e. a new prefix in place of ANS.

2.2 Cost

Cost of purchase of the Standard needs to be reasonable. ANS Forth is outrageously expensive and does not even include a printed copy, an essential for browsing the Standard.

2.3 Quoting from the Standard

It is important to make clear just what can be quoted form the Standard for those of us who are not copyright lawyers. An email query to ANSI over whether or not the Pascal grammar could be distributed in machine-readable form elicited no reply.

2.4 System name

It is highly desirable to give each system a name, so that conditional compilation can be done depending upon which system is being run at the time. This could be implemented as

SYSTEM-NAME (*c-addr u*)
Return the address and count for the name of the Forth system.

3 Fundamental Changes

Some changes seem to be essential to the proposals for the Standard.

3.1 Scanning vs. parsing

The terms 'parse' and 'parsing' are used incorrectly almost entirely throughout the text for the Standard.

In both the treatment of natural language and the theory of computer languages parsing implies analysing the structure of a sentence or language construct. In computing it refers to checking that the tokens extracted from the source code of a program follow the grammatical rules of the language [3].

The correct term, as used elsewhere in computing, is 'scan', which is the process of extracting language tokens from source code. This suggests that the vast majority of the descriptive text in the proposed Standard needs to be changed from variants of 'parse' to their 'scan' equivalents.

We cannot change `PARSE` itself, as it is now cast in stone (the penalty for inaccuracy in setting up the ANS Standard). However, it is possible adopt `SCAN-NAME` in place of `PARSE-NAME`. It would also be useful to have a multi-line `SCAN`, possibly similar to that in Forth / 68, which bypasses comments.

The argument that it has always been done this way is surely not valid in the face of the fundamental usage of the term, parsing, elsewhere: Forth cannot ignore the outside world and, at least in this case, it is irrefutably common practice to use scanning for this type of process.

3.2 Word lists and vocabularies

Word lists were poorly designed in ANS, despite the intention of producing a set of primitives that could be used in a variety of schemes.

The reason for the problems with the Standard is that there are unnecessary hidden effects on the search order. The committee indeed accepted that the implementation of `ALSO` and `ONLY` with their hidden and not very intuitive effects was controversial.

It is difficult to see how this scheme was accepted for the Standard. The way it should have been is

FORTH (-- *wid*)
Return the *wid* for `FORTH`.

ONLY (*wid* --)
Given a *wid*, make it the only word list in the search order.

ALSO (*wid* --)
Add the *wid* argument to the top of the search order.

DEFINITIONS (*wid* --)
Set the compilation vocabulary to *wid*. Add *wid* to search order if not already at the top. Options on search order are to leave it as is or set it to what it was when the vocabulary was defined.

VOCABULARY (*$* --)
Define a new named word list that returns its *wid* on execution.

With named vocabularies, which (fortunately) ANS avoids, it is possible to set up a specific search order without using `ONLY / ALSO`. In EBNF

```
  ENDVIA { <vocab-name> } ENDVIA
```

The search order is set to the list of vocabulary names, last named to be searched first.

Even though the ANS Standard defines `CODE` and `;CODE` which provide links to assembler code, it is acknowledged that the actual way that the assembler works is system-dependent, as it must be on different processors. For this reason, `ASSEMBLER` should have been left out of the standard. In Forth / 68, switching to assembler code sets `ASSEMBLER` as a transient at the top of the search order, which is discarded whenever a new definition is added to the dictionary or `ENDCODE` is executed.

Had the above scheme been adopted in ANS (and would surely have been less controversial) it would have made `FORTH-WORDLIST` redundant and, maybe less so, `SET-CURRENT`.

Forth / 68 uses the above scheme, for example, in the MetaForth cross compiler used to compile the minimal kernel for the system and for the Pascal parser [3], which makes extensive use of vocabularies. There is a software switch for a change of mode to ANS-compatible, although it is never used in practice.

Fortunately, it is possible to make an ANS Standard program compatible with the above by using very simple redefinitions.

It is unfortunate that the treatment of word lists has been cast in stone. While I would like to see the ANS scheme deprecated and replaced by the above, it is probably impractical to do so. However, it does highlight the dangers when what was technically a poor choice was made for the Standard.

3.3 LOCALS| and {:

`{:` is a big step-up from `LOCALS|` not least because the number of local definitions permitted has been doubled, though it is not clear why it needs to be limited at all: the original eight seems to have been determined by the number of D-registers in an MC680xx processor, so that only two instructions were needed to transfer locals from the parameter stack to the return stack.

