Stack Usage and Parameter Passing

Siem Korteweg

J. V. Stolberglaan 16
3931 KA Woudenberg
the Netherlands

Hans Nieuwenhuÿzen

Sterrewacht "Sonnenborgh"
Zonneburg 2
3512 NL Utrecht
the Netherlands

Abstract

Routines can use the arithmetic stack for parameter passing and for temporary storage. This paper describes a mechanism that formalizes stack usage by providing each routine with an "activation record"-like data structure, cf. Pascal. This data structure, consisting of a storage space and an arithmetic stack, provides an interface between routines and the arithmetic stack. The interface allows the flow of all possible data to and from routines, and offers extreme clarity of source code. The described implementation increases the execution time and the amount of generated object code by 10–30%.

Introduction

The Forth arithmetic stack is used for parameter passing and temporary storage. We can describe these functions in the following way:

— The "outside world" places input values to be used by a routine on the stack.
— The routine may use temporary scratch values on the stack.
— The routine performs arithmetic operations on the stack using the input and scratch values.
— The routine leaves output values on the stack to be used by the "outside world".

P. Bartholdi [BART 81] and R. Joosten [JOOS 82] developed and implemented a mechanism to address locations on the stack by means of specially formatted identifiers. This mechanism has proved its usefulness and it offers extra clarity compared with parameter-less Forth, but the definitions still have to deal with the stack structure in a very explicit way. In the following paragraphs we will describe a mechanism that offers the possibility to make Forth code independent of the arithmetic stack. Following most literature (see also [McKI 81]) we will call the input and scratch values "parameters".

Formalized Stack Usage and Parameter Passing

Storage spaces and stacks:

We associate with every colon definition the following structures:

— A storage space to contain the input and scratch values.
— A stack to communicate with other definitions and to do some "private" local arithmetic.

Whenever a routine is called by another routine, its input values are transferred from the stack of the calling routine to its storage space. Whenever a routine has finished its execution, its output values are transferred from its storage space to the stack of the calling routine.

Contents of the storage space:

Every location in the storage space has its own identifier and the location and identifier simulate the Fys-Forth datastructure VALUE [JOOS 83]. The correspondence between identifiers and locations in the storage space is made by means of "declarations" when the Forth code is compiled. The following example shows what these declarations look like:

```
: EXAMPLE
   INPUT: INPUT1,INPUT2
   LOCAL: LOCAL1,LOCAL2,LOCAL3
   ..... ;          ( use the INPUT's and the LOCAL's )
```

These declarations have the following meaning: EXAMPLE's storage space contains 5 numbers. Two input values are denoted by INPUT2, the topmost number on the stack of the calling definition, and by INPUT1, the second number on the stack. The storage space also contains 3 scratch values, denoted by LOCAL1, LOCAL2, and LOCAL3.

The declaration mechanism should meet the following demands:

— The number of parameters for one routine has an implementation dependent maximum.
— INPUTi and LOCALi are arbitrary names except that they cannot contain the character used to separate the identifiers, a comma.
— When a list of identifiers does not fit on one line, a convention is used for the additional identifiers: extra, consecutive occurrences of INPUT: and LOCAL:.
— When any of the identifiers is already in use for another routine, the parameter will act as a redefinition. The original routine cannot be used within the routine that is compiled; it "reappears" after the compilation.
— Parameters of a routine can be used as input values for any routine that uses parameters.
— The declarations should appear in a particular order; first the input values and then the scratch values, or vice versa.
— The INPUT:- and LOCAL:-statements should be the first executed statements of a routine.
— The scratch values have an initial value of zero.

Output values:

When EXAMPLE finishes execution, it might want to leave the contents of some of its parameters on the stack of the calling routine. This means that these values have to be copied from EXAMPLE's storage space to the stack of the calling definition.

```
: EXAMPLE
   INPUT: INPUT1,INPUT2
   LOCAL: LOCAL1,LOCAL2,LOCAL3
   .....          ( use the INPUT's and the LOCAL's )
   OUTPUT: OUT1,OUT2,OUT3,OUT4 ;
```

The output mechanism should meet the following demands:

