REPTIL—Bridging the Gap between Education and Application

Israel Urieli

Ohio University
Athens, Ohio 45701

Background

REPTIL (a REcursive Postfix Threaded Interpretive Language) is a Forth-like language which has been designed with the specific motive of being a viable alternative language to bridge the gap between education and application. It has been developed over the past few years (1,2,3) mainly because of a dissatisfaction with the current de facto educational environment, in which one is expected to learn at least three languages before reaching an application maturity.

Many people have stated that the various aspects of REPTIL could be written in Forth, that there is a plethora of languages proliferating the universe and that we do not need an extra language to add to the Tower of Babel. They have completely missed the point. Whereas it is true that one high level language can be written in terms of another—Forth has been written in Ada (4), Pascal has been written in BASIC (5), Logo has been written in LISP (6), and so on—there are fundamental purposes of computer language that should not be forgotten:

Purpose 1—A computer language should enable meaningful communication between humans and computers.

Purpose 2—A computer language should enable meaningful communication between humans.

The word 'communication' is meant here in a deeper 'Buberian' sense of dialogue and inner understanding, rather than that of debate, or a master-slave command language (7). Over the past two decades there has been a serious effort to promote Purpose 2, i.e. communication between humans. This has been done mainly by introducing the concept of structure, as epitomised in Pascal. Unfortunately all this is at the expense of Purpose 1; thus the user of a UCSD Pascal system may never even be aware of the fact that the language is compiled into a stack-oriented P-code interpreter (that is until one day she is confronted with a 'stack overflow' error). One of the reasons for a preference of C over the more readable Pascal (apart from being fashionable) is because it promotes a somewhat more direct contact with the computer. With this in mind I wish to propose two hypotheses:

Hypothesis 1—Traditional languages have been designed to form an 'Orwellian' barrier between human and computer (8).

In this sense the computer is presented as the language, and the richness or limitations of that language will inspire or limit the human cognitive processes in using the computer. As an example, FORTRAN is the current mainstay of the engineering world; however because parameter passing is by reference only, recursion is not allowed in FORTRAN. Thus FORTRAN promotes the traditional problem solving approach which is to explicitly isolate the unknowns in terms of some (often nebulous) function of the knowns, and thus divorce the problem space from the solution space. Those engineers who have been exposed only to FORTRAN will never even consider a recursive, or goal directed approach to problem solving

The current delight of the education world is Logo, which presents the computer as a simple turtle. However this simplicity belies the cumbersome complexity required to contort the computer to this image. It does allow rich concepts such as extensibility and recursion, however the user is barred from understanding or experimenting beyond the language. The current microcomputer implementations of Logo are slow, and trying to redefine or even examine primitives is considered taboo. Because of these limitations one soon outgrows Logo into something more general and useful such as BASIC which does not have the richness of Logo but is concise, accessible, and will even allow the masochist to 'PEEK' and 'POKE' around in memory. Unfortunately, for the enlightened world BASIC is a 'no-no', the Advanced Placement Test requires Pascal, and by the time our budding youth has reached the mature adult world of FORTRAN, C, Forth, Ada, Modula-2, Pascal, LISP, PL/1, Cobol (not to mention the omnipresent OS–JCL, VMS, UNIX, CP/M, MS-DOS), she has understandably reached a state of utter confusion.

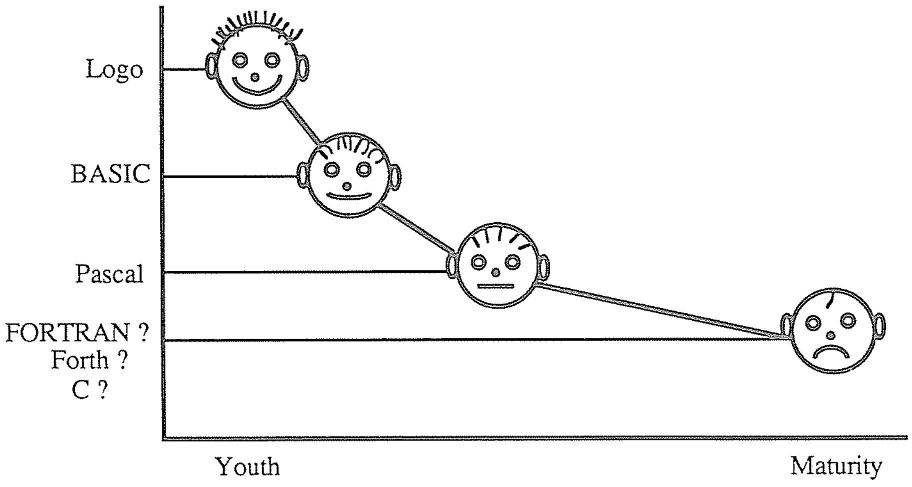

Figure 1. The de facto 'ideal' educational environment

What is the underlying reason for this confusion? I believe that it is because the math education system (which has been inherited over generations) has not yet accepted the fact that understanding computers requires a different approach to math. We are not yet ready to accept the fact that our sages misinterpreted the thumb for a finger, and thus gave us the decimal system, when in fact our creator meant each hand to be a hexadecimal digit, with the thumb being used as a carry (or overflow) bit. That when, as infants, we build with blocks, we are in fact developing an appreciation for stack manipulation–with the Top Of Parameter stack (TOP) allowing orderly access of parameters, and attempting to access the bottom of stack causing chaos (usually gleeful–especially in a supermarket!).

The conventional computer languages (such as BASIC) attempt to cover up this fundamental aspect of math by emulating our questionable infix and decimal heritage. Furthermore, this emulation is not complete. Thus we create confusion when we say that `N = N + 1` is an acceptable statement in BASIC (is this not the way we were taught the concept of proof by contradiction?), whereas `2 + X = Y` is unacceptable, since in our zeal we have distorted the meaning of the equal sign.

Hypothesis 2–Forth is the first language designed to break the 'Orwellian' barrier.

In Forth, for the first time, we have a small, simple but powerful system in which the computer is not only presented as the language, but it is the language. Nothing is hidden–every verb can be examined, molded, extended, and even forgotten. We can truly state that Purpose 1 has been satisfied–with Forth, there are no communication barriers between human and computer. Why then, has Forth not become the language to break the confusion and replace all other languages? Forth and Pascal have much in common: both were developed by one person at about the same time, and both out of a dissatisfaction with existing languages. Pascal, designed for education, succeeded admirably in fulfilling Purpose 2 (at the expense of Purpose 1) and was thus accepted by the multitudes, whereas Forth, designed for application, succeeded admirably in fulfilling Purpose 1 (at the expense of Purpose 2) and was thus accepted by the enlightened few. Because Forth programs do not attempt to emulate the human environment, they are generally considered unreadable by humans. They demand a strict self discipline (which is seldom apparent) on the part of the programmer concerning documentation and style. However even with that, since Forth is an evolved (rather than defined) language, written mainly by one person for one person, it suffers from a number of inconsistencies which make it pedagogically unsound. REPTIL is an attempt to overcome these limitations–not only with cosmetic changes such as a more meaningful symbology, but with some restructuring as well.

The creation of REPTIL has the advantage of hindsight, and its specification and development have been influenced by many different languages, suggestions, and language approaches. The token threaded infrastructure, decompiler and editor is based on RTL (9,10); many of the ideas of the defining verbs, prompts and basic structures are based on PISTOL (11); the conditional branch structure is based on FORTRAN 77; the conditional loop with multiple exits is based on Ada; various other factors have been influenced by C, Logo and of course Forth. Putting it together has resulted in a unique combination of these various influences, and it is hoped that REPTIL can bridge the gap between Purpose 1 and Purpose 2, and thus between education and application.

The various advantages of REPTIL can be summarized as follows:

1. The postfix and stack notation is made more readable and palatable.
2. The prefixed quote notation enables a more consistent and context-free postfix syntax.
3. There is a minimal (four) but nevertheless complete and powerful set of structured constructs, all of which can be invoked in the interpretive mode.
4. Recursion is intrinsically available.
5. There is no source code stored in any form in the system; if required, source code is recreated in a structured format from the list of tokens.
6. The entire language, including the system verbs, is defined in terms of REPTIL algorithms and mainly coded in REPTIL.

In the following we discuss advantages 1 through 4 in depth, as well as describe the inner interpreter and token threaded infrastructure of REPTIL. Advantage 5 concerns the structured decompiler `UNDO:` and line editor `REDO:`. Since there is no source code stored the concept of source code screens and the associated virtual memory (which is fundamental to Forth) is irrelevant in REPTIL. Saving a current version of REPTIL on secondary storage is accomplished by saving a memory image of the compiled code. There are 30 system verbs making up the decompiler/editor; however they will not be described in this paper.