Also, it is not clear just why it should not be possible to access locals within, say, an `>R ... R>` segment after `LOCALS|`. For systems that place locals and loop parameters on the return stack, return-stack tracking must be used anyway, so there is no problem with access to locals. Likewise, if the locals are placed elsewhere, there is no problem anyway.

There is an inconsistency in the wording of the proposed Standard: in 13.3.3.2 d) it is implied that the return stack can be manipulated after `LOCALS|` variables have been defined, so long as return-stack balance is maintained; A13.3 implies that the return stack cannot be manipulated at all at this point.

The ability to include uninitialised variables in a `{:` list is a big plus, as it does not involve any transfer from the parameter stack.

However, the impression given in the Standard proposal is that `{: ... :}` would be used in place of the parenthetical comments normally used at the front of a definition. This is both very limiting and surprising, given the large number of named variables allowed. In Forth/68 `LOCALS|` are invariably preceded by some stack manipulation and `{:` would be used in the same context. In this way of using it, the `-- <out>` becomes irrelevant. In this system, the stack arguments are given in short form at the front of a definition, while the local names are more descriptive, see [4]. For example, we might use *n-indx* as a mnemonic in a stack comment but `grammarIndex` for its more descriptive `LOCALS|` name.

If large numbers of locals are to be allowed, or even relatively few descriptive names are to be used, it is essential that `{:` segments be able to cover more than one line; the same goes for `LOCALS|`.

The following changes are therefore suggested:

- Use the same order on the stack as with `LOCALS|`. Reasons are, firstly, that the Standard must be technically consistent. Arguments that the proposed order is common practice do not hold water here, as it is admitted that there are several ways of implementing the `{:` function.

  Secondly, this will 'break' existing code for those that use different schemes anyway, so that argument does not apply.

  Thirdly, if items are added to the stack as commonly happens before the locals are defined, it is easier to track backwards through the stack to identify the local names.

- The `-- <out>` should be dropped, as it is irrelevant to the function of `{:`, unless the Standard specifies that `{:` is to be used *only* as a replacement for the normal `( ... )` argument comments at the front of colon and `CODE` definitions; this would betray a very limited view of the capability.

- Given the large number of possible locals, it essential that both `LOCALS|` and `{:` be allowed to cover more than one line.

4 Code vectoring

It is much more common elsewhere in computing to use the term vectoring for the capability that `DEFER` is intended to provide.

All references that I have been able to find to deferring of code refer to deferring of the *time of execution* of the code, for example in Java, where a horribly complex way of doing this is needed. Of course, in multi-tasking Forth, this is both much easier to implement and read.

However, it is suggested that the correct term to use is vectoring, which is in common use. For example it has been used for many years for handling of interrupts, and is even given conceptually in MVP-FORTH where an example is shown of how to do this for `PAGE`. This concept has also been in use in Forth/68 for for many years for the `TYPE`-style routines, `TYPE`, `PAGE` and `CR`, as a group, also for error message generation for routines in the kernel, which are needed before support for error-message display has been loaded.

It is therefore suggested that `DEFER`, `DEFER@` and `DEFER!` be replaced by

`VECTORED, VECTOR@ and VECTOR!`

This has the advantage that `VECTORED` does not get confused with `POSTPONE`, which is easy to do with `DEFER`.

In this respect it is useful to have a definition

NULL
Take no action. This is the equivalent of the usual `NOP` in assembler.

It is useful as a default for vectoring to avoid crashes if the vectored code is called accidentally before the execute token has been set up.

5 Strings and characters

Given that there is very little provision for string handling in the Standard proposal, it is something of a surprise that so many of the proposed changes to the Standard involve characters, strings and key strokes.

For those of us who do not normally use such extensions, the Standard needs to provide much more explanation as to how the various changes are to be used and why they should be included in the Standard, rather than being treated as specific to particular applications.

Again, it is not at all clear why operations such as `REPLACES`, `SUBSTITUTE` and `UNESCAPE` should be in the Standard rather than part of an application. They seem to belong in very specialised applications.

5.1 String operations

Some suggestions for very basic but highly desirable extensions for string support are:

CATENATE (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ *u*$_3$)
Append the first string the end of the second but limiting the maximum length to u_3. This is particularly useful for putting together file path-name strings.

STRING-COPY (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$)
Copy a string across to a buffer, truncating the length to u_2 if $u_1 > u_2$.