— The number of output values is limited by the size of the stack of the calling routine.
— When the list of identifiers does not fit on one line, a convention is used for the additional identifiers: extra, consecutive occurrences of OUTPUT: .
— Any parameter can occur any number of times in the list of identifiers, as in the following example:

```
: ?DUP
  INPUT: VAL.ON.STACK
  VAL.ON.STACK
  IF   OUTPUT: VAL.ON.STACK,VAL.ON.STACK
       ( leave number twice on the stack )
  ELSE OUTPUT: VAL.ON.STACK
       ( leave number once on the stack )
  THEN ;
```

— A routine may only "output" its own parameters; that is, identifiers occurring in the list following OUTPUT: also have to occur in a list following INPUT: or LOCAL: .
— No occurrence of OUTPUT: means that the called routine does not want to leave output values on the stack of the calling routine. This implies that there are no other ways to leave output values on the stack of the calling routine than through the use of OUTPUT:
— The OUTPUT:-statement should be the last executed statement of a routine.

Stack notation:

We can use the lists of identifiers following INPUT: and OUTPUT: to describe the net effects on the stack of the calling routine caused by the execution of the called routine. The net effects of the routine EXAMPLE can be described as:

(INPUT1,INPUT2 -> OUT1,OUT2,OUT3,OUT4)

Summary:

Assume that the routine CALLER uses EXAMPLE. The following diagram shows the stacks and storage spaces of CALLER and EXAMPLE when EXAMPLE is executing:

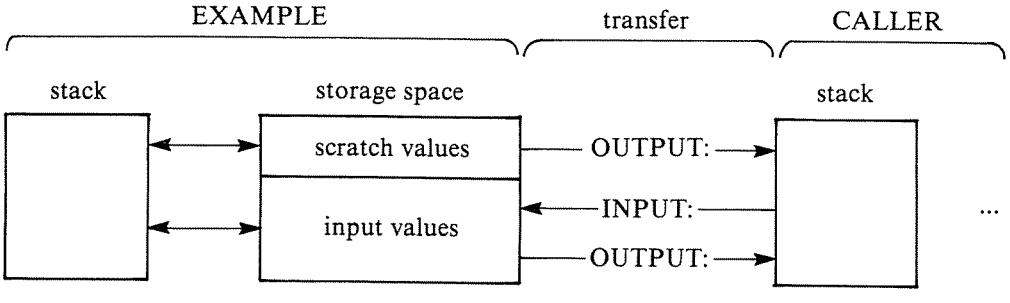


Figure 1.

For the sake of clarity, we have used OUTPUT: separately for the transfer of input and scratch values, but they can be left on CALLER's stack using one occurrence of OUTPUT:. Note that INPUT: and OUTPUT: transfer their values only once. INPUT: transfers its values when the execution of EXAMPLE starts, and OUTPUT: transfers its values just before the execution of EXAMPLE stops. The transfer of values between EXAMPLE's storage space and stack can happen any number of times during the execution of EXAMPLE.

Implementation

The Forth code for the high level implementation is given in Appendix 1. In this section we will describe our strategy for the implementation. We will use the following routines as examples:

```
: EXAMPLE
    INPUT: INPUT1,...,INPUTa
    LOCAL: LOCAL1,...,LOCALb
    ...
    OUTPUT: OUT1,...,OUTc ;

: CALLER
    ...        ( place EXAMPLE's input values on stack )
    EXAMPLE
    ... ;                ( process EXAMPLE's output values )
```

Allocation of storage spaces:

We could give every routine its storage space when it is compiled, but that would be a poor strategy as only those routines that have not yet finished their execution use their storage space. Therefore, we allocate a storage space only when the corresponding routine is used. This means that we do not know the location of this storage space when the routine is compiled and we do not know the runtime addresses of the parameters at compile time. We only know their relative addresses in the storage space given by the order of the identifiers in the declarations. When the routine is executed and a parameter is used, we have to add its relative address within the storage space to the base address of the storage space to obtain its runtime address.

The addresses on the return stack correspond to routines that have not yet finished their execution. We have to "remember" all their base addresses to continue with their execution later. A Sentry (Stack entry) stack will be used for this purpose. At execution time the calculation of a runtime address of a parameter consists of adding the relative address of the parameter to the topmost base address on the Sentry stack.

Allocation of stacks:

Analogously to the allocation of the storage spaces, we have to allocate the stacks when the corresponding routine is used. We can extend the Sentry stack to contain the bottoms of the stacks currently used (the upper bound of all the stacks are identical in our implementation). The calculation of the runtime addresses of parameters and the checks against stack underflow have to use the topmost pair of the base addresses and bottoms of stacks.