Advantage 6 concerns the various system verbs associated with the Outer Interpreter. The entire language (about 215 verbs) is defined in terms of REPTIL algorithms, and, apart from 47 primitives, is coded in REPTIL. In the current implementation on the Apple II computer, REPTIL is being used as a self-referential research tool; thus execution speed is knowingly compromised. All of the system verbs are visible, so as to enable meaningful decompilation, and comments are compiled as literal strings in the code. Thus for example the operating system verb `RUN` (equivalent of Forth's `INTERPRET`) can be decompiled and examined with the verb `UNDO:` as follows:

```
<16>     'RUN UNDO:
'RUN DO:
     LOOP[
          R^RESET  \   Reset the R-stack pointer.
          READ-LINE
          DO-LINE
     ]ENDLOOP
:END
<16>_
```

The system verbs making up the Outer Interpreter will not be described in this paper.

The Prompt and Stack Notation

In the interpretive mode, the input line begins with a highly informative prompt showing the system status, as follows:

```
n<b s>_
```

where `n` is the number of elements on the Parameter stack (P-stack).
`b` is the current radix base (displayed in decimal).
`s` are various symbols indicating that structured constructs have been opened and that the system is therefore in the compile mode. These symbols are used for nesting syntax checking.

Thus for example in a typical session:

```
<16>         ( 1 2 + , 3 4 + , 5 6 + )
3<16>          STACK?  \   display the three P-stack items in base 16
3  7  B  TOP
3<16>     DEC  =    \   convert to decimal and pop-and-display TOP
11
2<1Ø>     =?   \   are they equal?    \   =
Ø
<1Ø>_
```

The first line includes special grouping verbs, being the deferred execution parentheses `(, )` (described in detail below) and a no-op (`,`), used mainly for enhanced readability. Three items are pushed to the stack, as shown in the following prompt. The nondestructive stack display verb `STACK?` causes the three items to be displayed on the next line, with an indicator TOP showing the position of the Top of Parameter stack. The verb `DEC` converts the radix to decimal base and the verb `=` (equivalent to the Forth verb '`.`') pops the TOP. There is no confusion between `=` and the relational operator `=?` which checks for equality, since all relational operators in REPTIL have an appended question mark. Notice that (unlike Forth) the response of the computer is always on the following line. Thus there is no ambiguity as to which items were output by the computer, and which input by the user. Notice also the backslashes '\' which enclose comments.

There are three stacks in REPTIL, the Parameter (P-stack), Return (R-stack) and Symbol (S-stack). The Parameter and Return stacks are equivalent to those used in Forth. The Symbol stack contains the various symbols `s` shown in the prompt, and is used for both nesting syntax checking and determining whether or not the system is in the compile mode. This is described in detail in the sections following.

Quote – the Token Notation

Prefixing a name with a quote is used in some Forth-like languages, notably STOIC (12), PISTOL (11), and SPHERE (13). REPTIL is fundamentally a token threaded language, and has adopted the token threaded infrastructure of RTL (9). The prefixed quote is used as the token notation. Referring to figure 2, REPTIL is divided into five major entities, being a `NAMES` block containing all the names, a `NAMES^` table pointing to the various names, a `CODES` block containing all the codes, a `CODES^` table pointing to the various codes, and a `CELLS` block (not shown – including variables, buffers and stacks). Variable `EON` (End Of Names block) holds the first available memory space at the end of the names block, and similarly variable `EOC` (End Of Codes block) holds the first available memory space at the end of the codes block. The tokens consist of even numbered offset pointers varying from `Ø` to `EOT` (End Of Tokens), which link the names with their respective codes.

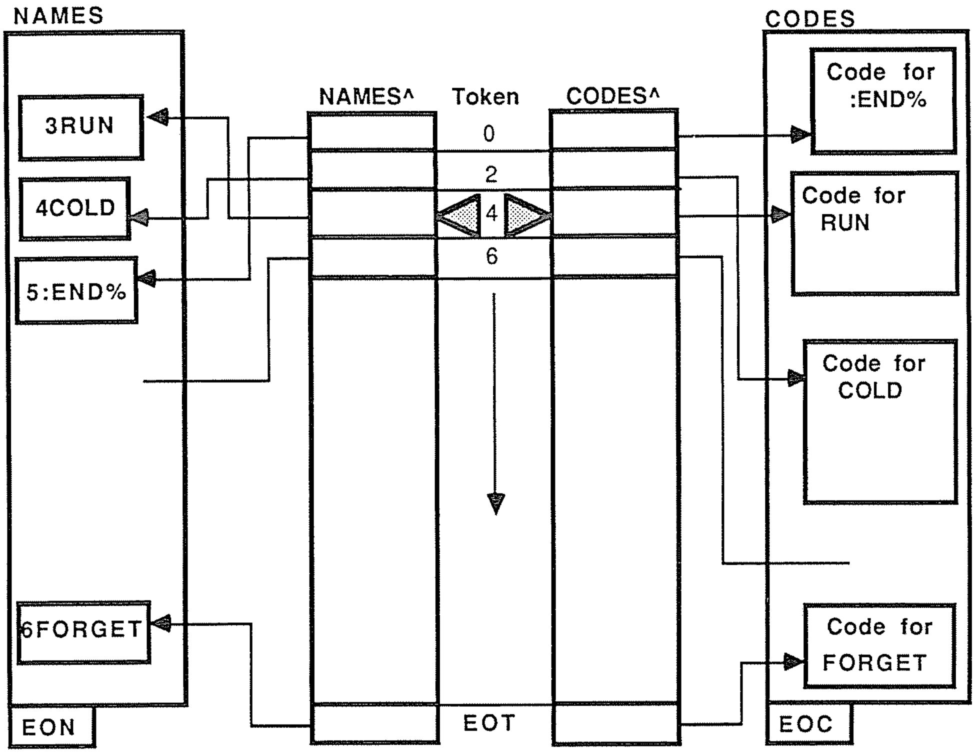

Figure 2. Token Threaded Infrastructure

A unique aspect of REPTIL is the association of the prefixed quote with the verb token. This gives the quote a fundamental significance similar to that of LISP, or Logo, as follows:

1. The quoted name is appended at `EON`.
2. If on searching through the list of names the name at `EON` is found to be a predefined verb, then its token is pushed to TOP and its execution is suppressed.
3. If it is not a verb, then it is a candidate for becoming a verb. The value −1 is pushed to TOP.

Consider the following example session:

```
<16>      'RUN   =     \    pop the token of RUN
6
<16>     'VALUE   =    \    not a predefined verb
-1
<16>      EON  $=      \    display the string at EON
VALUE
<16>_
```

With this quote prefix notation, variables and constants are created and handled in a unique, readable manner. The defining verb `:IS` creates a constant (it IS a value, and always will be) whereas `:HAS` creates a variable (it HAS a value cell which can contain a value, if so assigned).

In REPTIL, the content of a variable value cell (or the value of a constant) is accessed simply by invoking it. There have been many arguments as to the improved readability that this approach provides (14,15). Unfortunately, the associated extensions to Forth, including the `TO` or `QUAN` verbs, introduce a further undesirable context dependency of Forth verbs. This is avoided in REPTIL by using the quote notation. It could be argued that the prefixed quote also introduces a context dependency—that of suppressing the normal verb action. However the quote is more universally acceptable—it is used in the same context as in LISP, is applicable to any verb or name (not only variables) and adjoins the name, rather than being itself a separate verb. Thus the assignment of a value to a variable is achieved by supressing its action by means of a prefixed quote and using the assignment verbs `<-` (into) or `<+` (addto). Similarly the address of the variable's value cell is obtained by using the verb `CELL`. All of this is shown in the following example session:

```
<16>     'VALUE   :HAS              \  create the variable VALUE
<16>     5   'VALUE   <-            \  assign 5 to VALUE
<16>     42   'THE-ANSWER   :IS     \  create the constant
<16>                                \  THE-ANSWER
<16>    THE-ANSWER   'VALUE   <+    \  add 42 to the content
<16>                                \  of VALUE
<16>     VALUE   =   \   pop the content of VALUE
47
<16>      'VALUE  CELL   =          \  pop the cell address
<16>                                \  of VALUE
40EC
<16>   2  'BASE   <- \  change the radix base to binary
<2>  THE-ANSWER   \   to life and everything (in binary)   \  =
1000010
<2>      VALUE  'THE-ANSWER   <- \    simply experimenting...
THE-ANSWER HAS NO VALUE CELL
1<2>     DROP   HEX
```