COMPARE-TEXT (*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ -- *n*)
This is a case-insensitive version of `COMPARE`. It can be used for putting definition names in alphabetical order in `WORDS` listings but is also useful in other sorting situations.

These can easily be defined to work seamlessly with the Extended Character set.

5.2 Characters and bytes

There is a problem with the ANS Standard which seems to have been written to allow Forth systems to use character coding of fixed but unspecified length. It is presumably for this reason that `CHAR` has been used instead of the somewhat clearer ASCII.

However, Conklin and Rather [5] specify that `C,` should append a byte to the data space, which does not accord with the Standard. Similar considerations apply to `CMOVE` and `CMOVE>`, `C@` and `C!`. This assumption is made in several places in the text, despite the claim at the front of the book that it 'features Standard Forth'.

The limitation of a maximum of 255 characters in strings implies the use of `C@` to extract the length of a string for ASCII-based systems. Conklin and Rather make this explicit in a definition that they give for `COUNT`. However, this can all be generalised very neatly using definitions similar to those for vectoring, Section 4. Thus

COUNT (*c-addr*$_1$ -- *c-addr*$_2$ *u*)
This is the same as the ANS Standard definition, which returns the address/count for a string. The generalisation of this allows for the fact that strings might be implemented as, say, 16-bit counts with 8-bit characters. Also it allows for Extended Character strings, which might also have a byte length for the string following the count and which would be hidden from the user. The only requirement would be that the count itself would need to be character-aligned.

COUNT@ (*c-addr* -- *u*)
Fetch the string character count stored at the given address according to the system-dependent string count size.

COUNT! (*u* *c-addr*)
Store the string character count *u* at the given address according to the system count size.

This means that, to match the Standard, we need operations equivalent to `C@` etc. but that are in sizes that match the address units of the computer. It is suggested that these be prefixed by the letter 'B', which reflects the fact that the majority of processors use byte addressing (although this is not required):

B@ (*addr* -- *u*)
Fetch the item that is one address-unit wide stored at the address, unsigned.

B! (*u* -- *addr*)
Store *u* at the address, truncating its value if it is larger than the maximum unsigned number that will fit in one address unt.

B+! (*u addr*)
Add *u* to the address-unit value at the address, truncating the result if it is larger than the maximum unsigned number that will fit in one address unit.

While character operations may continue for bytes, their use should be deprecated and replaced by their B-equivalents for programs following the new Standard.

With this notation, `MOVE` should really have been `BMOVE`. However, accepting that it is an address-unit move, it needs to be generalised so that it can move bytes either up or down when the source and destination lists overlap; it is easy enough to implement this, as it only involves a check on the source and destination addresses. For speed, we need also to have a cell-aligned move.

MOVE (*$addr_1$ $addr_2$* -- *u*)
Move *u* addresss-unit values from *$addr_1$* to *$addr_2$*. The list of values may be moved to upper or lower addresses and may overlap.

CELL-MOVE (a-*$addr_1$* a-*$addr_2$* -- *u*)
Move *u* cell values from *a-$addr_1$* to *a-ddr_2*, with addresses aligned. The list of values may be moved to upper or lower addresses and may overlap.

5.3 Extended Characters Word Set

In what follows, it is assumed that the extended characters are used as the main input/output for the Forth system, rather than occasionally interpreting a file that uses the Extended Character Set. This is implicit in the Standard proposal, which indicates that `TYPE` and `ACCEPT` need to be generalised. The same must also apply to such definitions as `."`, `S"`, `C!` and `WORD`.

However, if different codings are to be supported on a given system, it is a simple matter to use code vectoring to switch between the character codings.

If we assume that the new definitions will also be found useful in systems using fixed character-sizes, then it is a simple matter to provide definitions that will work either within a fixed character-size system or one that uses the Extended Character Set. This means that X can be dropped as a prefix for the definitions.

As with the definitions mentioned above, most of the X-definitions can simply be generalisations of existing definitions. In particular

XHOLD, XCHAR+, XEMIT, XKEY, XKEY?, EKEY>XCHAR, XC,
--> HOLD, CHAR+, EMIT, KEY, KEY?, EKEY>CHAR, C,
These are straightforward generalisations of existing definitions.

XCHAR-, XC!+, XC@+ --> CHAR-, C!+, C@+
If these definitions are found useful for the Extended Character set, then equivalents are easy to define for fixed-width character sets.