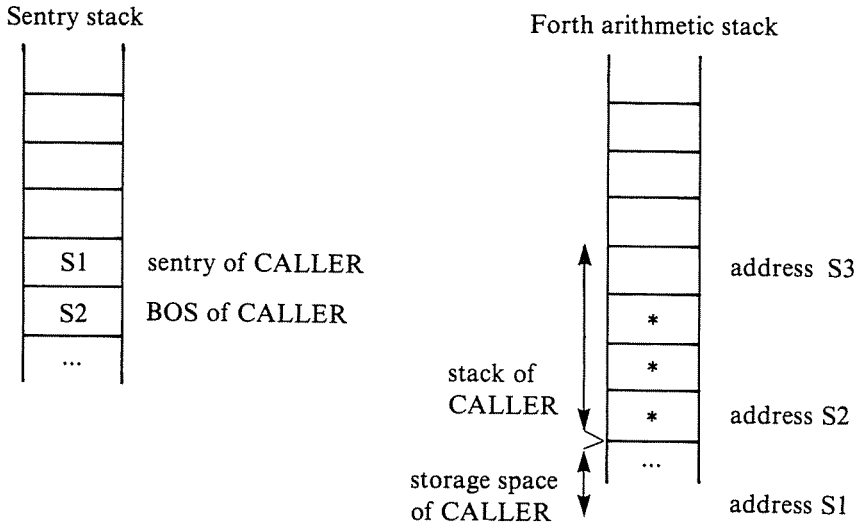
A possible implementation:

Programmers might doubt the practical use of the concepts described as they require extra space in memory and copy operations for the transfer of values from stacks to storage spaces or vice versa. However, having made a formal description of the concepts, in implementing them we follow a more pragmatic approach. We simulate all the storage spaces and stacks on the Forth arithmetic stack.

To save memory and copy operations, the storage space of EXAMPLE partly coincides with the stack of CALLER. The input values of EXAMPLE occupy the same locations in memory as the numbers that CALLER puts on its stack for EXAMPLE. The stack of EXAMPLE is placed on top of its storage space. Note that this is a possible implementation; our formalization does not force us to place the storage spaces on the Forth arithmetic stack. We do this merely for efficiency, but it is possible to place them somewhere else, and in systems with a small arithmetic stack this may actually be done.

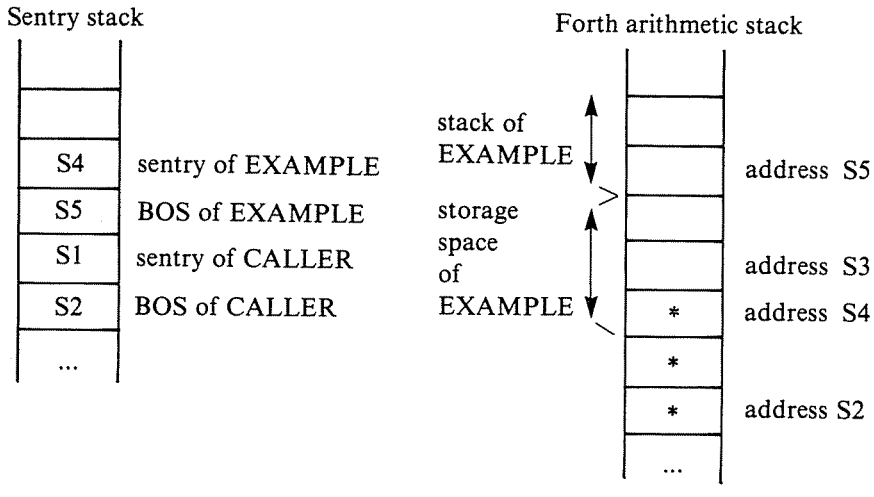
The following diagrams show the course of events when EXAMPLE is used by CALLER. BOS stands for Bottom Of Stack, sentry stands for the base address of a storage space.