We notice in particular that attempting to assign a value to the constant `THE-ANSWER` results in a polite but firm rejection. Values can only be assigned to variables. Any doubt as to what type the verb is can be resolved by the `UNDO:` verb as follows:

```
<16>      'THE-ANSWER    UNDO:
IS   42
<16>     'VALUE    UNDO:
HAS   47
<16>_
```

The use of the quote in the compile mode is shown in the following example of the definition of a new verb `BIN` to change the radix base to binary:

```
<16>          'BIN  DO:   \   binary radix
1<16:>            2  'BASE  <-
1<16:>        :END
<16>_
```

The `DO:` `:END` verbs form the defining verb pair in REPTIL (somewhat equivalent to Forth's `:` `;`) and are described in detail below. The name `BIN` is first treated as a candidate for a verb and is appended at `EON`. If `'BASE` were invoked in the interpretive mode, then the token of the predefined system variable `BASE` would be pushed to the P-stack. However, since `'BASE` is being invoked in the compile mode (notice the defining mode symbol `:` in the prompt), the runtime token literal handler `TOKEN%` is first compiled followed by the token of `BASE`. During runtime the token handler will push the following inline token to TOP for subsequent assignment by the verb `<-` (into). It is important to note that all this is transparent to the user, who is using the assignment statement in the compile mode in the same manner that she would in the interpretive mode. The complete action of the prefixed quote is shown in the Glossary. The dictionary search for the token of `BASE` is done during compilation rather than during execution, enabling a readable postfix notation combined with runtime efficiency.

The basic structured construct—deferred execution ()

The use of parentheses to define a structure is fundamental to many languages, such as LISP, Pascal (**begin end**) and C (`{ }`). Stack oriented languages such as Forth do not require a parenthesis-like structure for their operation. Recently, however, there has been a growing awareness that for all its power, Forth is simply not readable (or writeable) enough for widespread acceptance. Glass (16) proposed introducing parentheses and the comma into Forth as simple unstructured grouping no-ops for enhanced readability. Bergmann (17) presented a convincing argument for introducing parentheses as structured constructs in Forth-like languages, with full nesting syntax checking, and this approach has been adopted in REPTIL. Consider the following interactive example:

```
<16>        ( 1  2  +  ,  3  +  ,  4  +  ,
<16(>          5  +  ,  6  +  )  2  *  =
2A
<16>        (  (
<16((>    )  )  )
NEEDS (
<16>_
```

Whenever a structure is opened, an appropriate symbol (in this case the open parenthesis) is placed on the S-stack and indicated in the prompt. The system is placed in the compile mode and the nesting level is increased by one. All the following verbs entered are included at `EOC`. Note that the comma '`,`' is simply a no-op for enhanced readability as proposed by Glass (16). The parentheses can be nested to any depth, and the nesting level is indicated by the number of symbols in the prompt. Symbols are removed from the S-stack only if the appropriate closing structure verb is entered (in this case the close parenthesis). When the nesting level reduces to zero then execution is invoked, the system returning to the interpretive mode. Thus the parenthesis structure is referred to as the 'deferred execution' structure. The system verb `COMPILE?` simply checks for any symbols on the S-stack, as follows:

```
'COMPILE?   DO:
   S^     \   TOS (Top Of S-stack) pointer
   S^Ø    \   BOS (Bottom Of S-stack) pointer
   >?     \   Iftrue, then there is at least one symbol
          \   on the S-stack
 :END     \   T/F  |P
```

The main purpose of the parentheses structure is enhanced readability. It makes pedagogic sense that one should not be forced to execute a partially complete structure simply because the end of line has been reached. It is also the simplest of all the structures and the easiest to understand, allowing one to develop and study the underlying processes of structures without being encumbered with the relative branching requirements of other structured forms. In isolated cases the 'deferring' of execution would make a difference in the outcome, as in the following example:

```
<16>     " HI " , " BYE "   \   save two strings (maybe!)
2<16>   $=   $=    \    string-pop both strings...
BYE  BYE
<16>   Whatever happened to 'HI'?
Whatever  ?
```

Thus even though 2 addresses are indicated on the P-stack, both refer to EON (End Of Names), where 'BYE' has overwritten 'HI'. This can be prevented by using parentheses to force the system to temporarily compile the two strings sequentially at EOC, as follows:

```
<16>     ( " HI " , " BYE " )
2<16>  $=   $=
BYE  HI
<16>_
```

The deferred execution verb set includes the four verbs (,), (% and)%, as follows:

```
'( DO:  NOW
    COMPILE?     \  if not yet in the compile mode, then...
    ?IFFALSE     \  save EOC as a subsequent execution pointer
          EOC   'EOC-SAVE   <-
    ?ENDIF
    28    >S     \  '(' to the S-stack, increasing nesting
                 \  level by 1
    COMPILE:  (%   \  for meaningful future decompilation.
 :END
```

The verb EOC-SAVE saves the start of compilation address. Thus when compilation is complete then the value of EOC before compilation will be restored and execution will begin at this address. Notice the verb NOW which converts the (verb to the immediate mode. This is more readable than the Forth equivalent IMMEDIATE which is only invoked after the verb has been defined. The COMPILE: verb is only effective in the compile mode. It is one of the very few verbs which introduce a context dependency in REPTIL, in that the inline verb following is not invoked, but included at EOC. This is done by popping the following token address from the R-stack (the one we wish to compile), including the token, and incrementing past it before returning it to the R-stack, as follows:

```
'COMPILE:   DO:   #VERB
    R>  DUP      \  addr addr |P (of the following
                 \  inline token)
    2  +  >R     \  addr |P (increment past the address
                 \  of the token)
    FETCH        \  token |P
    INCLUDE      \  token at EOC rather than invoke it
:END
```

The immediate verb #VERB is a decompiler structure directive indicating that UNDO: should output the verb following COMPILE: on the same line. The default action of UNDO: is that the various verbs are decompiled onto different lines.

The verb) reduces the nesting level by popping a '(' symbol from the S-stack (if one exists). If it finds that the S-stack is now empty, then it invokes execution with the verb GO-EOC. If the TOS (Top Of S-stack) did not contain a '(' symbol, then a warning message is displayed. This is done without prejudice, however, and execution continues normally with the unmatched) verb being ignored.

```
') DO:  NOW
   S   28   =?    \   '(' on TOS?  (Nesting syntax check)
   ?IFTRUE
      S>   DROP   \  drop one level of nesting, and check if...
      COMPILE?   \   ...still in the compile mode?
      ?IFTRUE
           COMPILE:   )%    \   for meaningful future
                            \   decompilation.
      ?ELSE    \    no longer in the compile mode, so...
           GO-EOC  \  execute compiled code at EOC-SAVE
      ?ENDIF
   ?ELSE   \   incorrect nesting
      "  NEEDS  ("   $=
      NULINE   \   scroll the terminal screen and...
      COL-RESET    \    reset the screen column to left margin
   ?ENDIF
:END
```

The verb GO-EOC first completes compilation by compiling the verb :END% (every threaded verb must end with :END%), and then restores EOC so as not to permanently save this temporarily compiled verb. (Only the DO: :END structure is allowed to permanently compile a verb). Execution is invoked simply by pushing the start address at EOC-SAVE on the R-stack.

```
'GO-EOC  DO:
    COMPILE:  :END%    \    complete the compilation
    EOC-SAVE  'EOC  <-    \   restore EOC
    EOC-SAVE  >R    \   and prepare for execution
:END
```

The verbs (% and)% are no-ops, and are used exclusively by the decompiler UNDO: in order to correctly indent the structure according to the decompiler directives %VERB, RIGHT> and <LEFT.

```
'(%   DO:   %VERB   RIGHT>
 :END

')%  DO:   %VERB   <LEFT
:END
```

For example, assuming we wish to determine the value of π in terms of the rational fraction approximation 355/113. We may define a verb PI as follows:

```
<16>     DEC     \    change to decimal base
<1Ø>     'PI   DO:
1<1Ø:>      (  1ØØØØ  355  113
1<1Ø:(>    :END  \  simply testing the nesting syntax checker...
NEEDS  DO:
1<1Ø:(>     )  */     \    complete the definiton
1<1Ø:>      :END
<1Ø>    PI    =
31415
<1Ø>      'PI   UNDO:
'PI  DO:
   (
    1ØØØØ
    355
    113
   )
   */
:END
<1Ø>_
```

Notice that when we tried to complete the definition before closing the parenthesis, the computer politely rejected the attempt and allowed us to complete the definition without prejudice. The verb PI is decompiled in a structured manner showing clearly the grouping of the three numbers by the parentheses.