XC!+? --> C!+?
This can again be implemented in fixed character-size systems, although it may be trivial in such cases.

+X/STRING, X\STRING- --> +/STRING, \STRING
These have obvious meanings in the context of fixed-width encoding but there are queries over the meaning of the stack arguments (see below).

X-SIZE, X-WIDTH, XC-WIDTH --> CHAR-SIZE, STRING-WIDTH, CHAR-WIDTH
All these have obvious meanings in fixed-width encodings and might be considered redundant but there is an advantage in being able to use the same definitions in different coding systems.

The above approach is much neater than the proposed Standard in that they are either generalisations of existing standard definitions or are definitions that can easily be applied to fixed character-size codings.

5.3.1 Buffers

In several of the definitions for the Extended Character Words, the term 'string buffer' is used. While the address of the buffer is reasonably clear, the meaning of the count/size argument is not. For example `XC!+?` has arguments `(` *xchar xc-addr$_1$ u$_1$* `--` *xc-addr$_2$ u$_2$ flag*`)` but *u$_l$* is not defined. Is it the length of the buffer? If so, where is the character supposed to go? Is there a hidden size or place within the buffer? Generally, at least two parameters are required in such circumstances: some measure of the size of the buffer and the size of any string currently within that buffer.

In `+X/STRING`, the length parameter appears to refer to the buffer size rather than a string within it, while `-TRAILING-GRABAGE` seems to refer to a string as such.

In `X\STRING-`, the description refers to the 'penultimate' character when it appears to be that the last one that is the only character that is relevant.

5.4 K-definitions and EKEY

Again it is difficult to see what relevance many of these definitions have in the context of an event-driven system: the function keys need to be requested explicitly and do not initiate any action of themselves.

However, there is a problem with this on the Apple Mac, where the operating system uses several of the function keys to modify the way in which windows are displayed. Even though Forth/68 intercepts the keyboard interrupt, OS X does not give any output on `EKEY` when these particular keys are struck.

There is a similar question over user aborts, which are an essential for computers that control robotic scientific instruments. Here the query is: should the user be able to generate an abort when `EKEY` has been called? In Forth/68, a user abort is generated for Command-Escape and is much easier to implement if it is allowed to generate an abort on `EKEY`.

Given the problems with function keys, it is suggested that the Standard include a phrase for `EKEY` that permits response to control keys to be implementation-dependent, especially as it may not be able to respond to the function keys. `KEY` is slightly different but does need to include a similar comment that the user may generate an abort after `KEY` has been called, at which point the abort takes preference.

5.5 Pausing dumps

The ANS Standard rejected the use of `X-ON/X-OFF` (ctrl/Q amd ctrl/S, respectively) ostensibly because different systems respond differently to these characters, even though it is heavily slanted towards ASCII and these control characters are considered to be a 'standard' use of ASCII coding. `X-ON` and `X-OFF`, or equivalents, are an essential part of a programmer's toolkit and should be included in some form in the Standard. System-dependent coding for pausing dumps would nevertheless be entirely satisfactory.

Similarly, for pe-emptive multi-tasking systems at least, some sort of abort key should be provided to generate a user interrupt, again system-dependent.

6 Records and other structures

The Standard suggests use of `BEGIN-STRUCTURE` and `END-STRUCTURE`, presumably taken from C (the hyphen is inconsistent with `ENDCASE` and `ENDOF`). However, since an object might reasonably also be considered a structure, a better name might be `RECORD`, as used in Pascal, coupled with `ENDRECORD`.

We can then follow what is similar to common practice elsewhere by making `RECORD` define a new word list to which to add field definitions, in a fashion similar to that given in the Standard proposal.

Thus we might have

`RECORD ( `*`$ -- addr 0`*`)`

This expects the name of a new record-type and assigns a new word list, making that the compilation word list. It leaves the parameter field address for the named record on the stack and an initial zero field offset.

`FIELD DEFINERS`

These take a form similar to that in the Standard proposal and place definitions in the wordlist/vocabulary for the record:

```
INT: defines a new (32-bit) integer field;
WORD: defines a 16-bit unsigned integer field;
BYTE: defines an 8-bit unsigned integer field;
CHAR: defines a character field;
SFLOAT: defines a single-precision floating-point field;
DFLOAT: defines a double-precision floating-point field;
(n) STRING: defines a string field;
n FIELD: defines an uncommitted field of length n;
<record-type> RECORD: defines an embedded record.
```

As each field is defined, the offset is first aligned in accordance with the data type and then incremented according to the data-item size. Each field definition includes a data size and has its offset within the record as a parameter.