Figure 2a: Just before execution of EXAMPLE



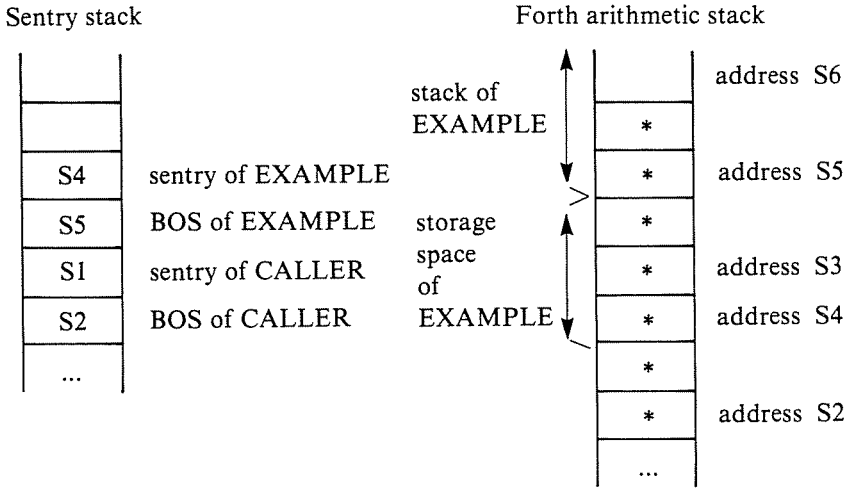
S3 denotes the top of CALLER's arithmetic stack. We check whether S3-S2 >= a (i.e., enough input values?).

Figure 2b: Just after the call is made



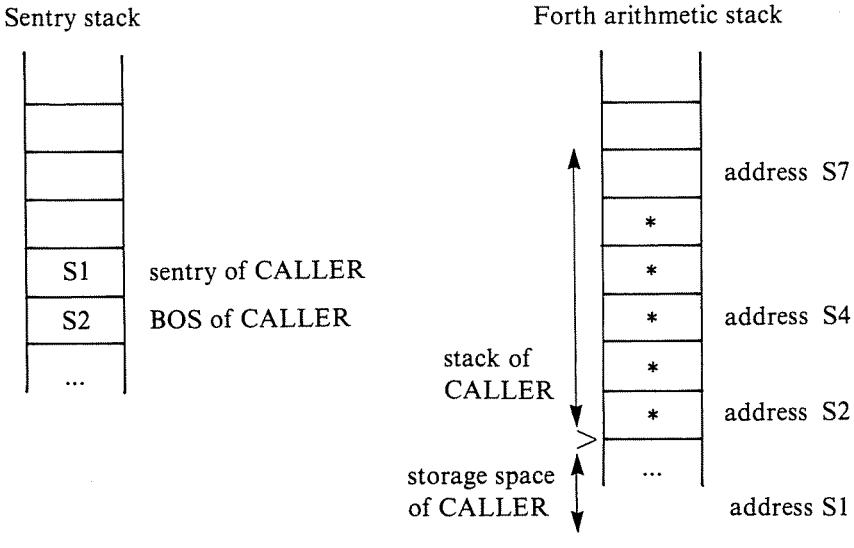
S5 denotes the top of EXAMPLE's empty stack. The input values are located from S4 (=S3–a) to S3 (excluding S3), the scratch values from S3 to S5 (=S3+b).

Figure 2c. During execution of EXAMPLE



S6 denotes the top of EXAMPLE's non-empty stack.

Figure 2d. Just after execution of EXAMPLE



S7 denotes the new top of CALLER's stack. EXAMPLE's output values are placed from S4 (former sentry of EXAMPLE) to S7 (=S4+c). EXAMPLE's input values on CALLER's stack are replaced by EXAMPLE's output values.

Generation of Code

Calculation of addresses of parameters

Every occurrence of a parameter in EXAMPLE's source code has to correspond in EXAMPLE's object code with the occurrence of code resulting in the proper actions for the simulated VALUE. This code uses the routine CALC to add the relative address within the storage space to the topmost sentry on the Sentry stack to obtain the runtime address of the parameter. After this, the routine EXECTO is used to take appropriate action with the calculated address (see [JOOS 83] for the implementation of the data structure VALUE). We define the routine CALC+EXECTO to use the next byte in the object code as an inline argument to calculate the corresponding run time address and to call EXECTO to deal with this address. In this way, every occurrence of a parameter corresponds to three bytes of object code (of course this is machine-dependent). See [JOOS 82] for the concept of inline arguments.