The Conditional Branch structure

Logical relations are always associated with a question. Is A greater than B? Is X less than zero (negative)? and so on. We therefore naturally append a question mark to all REPTIL relational operators. The conditional branch structure checks the condition on TOP (whether TRUE or FALSE) to determine the subsequent action to be taken. The conditional branch structure in REPTIL is more powerful and readable than the standard IF ELSE THEN of Forth, and is similar to that available in FORTRAN 77. Consider for example a simple verb defined in Forth which incorporates two IF structures, one nested inside the other. When invoked, the verb will print one of three grades, 'Fail', Pass', or 'Distinction' depending on a score on top of stack (18).

```
:  GRADE  DUP  4Ø  <  IF
                          ."  Fail"        (  Less than 4Ø )
                          DROP
                      ELSE
                      7Ø  <  IF
                                 ."   Pass"  ( 4Ø-69 )
                             ELSE
                                 ."  Distinction" ( Greater than 7Ø )
                             THEN
                      THEN  ;
```

Intuitively, this structure should not have more than one nesting level, since there should be no structural difference between the three grades. In REPTIL the verb GRADE would be defined as follows:

```
'GRADE  DO:  \    n  |P
   DUP  40  <?
   ?IFTRUE
       "  FAIL"  $=
   ?ELSEIF
   DUP  70  <?
   ?IFTRUE
       "  PASS"  $=
   ?ELSE   \   must be >=  70
       "  DISTINCTION"  $=
   ?ENDIF
   DROP
:END
```

We note the following:

1. The general form of the conditional branch structure includes the verb ?IFFALSE. The verbs?IFTRUE and ?IFFALSE are structurally interchangeable.
2. The single ?ELSE clause is optional.
3. As many ?ELSEIF-?IFTRUE clauses as are needed can be optionally used.
4. Full nesting syntax checking is done, the symbol for ?IFTRUE and ?IFFALSE being a '?', for ?ELSE an 'E' and for ?ELSEIF a 'F'. The ?ENDIF pops all the relevant symbols from the S-stack, reducing the nesting level by one.
5. Other conditional branch structures (or any other structures) can be wholly nested within any of the clauses of the basic structure. The conditional branch structure is not restricted to the defining verb and can be used in the interactive mode, similarly to the deferred execution parentheses. When the level of nesting is reduced to zero then execution will begin.

REPTIL is a fully recursive language, and the conditional branch structure enables full advantage to be taken of its recursive nature. Consider for example the recursive definition of the factorial verb '!':

```
<16>        '!  DO:    \    n |P
1<16:>           DUP  1  <=?    \    the stopping rule
1<16:>           ?IFTRUE
2<16:?>             DROP 1
2<16:?>          ?ELSE
2<16:?E>            DUP  1  -    \    n,  (n-1) |P
2<16:?E>             !    \    n,  (n-1)!  |P  (the recursive step)
2<16:?E>             *    \    n*(n-1)!  |P  (recursive definition
2<16:?E>                  \    of n!)
2<16:?E>         ?ENDIF
1<16:>         :END
<16>  DEC
<10>  7 ! =
5040
<10>_
```

The eleven verbs making up the conditional branch structure are ?IFTRUE, ?IFFALSE, ?ELSE, ?ELSEIF, ?ENDIF, ?END%, ?ENDIF%, ?IFTRUE%, ?IFFALSE%, ?ELSE% and ?ELSEIF%. The first five verbs are the user verbs. The verb ?END% is invoked by ?ENDIF to resolve branch addresses. The verb ?ENDIF% is a runtime no-op and is used as a decompiler directive and the last four verbs are runtime primitives which execute the various branches. Figure 3 shows the relationship between the five user verbs and the various system runtime verbs which they compile.

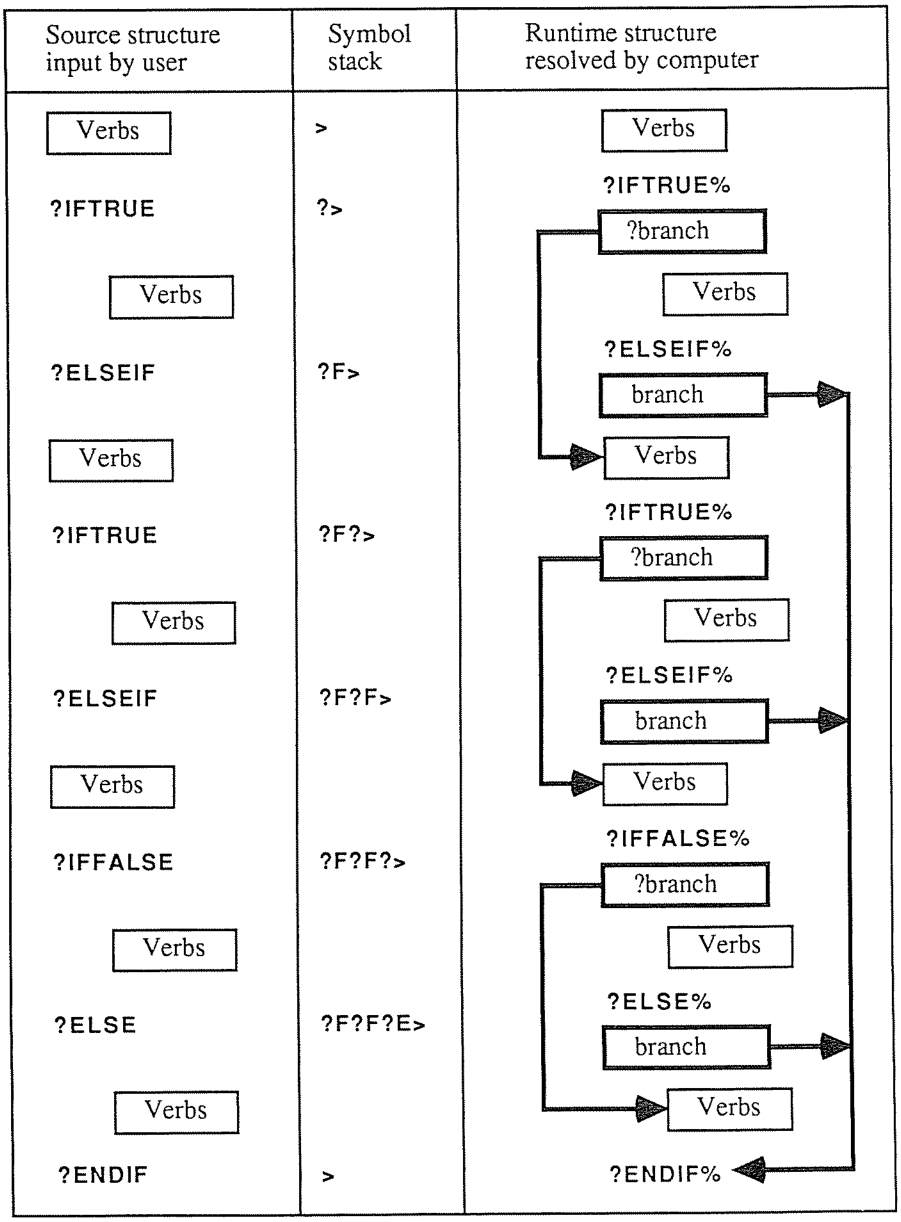

Figure 3. Conditional Branch Structure

The coding of the seven threaded secondary verbs is given in the Appendix for reference.

In the following we present a simple application example to illustrate the goal directed approach to programming that REPTIL promotes (3). Euclid (who was born at about 325 b.c.) devised the following recursive algorithm for finding the Greatest Common Factor (GCF) between two positive integers m and n:

GCF(m,n) is m if n = 0 (stopping rule)
  otherwise it is GCF(n, m **mod** n)

Thus for example GCF(8,12) → GCF(12,8) → GCF(8,4) → GCF(4,0) → 4

The REPTIL equivalent verb could be defined as follows:

```
<16>        'GCF   DO:         \    m   n  |P
1<16:>             DUP   =Ø? \    m  n  T/F  |P
1<16:>             ?IFTRUE   \    stopping rule
2<16:?>                DROP \    m |P (the required result)
2<16:?E>           ?ELSE      \    recursive step
2<16:?E>               SWAP \    n  m   |P
2<16:?E>               OVER \    n  m  n  |P
2<16:?E>               MOD   \    n  (m  MOD  n)  |P
2<16:?E>               GCF   \    and try again...
2<16:?E>           ?ENDIF
1<16:>          :END
<16>          DEC
<1Ø>          357  629  GCF  =
17
<1Ø>_
```

Thus as long as full stack commenting is included in the definitions, programming in REPTIL becomes intuitively obvious. It has been argued that stack manipulations can never become intuitive (particularly when there are more than two parameters involved), and REPTIL has been criticised for not including local variables. My response is that REPTIL is a fundamental core, and that if it becomes necessary to enhance that core in the future then that should be done by extending the core with various application shells of verbs. Throughout the development of REPTIL I have not found it necessary to specify local variables.