There are questions over the string field. Options are to use an in-line counted string or a handle for a relocatable block containing a string. This is best left for discussion elsewhere.

ENDRECORD (*addr offset*)
Terminates the record structure by setting the data size for the record from the final field-offset.

6.1 Record Instances and dot notation

If RECORD defines a new record *type* that, when executed, returns the data size and word-list identifier for the fields in the record, then we can create an instance of the record in, say, Forth data space (e.g. the dictionary). We can then define an instance of the record:

```
<record-type> NEW-RECORD <instance-name>
```

When <instance-name> is executed, it returns the address of the instance data, along with the word-list identifier for <record-type>. The *wid* can be on the parameter stack, though there is a strong case for placing it in an (anonymous) user variable, as it never needs nesting. If, in addition, field names simply add their offset to whatever is on the stack, the way is open to use dot notation. For example,

```
<instance-name>.<field-name>
```

can be split into two tokens, each of which is processed separately.

In execute mode, the first token is executed and returns the record-instance address plus *wid*. The next token (i.e. field) is looked up in the word list and executed so that the address of the field is returned. If compiling, an address literal must be assembled that returns the address on the stack. This is very straightforward for systems using direct-execution / subroutine-threading in colon definitions but would need a little more thought in other forms of threading.

This would allow the use of forth accessors and setters for the fields, including TO.

An advantage of using a *wid* hidden inside a user variable is that we can write

```
<instance-name>
```

simply to return the address of the record without having to pop the *wid*, which then remains hidden away and unused. This would work in both execution and compilation modes.

The above can be used with a new word, FROM, as

```
FROM  <instance-name>.<field-name>
```

which would push the contents of the field onto the parameter stack. FROM is very similar to TO and would be system-dependent in the way it was implemented. For subroutine-threaded systems, this could be implemented as an assembler push to the stack, either as an address or a field value, allowing for the size of the appropriate field, e.g. extending character values to, say, 32-bit.

TO, with slight generalisation, could be used in the same way:

```
TO  <instance-name>.<field-name>
```

In both cases, the size of the data field would need to be accessible, which would modify the way in which the token processing took place but that happens anyway in the case of a `(2)VALUE` variable.

6.1.1 Object-oriented programming

Considerable interest has been shown in object-oriented programming at this conference, see Ertl [6] Schleisiek [7] and Haley [8]. The above can then readily be extended for this purpose. While this is not the place to put forward a full OOP proposal for Forth, the dot notation follows that used in C, Java and Delphi and is both technically sound and easily implemented in Forth.

For OOP, we could use `CLASS`, in place of `RECORD`, to define a root class and `SUBCLASS` to define a subclass of another class, given its name. `FROM` and `TO` could then also be applied to objects, whether they are addressed directly or, perhaps more usually, through a handle, in a very Forth-like way and would integrate well into existing Forth techniques for handling stack data.

A couple of points might be noted. Firstly, the distinction between *private* and *protected* class members is often over-egged by OOP followers. It is possible to get close to this by using `GLOBAL` and `LOCAL` definitions, as described in Section 8.5. *Private* can be implemented by calling `LOCAL`, *public* by calling `GLOBAL`, with `INVISIBLE-TO` at the end of the file removing the private definition names from the dictionary. Subclasses for the given class can access private members if they are defined within the same file. There would then be no equivalent for protected members accessible to subclasses defined outside the confines of the file (Forth / 68 could implement all three categories but the above scheme is simpler). In any event, having three levels of visibility is only a problem for bad programmers, which surely we don't have in Forth!

Access to inherited members, particularly methods, requires access to superclasses. This is not too difficult to do but would be system-specific in its implementation.

6.1.2 Advantages

By using dot notation, coupled with `TO` and `FROM`, we have a scheme that not only conforms to existing practice outside Forth but, at the same time, dovetails very neatly into the existing ways in which stack arguments are processed using existing Forth operators. It is also simple to understand and elegant to use. It is easy to implement in Forth in such a way that what the Forth programmer sees is system-independent, apart maybe from deciding on sizes of the integer fields.

It also has the added advantage, not possessed by the current `BEGIN-STRUCTURE` proposal, that field names are attached to particular record types and so can be used freely for definitions elsewhere in an application, including fields in other records. The scheme also provides a natural progression from support for records to the implementation of object-oriented programming.