We will implement the generation of the code for the calculation of the run time addresses of the parameters in the following way. The declarations will create for every parameter a 'generator routine' that compiles the routine CALC+EXECTO and the corresponding relative address within the storage space as an inline argument when the corresponding parameter is used. As these generator routines have to execute during EXAMPLE's compilation, they must have precedence.

The generator routines should be removed from the dictionary when EXAMPLE's compilation has finished, otherwise they would occupy too much memory. We implement this removal in the following way: The generator routines reside between EXAMPLE's header and EXAMPLE's executable code. After EXAMPLE's compilation, the generator routines are erased by moving EXAMPLE's executable code towards its header. This works well with Forth versions using relative branches for the implementation of the control structures. Other Forth versions will require other implementations.

Generation of storage spaces and stacks

When the last declaration has finished, it compiles the routine SETUP, with the number of input values and scratch values as inline arguments. The runtime actions of the routine SETUP are:

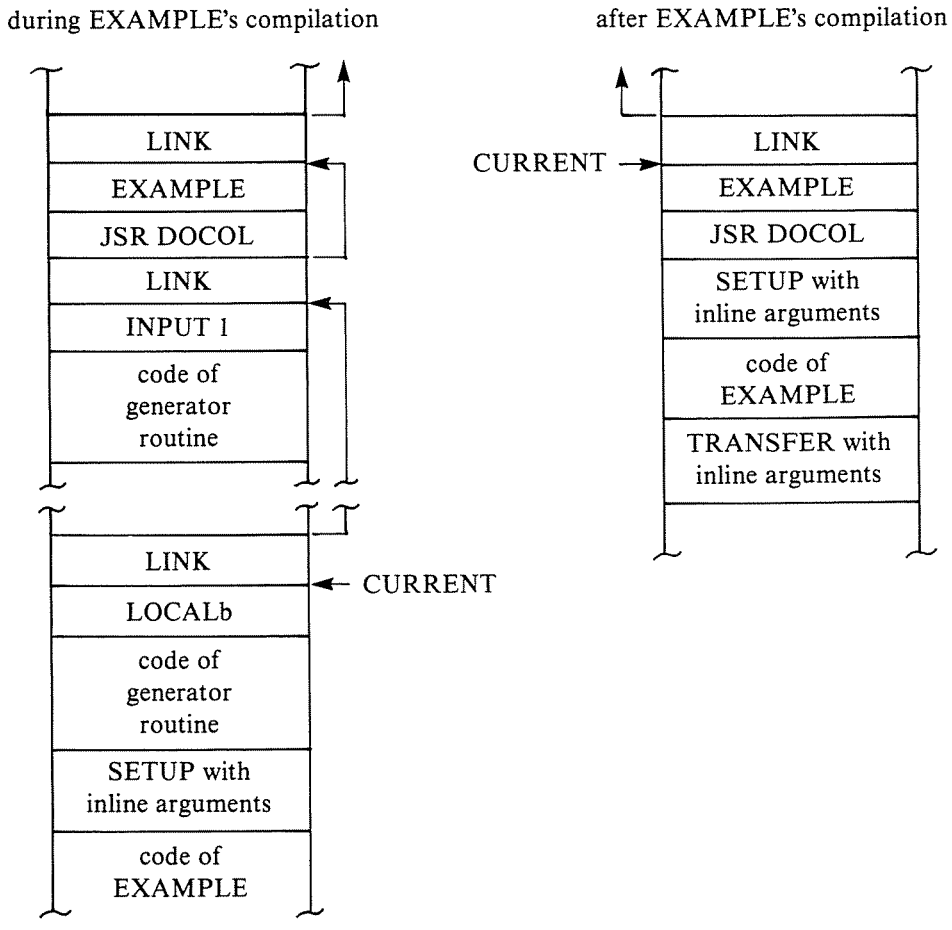
— Allocation of a new storage space, implicitly transferring the input values.
— Allocation of a new stack.

We cannot transfer EXAMPLE's output values directly from its storage space to CALLER's stack as the transferred output values cause CALLER's stack to coincide with EXAMPLE's storage space, possibly erasing some parameters that still have to be used as output values. To implement the transfer of output values properly, OUTPUT: compiles the routine TRANSFER, the number of output values and the relative addresses of the output values in the storage space. The runtime actions of the routine TRANSFER are:

— Use the compiled inline arguments as relative addresses to place the output values in the right order on EXAMPLE's stack.
— Transfer the output values to CALLER's stack.
— Remove EXAMPLE's storage space and stack.