The loop structure - `LOOP[ ]ENDLOOP`

One of the peculiarities of Forth is the large number of loop forms available. In an interesting empirical study Soloway et al (19) showed that a single loop form with an arbitrary exit condition most closely links the programming strategy with a preferred cognitive strategy. Thus REPTIL provides only one loop construct having optional multiple exit conditions, similar to the conditional loop form of Ada. There is no directly equivalent loop form available in Forth (due to the capability of an arbitrary number of conditional exits), however all of the Forth conditional loop forms can be simulated. When no exit verbs are used then the resulting loop form is an infinite loop, an example of which in REPTIL is the outer interpreter verb `RUN`. A strong case can still be made for the Forth indexed `DO-LOOP` form, and this may be provided as a future REPTIL extension.

As with all the REPTIL structured forms, the loop structure can also be used interactively (and not only within a definition), as is shown in the following example session using the factorial verb '!' defined above:

```
<1Ø>        8  LOOP[
2<1Ø[>          DUP    =     " != "    $=
2<1Ø[>          DUP   !    U=
2<1Ø[>          1   -    DUP   <Ø?
2<1Ø[>        ?EXIT
3<1Ø[X>         NEWLINE
3<1Ø[X>       ]ENDLOOP
8!= 4Ø32Ø
7!= 5Ø4Ø
6!= 72Ø
5!= 12Ø
4!= 24
3!= 6
2!= 2
1!= 1
Ø!= 1
1<1Ø>_
```

The seven verbs which make up the loop structure set are LOOP[, ?EXIT,]ENDLOOP, EXITS%, LOOP[%, ?EXIT% and]ENDLOOP%. The first three verbs are user verbs. The verb EXITS% is invoked by]ENDLOOP to resolve the exit branch addresses. The verb LOOP[% is a runtime no-op and is used as a decompiler directive, and the last two verbs ?EXIT% and]ENDLOOP% are runtime primitives which execute the various branches. Figure 4 shows the relationship between the three user verbs and the various system runtime verbs which they compile.

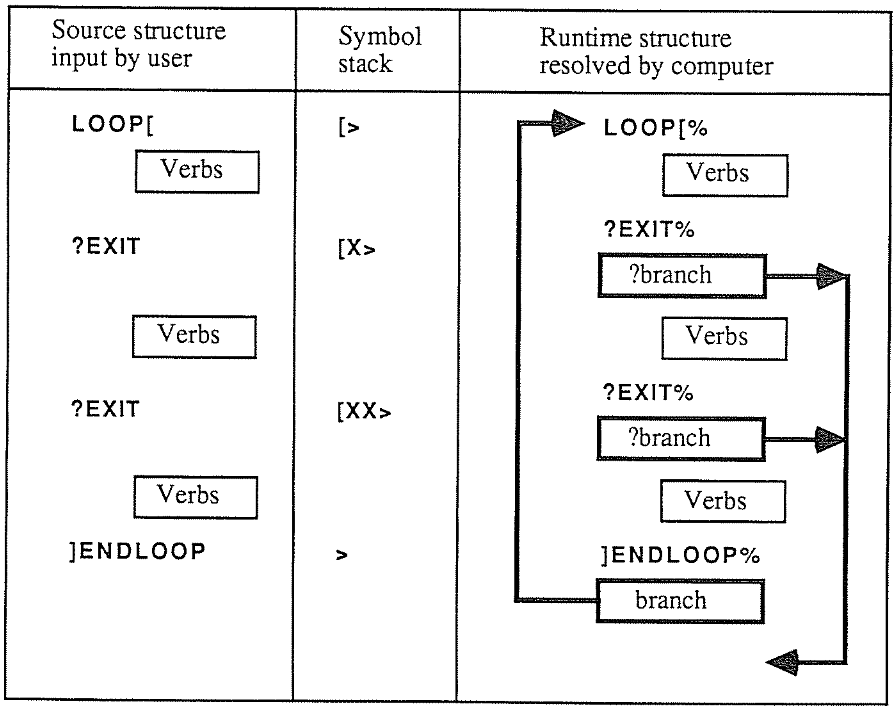

Figure 4. Conditional Loop Structure

The coding of the five threaded secondary verbs is included in the Appendix for reference. In the following we present the Outer Interpreter verb DO-LINE as a good example of the usage of the conditional loop and branch structures.

```
'DO-LINE  DO:
    LOOP[
         SKIP-BLANKS
         EOL?    \    End-Of-Line ?
    ?EXIT
         NAME?    \    prefixed quote?
         ?IFTRUE
              DO-NAME
         ?ELSE
              BLANK  WORD^Ø   SCAN     \     lexical scan to the
                                       \     Word buffer
              WORD^Ø  VERB?            \     is it a predefined
                                       \     verb?
              ?IFTRUE
                   DO-VERB
              ?ELSEIF
              WORD^Ø  NUMBER?          \     a number maybe?
              ?IFTRUE
                   DO-NUMBER
              ?ELSE
                   WORD^Ø    UNKNOWN
                   RUN    \  ignore it and back to the Outer
                          \  Interpreter
              ?ENDIF
           ?ENDIF
     ]ENDLOOP
:END
```

This definition uses many Outer Interpreter system verbs which are not defined in this paper, however they should be intuitively obvious within the context of interpreting an input line. Thus we find that the conditional structures enable development of extremely readable verbs. All of the system verbs have headers and are accessible; and when the student becomes interested in what makes the whole system come together then she will find that it was developed using the same pedagogic principles that it teaches.

The defining verb pair – DO: :END

The defining verb set used in Forth, including : ; <BUILDS DOES> CONSTANT VARIABLE, can be extremely confusing because of the strange syntax and context dependency of the verbs (20). The concept of improving the defining verb syntax by means of 'deferred nested definitions' was proposed by Frank and Johnson (21) in which two new building words could be nested within the colon definition to replace <BUILDS and DOES>. Bergmann (11) subsequently showed that one could extend the scope of the basic defining verb pair : ; to include deferred nested definitions, and thus limit the entire range of defining verbs to : and ; alone. This approach was used as the basis for the DO: :END defining verb pair in REPTIL and all other required defining verbs and data structures are derived from suitable nesting of DO: :END sets.

Consider for example the definition and use of a defining verb CONSTANT:

```
<16>      'CONSTANT DO:    \     value 'name |P
1<16:>       DO:
2<16::>         42  \    the answer to life, the universe, and
2<16::>             \    everything...
2<16::>       :END
1<16:>       ARGPATCH     \     the 42 with the value on TOP
1<16:>    :END
<16>
<16>        5   'WEEKDAYS   CONSTANT
<16>
<16>  WEEKDAYS    =
5
<16>_
```

When CONSTANT is invoked, the internal DO: :END pair converts the name (i.e. WEEKDAYS in the example shown) to a verb, and assigns it a value 42. The verb ARGPATCH then patches the first argument of the newly created verb by replacing it with the value at TOP (Top Of P-stack).

The nesting of DO: :END gives a more consistent, flexible and readable form than that of the Forth equivalent <BUILDS DOES>. Nesting can be done to any required level and the concept of 'deferred definition' is easier to understand.