7 Modern developments

Since most Forth systems are run on modern desktop operatings systems, some attention needs to be paid to these.

7.1 File handling

When including files, it is essential to be able to use both full and partial file names. For example in the file that is included when processing the LR parser described in [3], the string

```
S" ::Compilers:Pascal:Character Classes"
```

is used for loading the file containing the character classes for Pascal. This involves backing up by a couple of folders from that containing the Forth system, then coming back down through the Compilers and Pascal folders to reach the required file. In Mac OS Classic, this is achieved using colons.

It is necessary to standardise the way in which these strings are expressed so that it is possible to open files using the following:

(i) a full path name starting at the root directory;

(ii) a parth name relative to a directory backed up from the one in which Forth resides;

(iii) a simple name for a file located in the same directory as the Forth application.

The Apple Mac only needs one character to cover these requirements.

No case is made here for following Apple Mac practice. Rather, a standard way is required that allows path names to be converted to a form that works on a particular system.

Use of wildcards, as in Unix, could provide a considerable challenge on some operating systems and it seems unlikely that the Standard should attempt so support this way of specifying file names.

An important feature of GUIs is the ability to open a file using a system-dependent dialogue. It is therefore important for Forth to provide a standard means of bringing up such a dialogue.

`GET-FILE ( u-mode -- fileid ior)`
: Open a dialogue that opens a file by name using the read / write mode on the stack and return its *fileid,* zero if the user cancelled, and an io-result, zero if no error.

`LOAD-FILE`
: `INCLUDE` a file selected from the OS standard file dialogue. `INCLUDE-FILE` is already in use in the ANS Standard, so cannot be used here.

`INCLUDE` and `REQUIRE` cannot be used as defined, since file (and directory) names often contain spaces. Also file names can start with spaces, which are significant. Instead something like `INCLUDE"` is needed.

`INCLUDE"`
: Scan the input stream for a string delimited by a double quote. Open the file whose name is specified by the string, which can be an extended path name, and proceed as specified in `INCLUDED`.

`REQUIRE` has the same problem for file names as `INCLUDE`. However, there is potentially a fundamental flaw with `REQUIRE` in that the files are not necessarily loaded in a specified order so, if there is a name clash with definitions loaded in other files, there will be an ambiguity over which definition is the one that should be used. `REQUIRE` and `REQUIRED` therefore really need to be abandoned. Note that a close equivalent, `Uses`, in Delphi still requires the 'unit' names to be given in their correct order.

Also extremely useful for examining the contents of files is

`DUMP-FILE ( wid ud-start u-size)`
: Display the contents of a file in the normal `DUMP` format starting at location *ud-start* covering *u-size* address units

7.2 Relocatable blocks

Support for relocatable blocks is provided by the operating system for desktop computers, by means of handles, i.e. indirect addresses, for access. This allows the size of the a block to be changed, moving the block when necessary. For embedded systems, this can be accomplished by placing the block in the Forth data space and moving it explicitly when the size is increased but, again, it is necessary to use indirect addressing.

Relocatable blocks are very useful for structures whose size is not known beforehand, particularly for stacks and queues. Forth / 68 uses relocatable blocks for these purposes, especially dictionary headers, with control parameters placed at the start of each block. Stacks and queues may use fixed or variable sizes of data items and are automatically extended in size, when needed.

At the very least, basic definitions are needed for relocatable blocks:

`NEW-HANDLE ( u -- hndl rslt)`
: Allocate a relocatable block of size *u* address units. Return a handle and a result code that is zero if okay, non-zero if it has not been possible to allocate a block of the required size.

`FREE-HANDLE ( hndl -- rslt)`
: Dispose of a relocatable block, given its handle. Return a zero if the operation was performed correctly, non-zero if there has been an error, e.g. if the input argument is not a valid handle.

7.3 Menus

Menus are an essential part of GUIs and, indeed, had been used with RS-232 video terminals at the Royal Greenwich Observatory (RGO) well before Mac OS and Windows started to use them.

It is therefore important to have a standard scheme so that menus can be transportable. At present it is necessary to include a separate menu file for each system that a program is to be run on. Forth / 68 makes use of the primitive definition, ENTER, to enter names into the dictionary. As this expects a string address / count, it can enter names that include spaces. Each name within a menu is defined in a vocabulary that belongs to that menu, so the names can be looked up in the dictionary in a robust manner.