There are 6 system verbs making up the defining verb set, being DO:, DO:%, DO:MOVE%, CREATE, :END and :END%, as follows:

```
'DO:    DO:    NOW      \      'name
     COMPILE?
     ?IFFALSE
          CREATE
     ?ELSE
          COMPILE:    DO:%
     ?ENDIF
     EOC     \     addr  |P  (start of definition)
     Ø   INCLUDE      \      space for status word
     3A    >S      \       ':' to Symbol-stack
 :END
```

Thus the DO: verb will only CREATE a verb if it is in the outer nesting shell (not in the compile mode), otherwise it will defer creation to when DO:% is invoked.

```
'DO:%  DO:    #VERB    RIGHT>
     CREATE
     DO:MOVE%
 :END
```

The verb DO:MOVE% increments the interpretive pointer past the inline deferred definition, and moves the entire definition to EOC. Note that REPTIL threaded secondaries can only access the interpretive pointer indirectly through the R-stack, which holds the address of the following inline token to be popped to the interpretive pointer by :END%.

```
'DO:MOVE%   DO:    \    invoked only by DO:%
     R>   DROP    \    drop the return to :END% in DO:%
     R    \    source |P (start address of deferred definition)
     EOC    \    source, dest |P
     OVER  1  +  CFETCH     \     source, dest, length |P
     2   *    \    source, dest, #bytes |P
     DUP  R>  +  >R     \     increment R-stack by #bytes
     DUP   'EOC   <+    \    increment EOC by #bytes
     CMOVE     \     move #bytes from source to dest
 :END
```

The verb CREATE does not do any creation as such, but only takes the name which exists in limbo at EON, and gives it the permanence of a verb. This is done by incrementing EOT (End Of Tokens), associating EON and EOC (End Of Code) with EOT, and incrementing EON past the name, as follows (refer Figure 1):

```
'CREATE  DO:    \    'name
     -1   =?    \    check for no existing verb of same name
     ?IFFALSE    \    output a warning message
          EON   $=
          SPACE   "  ALREADY EXISTS"   $=
          NULINE
          COL-RESET
     ?ENDIF
     2  'EOT  <+    \    increment EOT
     EON
     NAMES^   EOT   +
     STORE     \     include EON in the names pointers
     EON  CFETCH  1  +     \     #bytes |P (length of name)
     'EON  <+    \    increment EON past the current name
     EOC
     CODES^   EOT   +
     STORE     \     include EOC in the codes pointers
 :END
```

Once the verb has been opened with the DO: verb, then any further verbs are simply included at EOC, ultimately being closed with the :END verb. The :END verb compiles the verb :END% (the last verb in every definition), and pulls it down a level of nesting by popping a ':' symbol from the S-stack. The length of the verb is evaluated and saved in the status word (high byte). Once the S-stack is empty, then compilation is complete and the system returns to the interpretive mode. Note that the :END verb can only be used to correctly close a DO: verb nest, and a careful check is made of the symbol on TOS (Top Of S-stack).

```
':END   DO:   NOW      \     addrØ |P (start of definition)
     S  3A   =?     \     ':' ?
     ?IFTRUE     \     correct nesting syntax
          S>   DROP     \     reduce one level of nesting
          COMPILE:  :END%
          EOC     \     addrØ, addr1 |P (end of definition)
          OVER    -    /2     \    addrØ,  #words |P
          SWAP 1 +  \  #words, addrØ+1 |P (addr of length byte)
          CSTORE
          COMPILE?
          ?IFFALSE
               FALSE   REDO?    <-    \  out of redo (edit) mode
          ?ENDIF
     ?ELSE    \  incorrect nesting - warning message and exit
          "  NEEDS  DO:"   $=
          NULINE
          COL-RESET
     ?ENDIF
:END
```

The verb `:END%` is always coded as a primitive and links the verb with the inner interpreter. It is described in the section on the inner interpreter below.

Execution `GO`, *the inner interpreter* `NEXT` *and the threaded return* `END%`

The inner interpreter for a token threaded language is similar to that of an indirect threaded language, the distinction being that a program consists of a list of tokens rather than a list of addresses. The REPTIL inner interpreter is adapted from that of RTL (9) and uses the sign bit of the status word (which heads every verb) to determine if that verb is a primitive or a threaded secondary. The inner interpreter `NEXT` is always coded as a headerless primitive; however it is convenient to present it in terms of REPTIL algorithms in order to describe its function. We first describe the fundamental verb `GO` which executes a verb token given on the P-stack. The verb `NEXT` then simply fetches a token from the address in the interpretive pointer before invoking `GO`. In practice the primitive code for `GO` is simply repeated in the verb `NEXT`. In the following `I^` represents the interpretive pointer, and `DO-IT` represents a routine to jump to and execute primitive code.

```
'GO  DO:     \    token |P (coded as a primitive)
     CODE^   \  code^ |P (the address pointing to the verb code)
     DUP  CFETCH    \   code^, status |P
     <Ø?    \    status sign bit check
     ?IFTRUE    \    a primitive
          2  +    \  code^+2 |P (increment past the status word)
          DO-IT    \    execute primitive code
     ?ELSE    \    a threaded secondary
          I^   >R      \      save return address on R-stack
          2  +      \     code^+2 |P (the next token address)
          'I^  <-
          NEXT      \     threaded call
     ?ENDIF
:END
```

```
'NEXT  DO:     \    coded as a headerless  primitive
   I^  \  addr |P (of current token)
   FETCH  \  token |P
   2  'I^  <+  \   prepare I^ for the next token address
   GO   \  execute the current token
:END
```

Every primitive ends with a jump to **NEXT**, and every threaded secondary ends with the verb **:END%**. The verb **:END%** is the runtime verb of **:END**, and it too is always coded as a primitive. Its equivalent REPTIL algorithmic form follows:

```
':END%   DO:      \      threaded return - coded as a primitive
    R>   'I^   <-      \      pop address of next token to I^
    NEXT
:END
```

Discussion

Since its inception in 1984, REPTIL has undergone a number of incarnations (1,2,3). It is still being conceptually developed, and in its current implementation is lacking in many important features such as input/output routines for interfacing with external equipment. Nevertheless it is gradually evolving into a unique language in its own right which in some ways is more fundamental than its predecessor Forth. The major departures of REPTIL from Forth include the prefixed quote notation, the user interface (including the informative prompt, decompiler and editor) and the structured constructs. This paper has mainly concentrated on the evolution and description of the four fundamental structured constructs of REPTIL, being deferred execution, conditional branch, conditional loop, and deferred definition. Structure is a rather nebulous concept and in the programming language context can be considered as augmenting the linear description of a language with a new dimension, and thus further enhancing the human visual thinking process (22). Thus structure is not only the structured constructs, but includes the visual form that accompanies them. In Forth the visual form is largely lost by the constraint of suitably filling the virtual screen blocks– "If it cannot fit into three screens then its not worth programming." Furthermore one can easily define Forth words with improper structured constructs which will hopelessly crash the system. REPTIL has an automatic unconstrained structured format coupled with a non-fatal intuitive syntax checking that ensures correctly nested structures. All of the REPTIL structures place the system in the compile mode, allowing them to be invoked and experimented with outside of a definition.

The current implementation of REPTIL is on the Apple II computer, mainly since it still dominates the schools. This implementation includes about 215 verbs, 47 of which are primitives. Plans for the future development of REPTIL include the following:

1. Enhance the Apple II version with a basic 'Reptile' graphics facility including a plotter interface in order to promote its acceptance in the elementary school environment.
2. Convert and port REPTIL to a 68000 based computer so as to stabilize a standard fundamental verb set and promote its acceptance in the application environment.
3. Attempt to justify the claims that REPTIL can bridge the gap between education and application by means of controlled experiments.

Acknowledgements

The author wishes to thank the editors and referees for their constructive comments which have helped to improve this paper considerably.

References

[1] I. Urieli, 'HELLO, A REPTIL I AM', *Proceedings of the 1984 Rochester Forth Conference,* pp. 236-243.

[2] I. Urieli, 'REvised REcursive AND? 'REPTIL :IS', *Proceedings of the 1985 Rochester Forth Conference,* pp. 229-231.

[3] I. Urieli, 'REPTIL—promoting dialog between humanoid and computer', *Proceedings of the 1986 Rochester Forth Conference,* pp. 229-232.

[4] J. C. Bender, 'Forth implementation in a high-level language', *Proceedings of the 1985 Rochester Forth Conference,* pp. 85-87.

[5] K. M. Chung, H. Yuen, 'A "Tiny" Pascal compiler', *BYTE,* Sept., Oct., Nov. 1978.

[6] J. R. Allen, 'Computing, LISP and You', *Microcomputing*, Feb. 1982, pp. 28-42.

[7] M. Buber, *Between Man and Man,* MacMillan Publishing Co., 1965.

[8] G. Orwell, *Nineteen Eighty Four,* Harcourt Brace, New York, 1949.

[9] R. Buege, 'Status Threaded Code', *Proceedings of the 1984 Rochester Forth Conference,* pp. 103-104.

[10] R. Buege, 'A decompiler design', *Proceedings of the 1984 FORML Conference.*

[11] E. E. Bergmann, 'PISTOL—a Forth-like Portably Implemented Stack Oriented Language', *Dr. Dobb's Journal,* Number 76, Feb. 1983, pp. 12-15.

[12] J. M. Sachs, S. K. Burns, 'STOIC, an interactive programming system for dedicated computing', *Software—Practice and Experience,* Vol. 13, 1983, pp. 1-16.

[13] E. L. Solley, 'SPHERE: an in-circuit development system with a Forth heritage', *Proceedings of the 1984 Rochester Forth Conference,* pp. 25-31.

[14] P. Bartholdi, 'The "TO" solution', *Forth Dimensions,* Vol. 1, No. 5, Jan. 1979.

[15] E. Rosen, 'QUAN and VECT—High speed, low memory consumption structures', Proceedings, *Fourth FORML Conference,* October 1982.

[16] H. Glass, 'Towards a more writeable Forth syntax', *Proceedings of the 1983 Rochester Forth Conference.*

[17] E. E. Bergmann, 'Languages and Parentheses—a suggestion for Forth-like languages', *Dr. Dobb's Journal,* July 1984, pp. 102-108.

[18] A. F. T. Winfield, *The Complete Forth*, Sigma/Wiley NY, 1983, p. 33.

[19] E. Soloway, J. Bonar, K. Ehrlich, 'Cognitive Strategies and Looping Constructs: An Empirical Study', *Communications of the ACM*, Vol. 26, No. 11, November, 1983, pp. 853-860.

[20] W. F. Ragsdale, 'A new syntax for defining Defining Words', *Forth Dimensions,* Vol. II, No. 5, pp. 121-128.

[21] D. C. Frank, G. W. Johnson, 'Generalized Building Words', *Proceedings of the 1982 Rochester Forth Conference,* pp. 207-210.

[22] P. M. van Hiele, *Structure and Insight—A Theory of Mathematics Education,* Academic Press, 1986.

Israel Urieli obtained a M.Sc. in Electrical Engineering from the Technion, Israel, and a Ph.D. in Mechanical Engineering at the Witwatersrand University in South Africa. He spent about 20 years in Israeli industry doing various research and development projects, ranging from Radar to Stirling Engines. He recently joined Ohio University and is responsible for developing and teaching the core programming courses (Pascal and FORTRAN, of course) to engineering students.

Manuscript received January 1986.

Appendix

There are eleven verbs which make up the conditional branch structure. These include the verbs `?IFTRUE`, `?IFFALSE`, `?ELSE`, `?ELSEIF`, `?ENDIF`, `?END%`, `?ENDIF%`, `?IFTRUE%`, `?IFFALSE%`, `?ELSE%` and `?ELSEIF%`. The first seven verbs are threaded secondaries and are coded as follows:

```
'?IFTRUE  DO:   NOW
     COMPILE?
     ?IFFALSE
          EOC  'EOC-SAVE  <-
     ?ENDIF
     3F  >S    \    '?' to S-stack
     COMPILE:  ?IFTRUE%
     EOC    \    addr1 |P
     0  INCLUDE    \    space at addr1 for #words to branch
 :END

'?IFFALSE   DO:   NOW
     COMPILE?
     ?IFFALSE
          EOC   'EOC-SAVE   <-
     ?ENDIF
     3F  >S
     COMPILE:  ?IFFALSE%
     EOC    \    addr1 |P
     0  INCLUDE
 :END
```

Note the similarity between the `?IFTRUE` and `?IFFALSE` verbs. Their difference in behaviour is due only to their respective runtime verbs `?IFTRUE%` and `?IFFALSE%`.

```
'?ELSE  DO:   NOW    \    addr1 |P
     S  3F   =?   \    '?' on S-stack? (Nesting syntax check)
     ?IFTRUE
          45  >S    \  'E' to S-stack
          COMPILE:   ?ELSE%
          >R   \    addr1 |R
          EOC   \    addr2  |P     addr1  |R
          Ø  INCLUDE    \  space at addr2 for #words to branch
          EOC  R - /2 addr2, (addr3-addr1)/2 |P  addr1 |R
          R>   \   addr2, (addr3-addr1)/2, addr1  |P
          STORE   \  addr2  |P  (store #words to branch in addr1)
     ?ELSE
          " NEEDS ?IFTRUE OR ?IFFALSE" $=
          NULINE
          COL-RESET
     ?ENDIF
  :END
```

```
'?ELSEIF   DO:    NOW      \      addr1 |P
     S  3F  =?     \     '?' on S-stack?
     ?IFTRUE
         46  >S     \  'F' to S-stack
         COMPILE:   ?ELSE%
         >R   \    addr1 |R
         EOC    \    addr2  |P     addr1   |R
         Ø  INCLUDE    \  space at addr2 for #words to branch
         EOC  R - /2 addr2, (addr3-addr1)/2 |P  addr1 |R
         R>    \   addr2, (addr3-addr1)/2, addr1   |P
         STORE    \  addr2  |P  (store #words to branch in addr1)
     ?ELSE
         " NEEDS ?IFTRUE OR ?IFFALSE" $=
         NULINE
         COL-RESET
     ?ENDIF
   :END
```

Note that the verbs ?ELSE and ?ELSEIF are almost identical. Their runtime behaviour is also identical. Their only difference is in the syntax symbol which is resolved by ?ENDIF.

```
'?ENDIF   DO:   NOW      \      addr1,addr2,...,addrn |P
     S  45    =?     \     'E'?  (is there an ?ELSE clause?)
     ?IFTRUE
          S>    DROP
     ?ENDIF
     S   3F   =?     \     '?'?
     ?IFTRUE
          ?END%    \     resolve all branch addresses
          COMPILE:  ?ENDIF%
          COMPILE?
          ?IFFALSE
               GO-EOC
          ?ENDIF
     ?ELSE
          " NEEDS ?IFTRUE, ?IFFALSE OR ?ELSE"  $=
          NULINE
          COL-RESET
     ?ENDIF
  :END
```

The verb ?END% resolves all the branch addresses of the various clauses. Each ?ELSEIF clause causes a 'F?' pair of symbols to be pushed to the S-stack, which is popped as each address is resolved. Since ?END% does not know how many clauses to resolve, it uses the conditional loop structure with exit conditions determined by the S-stack.

```
'END%  DO:    \    addr1, addr2, ..., addri |P
     LOOP[    \   until all branch addresses are resolved
          S  3F  =?  NOT     \    exit if not a '?'
     ?EXIT
          S>  DROP    \    pop the '?'
          >R     \    addri |P
          EOC   R   -   /2     \    (addrj-addri)/2 |P
          R>      \    (addrj-addri)/2, addri |P
          STORE      \     #words to branch in addri
          S  46  =?  NOT     \    exit if not a 'F'
     ?EXIT
          S>  DROP     \    pop the 'F'
     ]ENDLOOP
 :END

'?ENDIF%    DO:   %VERB   <LEFT      \     mainly for decompilation
   :END
```

There are seven verbs which make up the conditional loop structure including `LOOP[`, `?EXIT`, `]ENDLOOP`, `EXITS%`, `LOOP[%`, `?EXIT%` and `]ENDLOOP%`. The first five verbs are threaded secondaries and are coded as follows:

```
'LOOP[  DO:  NOW
     COMPILE?
     ?IFFALSE
          EOC  'EOC-SAVE  <-
     ?ENDIF
     5B  >S    \   '[' to S-stack
     EOC     \   addr1 |P (to evaluate #words to branch)
     COMPILE:  LOOP[%
 :END

'LOOP[%  DO:  %VERB  RIGHT>     \    for meaningful decompilation
 :END

'?EXIT DO: NOW
    S  5B  =?      \  '['?
    S  58  =?  OR   \  or 'X'? (nesting syntax check)
    ?IFTRUE
         58  >S     \  'X' to S-stack
         COMPILE:  ?EXIT%
         EOC      \   addr1 |P
         0  INCLUDE   \   space for #words to branch
    ?ELSE   \   incorrect nesting
         " NEEDS LOOP[ OR ?EXIT" $=
         NULINE
         COL-RESET
    ?ENDIF
 :END
```

```
'JENDLOOP  DO:  NOW     \   addr1, addr2, ..., addri |P
          EXITS%   \   resolve exits only (addr2, ...,addri)
          S  5B  =?    \   '['? (nesting syntax check)
          ?IFTRUE
               S>  DROP  \  drop a level of nesting
               COMPILE:  ]ENDLOOP%  \  unconditional branch to LOOP[%
               EOC  -  /2   \   (addr1-addrj)/2 |P (#words to branch back)
               INCLUDE
               COMPILE?
               ?IFFALSE    \   in execution mode?...
                    GO-EOC    \   ...then execute loop structure
               ?ENDIF
          ?ELSE   \   nesting error
               " NEEDS LOOP[" $=
               NULINE
               COL-RESET
          ?ENDIF
      :END

'EXITS%  DO:   \   addr2, addr3, ..., addri |P (resolve exit addresses)
     LOOP[   \   until all the exit addresses have been resolved
          S  58  =?  NOT   \   not an 'X'?...then exit
     ?EXIT
          S>  DROP   \   drop the 'X'
          >R   \   addri |R
          EOC  R  -  /2   \   (addrj-addri)/2 |P,   addri |R
          2  +
          R>  \  2+(addrj-addri)/2, addri |P
          STORE
     ]ENDLOOP
 :END
```

Glossary

The following glossary includes most of the verbs referred to in this paper, as well as some other pertinent verbs. The various system verbs which make up the Outer Interpreter (RUN), decompiler (UNDO:) and editor (REDO:) are not included.