For example, in an application for renumbering digital images by capture date, in an Actions menu:

```
VOCABULARY Actions  Actions DEFINITIONS
S" Tag Movies..."   :ENTER   TAG-PREAMBLE  RENUMBER-FILES ;
```

where ENTER: is the 'colon' version of ENTER.

No case is made here for using this scheme in the Standard, although it is easy to implement. However, a standard method of setting up menus is highly desirable.

8 Miscellany

8.1 Interpretation Semantics

Some definitions don’t have interpretation semantics but have very obvious behaviour in that situation.

EXIT

This is very useful during development for exiting from compiling a file.

."

." has an obvious meaning for interpretation and is useful for displaying progress of a compilation. It also looks more elegant in a source-file when displaying text during compilation than the rather ugly .(in the ANS Standard, which is redundant anyway if ." is allowed to have interpretation semantics.

S"

Although S" has interpretation semantics in the File-Access Word Set, it does not in the Core Word Set. Essentially, this makes the two versions incompatible and is somewhat confusing. It is therefore suggested that the two versions be given the same interprettion semantics.

It is not clear why the ANS Standard adopted S" in favour of " since the latter had been used since Charles Moore's original Forth came out and surely could have been able to claim common practice at the time.

8.2 UNLOOP

In the ANS Standard, UNLOOP is a somewhat enigmatic definition in that it must be followed by either EXIT or UNLOOP itself and it is not clear what happens if neither of these follows.

However, there is a very useful feature if it can be followed by LEAVE. If used inside an inner nested loop, UNLOOP LEAVE would perfom the task of leaving the outer loop and would branch past its LOOP terminator.

More than one UNLOOP could be used in a similar way. This avoids what can be a somewhat complicated way of getting past the LOOP terminator in an outer LOOP.

8.3 EMPTY

EMPTY is essential in multi-tasking systems for emptying a terminal task dictionary, as it includes setting of defaults, such as search order, and closing any files opened by a previous application. These are functions that cannot be undertaken using MARKER which, in any case, would always need to leave a marker at the start of each terminal task.

8.4 Stack pointer access and SP@

Accepting that not all systems will have access to the stack pointer via SP@, it is nevertheless very useful on desktop and similar systems, where it will normally be available. This allows data structures placed on the stack to be accessed, an important feature if other languages are also supported. The fact that the stack is not visible on all systems should not prevent SP@ from being included in the Standard as an option.

In this context, it can be useful to have a definition that reverses the stack order, especially in those systems where the stack pointer is not visible:

>STACK< (*u*)
Reverse the order of the top *u* items underneath the argument on the parameter stack.

8.5 WORDS

To be usable, WORDS really needs the list of definition names to be able to be displayed in alphabetical order. For example, in MacForth, WORDS displays the definition names in chronological order, making it very difficult to see what words have been included in the dictionary. The problem is made worse if all words used in building the system are included in the list, as many of these are of no interest to the user.

Forth/68 includes some useful definitions that have been in use for many years and which discard the headers for selected words:

LOCAL
This is a compiler directive that causes subsequent definitions to be marked as local for discarding later.

|
Makes the next definition LOCAL, then retruns to the LOCAL/GLOBAL mode current at the time.

GLOBAL
Causes subsequent definitions to be marked to be kept visible.

INVISIBLE
Removes the headers for all definitions currently visible in the dictionary that have been marked as LOCAL. There is also an INVISIBLE-TO which makes LOCAL definitions back to a particular point invisible.

This means that the many field names for certain data structures, subroutine labels and Mac OS 9 assembler traps in Forth/68 can be discarded to help reduce dictionary 'entropy'. They therefore do not appear in WORDS listings.

It is important to note that the purpose here is not to reduce memory usage but instead to simplify the dictonary.

9 Number conversion

The proposals for converting numbers are very complex. Certainly, these numbers would look more elegant if the discriminator followed the sign.

I would also suggest using the prefix 0x or 0X for hexadecimal numbers as used in C. I have used this form for many years (common practice?). Other numbers, including octal and binary were usually accomplished simply by base-switching, since they are only used occasionally. It is not at all clear that all these different forms are really necessary.

10 Drawing tools

Since the Standard includes AT-XY for character-based positioning, it would seem desirable to include definitions for drawing simple lines. Forth/68 has

MOVE-TO (*u-h u-v*)
Moves the 'drawing pen' to a position in the output window, *u-h* pixels in the horizontal position and *u-v* vertical pixels, measured from the top left of the drawing area of the output window.