The following stack notation is used:

|P denotes the top of the Parameter stack (P-stack).
|R denotes the top of the Return stack (R-stack).
|S denotes the top of the Symbol stack (S-stack).
n refers to a signed 16 bit number on a stack.
u,addr refer to an unsigned 16 bit number or address on a stack.
b refers to an unsigned byte on a stack.
T/F refers to a logical (TRUE or FALSE) value on a stack (FALSE = 0).
d refers to a signed 32 bit number on a stack (d_{hi} is the most significant 16 bits, and d_{lo} the least significant 16 bits).
ud refers to an unsigned 32 bit number on a stack.
token refers to a token (even unsigned number, 0...EOT) on a stack.

'name (prefixed quote) do a lexical scan of the name (terminated by a blank) to EON. The following action is governed by the state table as follows:

		COMPILE? \ in the compile mode?	
		TRUE	FALSE
VERB?	TRUE	1 Include the token literal handler followed by the verb token at **EOC**	2 Push the verb token to the P-stack
	FALSE	3 Include the name string literal handler followed by the name at **EOC**	4 Candidate for a verb. Push -1 to the P-stack

EOT \|P → token \|P (End of Tokens—most current token defined)

Parameter stack (P-stack) verbs:

DUP	n |P → n n |P
DROP	n |P → |P
SWAP	n1 n2 |P → n2 n1 |P
OVER	n1 n2 |P → n1 n2 n1 |P
PICK	b |P → n |P (pick the b'th element from the P-stack. Thus DUP ≡ 1 PICK, OVER ≡ 2 PICK)
P^RESET	n1 ... ni |P → |P (reset the P-stack pointer)
P^	|P → addr |P (push the P-stack pointer)
P^0	|P → addr |P (push the Bottom Of P-stack (BOP) pointer)

Return stack (R-stack) verbs:

`>R`	n |P |R → |P n |R
`R`	|P n |R → n |P n |R
`R>`	|P n |R → n |P |R
`R^RESET`	n1 ... ni |R → |R (reset the R-stack pointer)

Note: Even though the R-stack can be used as a very convenient temporary holding stack, this must be done with caution, since the main purpose of the R-stack is to hold the return address of the next in-line token to be invoked. Thus if the current token is that of a threaded verb, then the next in-line token address is pushed to the R-stack by `NEXT`, and subsequently popped by `:END%`.

`:END%`	runtime verb ending every threaded verb, removes address of following in-line token from the R-stack and stores it in the 'Interpretive Pointer' before invoking the primitive `NEXT`.

Memory storage and retrieval verbs:

`STORE`	n addr |P → |P (store n at addr)
`+STORE`	n addr |P → |P (add n to content at addr)
`CSTORE`	b addr |P → |P (store b at addr)
`+CSTORE`	b addr |P → |P (add b to content at addr)
`FETCH`	addr |P → n |P (fetch n from addr)
`CFETCH`	addr |P → b |P (fetch b from addr)

Arithmetic verbs:

`*`	n1 n2 |P → (n1*n2) |P (product)
`+`	n1 n2 |P → (n1+n2) |P (sum)
`-`	n1 n2 |P → (n1−n2) |P (difference)
`/`	n1 n2 |P → (n1/n2) |P (quotient)
`/2`	n |P → n/2 |P (signed divide by two)
`MOD`	n1 n2 |P → (n1 **mod** n2) |P (unsigned remainder)
`*/`	n1 n2 n3 |P → (n1*n2/n3) |P (scaling operator—32 bit intermediate product)

Relational verbs:

<0? n |P → T/F |P True if n is less than zero.

<=? n1 n2 |P → T/F |P True if n1 is less than or equal to n2.

=? n1 n2 |P → T/F |P True if n1 is equal to n2.

Deferred Execution structure:

(|S → '(' |S (open deferred execution structure)

) '(' |S → |S (close deferred execution structure)

, a no-op.

Conditional Branch structure:

?IFTRUE T/F |P |S → T/F addr |P '?' |S (open conditional branch structure. When structure is subsequently executed (S-stack empty), if T/F condition is FALSE then code will branch to pointer in addr, otherwise will continue with following code.

?IFFALSE T/F |P |S → T/F addr |P '?' |S (open conditional branch structure. When structure is subsequently executed (S-stack empty), if T/F condition is TRUE then code will branch to pointer in addr, otherwise will continue with following code.

?ELSE addr1 |P '?' |S → addr2 |P '?E' |S (continue conditional branch structure—resolve branch pointer in addr1, reserve addr2 for subsequent unconditional branch pointer to be resolved by ?ENDIF)

?ELSEIF addr1 |P '?' |S → addr2 |P '?F' |S (continue conditional branch structure—resolve branch pointer in addr1, reserve addr2 for subsequent unconditional branch pointer to be resolved by ?ENDIF)

?ENDIF addr1 ... |P '?...' |S → |P |S (close conditional branch structure—resolve all branch pointers (addr1 ...))

Conditional Loop structure:

LOOP[|P |S → addr |P '[' |S (open conditional loop structure—reserve addr for subsequent unconditional branch pointer to be resolved by]ENDLOOP)

?EXIT addr1 |P '[' |S → addr1 addr2 |P '[X' |S (continue conditional loop structure—reserve addr2 for subsequent conditional branch pointer to be resolved by]ENDLOOP)

`]ENDLOOP`	addr1 ... |P '[...' |S → |P |S (close conditional loop structure—resolve all branch pointers (addr1 ...))

Deferred Definition structure and related verbs:

`DO:`	'name |P |S → addr |P ':' |S (open deferred definition)
`:END`	addr |P ':' |S → |P |S (close deferred definition—resolve length of definition (number of tokens) and store at addr)
`:HAS`	'name |P → |P (define a value cell without assigning it a value)
`:IS`	n 'name |P → |P (define a constant having a value n)
`UNDO:`	token |P → |P (decompile the verb of token)
`REDO:`	token |P → |P (redo (edit) the verb of token)
`ID=`	token |P → |P (display the name of token)
`LIST16`	token |P → |P (display the names of the previous 16 tokens)
`LIST`	output list of verbs and their associated tokens starting from `EOT`
`NAME^`	token |P → addr |P (address of token name)
`NAMES`	|P → addr |P (address of start of names block)
`NAMES^`	|P → addr |P (address of start of names pointers table)
`EON`	|P → addr |P (address of next available space in names block)
`CODE^`	token |P → addr |P (address of token code)
`CODES`	|P → addr |P (address of start of codes block)
`CODES^`	|P → addr |P (address of start of codes pointers table)
`EOC`	|P → addr |P (address of next available space in codes block)
`CELLS`	|P → addr |P (address of start of value cells block)
`CELL`	token |P → addr |P (cell address of token, only if verified to be that of a value cell)
`<+`	n token |P → |P add n to the value in the cell of token.
`<-`	n token |P → |P assign value n to the cell of token.

Strings:

`"`	If in the compile mode then compile the string literal handler `$%` followed by the following in-line string (terminated by another `"` or a Return) at `EOC`, else lexically scan the string to `EON` and push `EON` to the P-stack.
`\`	If in the compile mode then compile the comment literal handler `\%` followed by the following in-line string (terminated by another `\` or a Return) at `EOC` else lexically scan the string to `EON`.
`$=`	addr |P → |P (display the string located at addr)

Input/Output verbs:

`=`	n |P → |P	pop-and-display the value at TOP.
`U=`	u |P → |P	pop-and-display the unsigned value at TOP.
`CGET`	|P → b |P	get a character from the keyboard.
`CPUT`	b |P → |P	pop-and-display the character at TOP