LINE-TO (*u-h u-v*)
This is similar to MOVE-TO but draws a straight line from the current position to that specified.

Forth/68 also has a definition that allows the output font to be set by name, FONT"; its size in pixels/points is set by PT. Other definitions allow the fore-colour and background colour to be set to some basic colours such as BLACK, RED and MAGENTA. The fore-colour specifies the colour that will be used for drawing lines and text, while the background colour specifies the colour for the general background.

11 Execution timing

It is often useful to be able to compare program execution times on different systems. A standard means of accessing a clock is needed so the execution speeds can be compared when the same definitions are run on different systems. Resolution does not matter too much but does need to be much less than one second. Mac OS 9 has a system variable called `'Ticks'`, that is measured in 60Hz clock ticks and is generally adequate for this purpose, although a resolution of one millisecond is preferable.

12 Aborting

In the control of what has come to be called robotic instruments, it is important to be able to abort in a controlled fashion. This is also generally useful if there is a chance that a definition will end up in an infinite loop.

One of the first requirements found with the RGO microdensitometer, [3], was the need for both pre-emptive multi-tasking and the availability of a controlled user abort from the keyboard. The practical need for this was that, if a scan were aborted without switching off the motors, these could run onto end stops and required manual rewinding of the drive screws, resetting of the power supplies and loss of registration of the carriage position, which could also mean having to discard any scans made previously. The multi-tasking had to be pre-emptive to ensure that the system would respond fast enough to the abort key.

The solution adopted was to implement an `ESCAPE...ENDECAPE` structure at the start of any definition that needs a controlled abort exit. For example, if `motor` were a `LOCALS|` variable to identify a motor that was being switched on and off, the phrase

```
ESCAPE  motor OFF  ENDESCAPE   motor ON ...
```

could be used. Although `CATCH/THROW` can potentially do this, it is not straightforward in use, the principal reason being that handling an exception is split between two definitions, making the caller responsible for an abort generated from within the called routine (including the case where the user might have pressed the abort key).

The `try...catch` mechanism, as implemented in languages, such as C^{++} and Java does put the `catch` in the same routine but the `catch` only catches specific exceptions.

The `ESCAPE` mechanism is probably the easiest to use. Firstly, it occurs in the routine where the exception is generated. Secondly, in Forth/68, the `THROW` code for an exception can be accessed through a user variable, `THROW-CODE`, so that special action can be taken for any given type of exception. However, this is not proposed for the Standard.

Since `CATCH` has been used in ANS Forth in a way that is different from other languages, it is suggested that the `TRY...FINALLY...ENDTRY` mechanism, similar to that in Delphi, be adopted for the Forth Standard. The above code would then be written as

```
TRY  motor ON ...  FINALLY   motor OFF  ENDTRY
```

The code following `FINALLY` is always executed at the end, whether or not there has been an exception. There are two options on the way in which `FINALLY` works. The first would be for it to return the `THROW` code on the parameter stack, in a manner similar to `CATCH`, zero if there has been no abort; the second would be for it not to return anything but to provide a user variable, like `THROW-CODE` above, which could allow different exceptions to be handled individually or, indeed, allow different actions depending upon whether or not there had been an abort.

Big advantages for this mechanism over `CATCH` are:

- Crucially, the exception handler occurs at the 'heart of the action' and is handled at the same level as the point at which the exception occurs;
- Code following the `FINALLY` follows on naturally from the code before it;
- It is not necessary to access the code before the `FINALLY` with the somewhat ungainly `['];`
- Any `LOCALS|` words can be used within the `FINALLY` code, including when there is an exception – these are not accessible with `CATCH`;
- At least in robotic instrument control, it is likely that the code following the `FINALLY` will be executed whether or not there is an exception.

References

[1] Forth Standards Committee. *Forth 200x Draft 11.1*. 29th February, 2012

[2] R. Cezzar. *A Guide to Programming Languages: Overview and Comparison*. Computer Science Library, Artech House. 1995.

[3] LG. van Breda. Building an LR parser for Pascal using Forth. *EuroForth* 2012.

[4] S. Pelc. Notation Matters. *EuroForth* 2012.

[5] E.K. Conklin and E.D. Rather. *Forth Programmer's Handbook*. FORTH, Inc. 1998.

[6] M.A. Ertl. Objects 2. *EuroForth* 2012.

[7] K. Schleisiek. Explaining simpleOOP. *EuroForth* 2012.

[8] A. Haley. Standardise Forth OOP. *Euroforth* 2012