When TRANSFER is executed, the parameters cannot be used anymore, as EXAMPLE's storage space and stack no longer exist. The routine ; has to return control to the calling routine in the usual way. The parameters are therefore local to the respective routines that use them.

Figure 3: Summary



1. In Fys-Forth the link-field is placed in front of the headers of dictionary entries.
2. The code resulting from the declarations and the OUTPUT-statement(s) consists only of the addresses of SETUP and TRANSFER with their inline arguments.
3. When an error occurs during the compilation of EXAMPLE, all the generator routines and the existing code of EXAMPLE are removed from the dictionary.

Generalization: Parameter Data Structures

Most FORTH systems support the data structures CONSTANT and VARIABLE. Whenever a user needs other data structures, he can use the CREATE-DOES>-mechanism to implement them. We propose an analogous mechanism for the creation and implementation of data structures that can be used to pass information to and from routines. We will use the notion "parameter data structures" for these data structures. The new mechanism should be used whenever the parameters implemented in the previous chapters do not suffice.

Implementation

In order to be passed to routines, the input and scratch values have to be placed on the stack of the calling routine. When parameter data structures themselves have to be placed on a stack to be passed to a routine, we have to restrict their allowable size severely. To avoid this, we implement these data structures in the storage space as pointers to the specified data structures. This implies that these data structures are passed to and from routines by means of pointers.

Dynamic allocation

When a definition starts to execute, its parameter data structures used for input exist in memory. Those used for temporary storage still have to be generated. There are several ways to implement this generation, but they all have to face the following problems:

— Where in memory are these data structures allocated?
— Is the entire required space allocated when the routine starts to execute, or is the space separately allocated for every entity?
— Is the entire allocated space de-allocated when the routine finishes its execution? When this is the case, it is impossible to use the generated entities for output. The other case can only be properly handled by means of a memory manager.

Syntax

We propose the following syntax for the specification of parameter data structures for input:

```
INPAR PARAM
    ...                 ( describe the runtime actions; the runtime address in the )
;                       ( storage space is found on the stack.                     )
```

Whenever PARAM, which is an arbitrary identifier, is used within a definition, it creates generator routines for every identifier in the list of identifiers following it. This implies that PARAM has precedence. It is not possible in our implementation that parameters or parameter data structures be used to specify the new parameter datastructure. However, routines using parameters or parameter datastructures can be used. The order of the declarations does not matter, as long as the scratch values are declared either last or first.

Some examples of possible parameter datastructures for input are:

```
INPAR INCONST:
    ( Routines expect on the stack a pointer to a CONSTANT. Within the routines, )
    ( a CONSTANT is "seen".                                                       )

    @ ;

    ( Fetch the contents of the location the pointer in the storage space points to, )
    ( to simulate a CONSTANT for the routines.                                       )

INPAR INPTR:
    ( Routines expect on the stack a pointer to a VALUE. Within the routines, )
    ( a VALUE is "seen".                                                       )

    EXECTO ;

    ( Call EXECTO to act upon the location the pointer in the storage space points )
    ( to, to simulate a VALUE for the routines.                                    )
```

```
INPAR INVARRAY:
    ( Routines expect on the stack a pointer to an array of VALUE's. Within the )
    ( routines, an array of VALUE's is "seen".                                   )

    SWAP                                          ( put offset on top )
    2*                                            ( object size is two bytes )
    +                                             ( calculate address of element )
    EXECTO ;                                      ( simulate VALUE in the array )

    ( Use the offset on the stack to index within the array the pointer in the storage )
    ( space points to, to simulate a VALUE in an array.                                 )
```

It is also possible that the datastructures pass some actions to routines, as in the following examples:

```
INPAR INSTRING:
    ( Routines expect on the stack a pointer to a string. Within the routines, a string )
    ( is typed.                                                                          )

    COUNT TYPE ;

    ( Call COUNT and TYPE to type the string the pointer in the storage space points )
    ( to, to simulate the "TYPE"ing of a string for the routines.                    )

INPAR INROUTINE:

    ( Routines expect on the stack a pointer to a routine. Within the routines, )
    ( a routine is "executed".                                                   )

    CFA        ( change the parameter field address into a compilation field address. )
    EXECUTE ;                                                    ( execute the routine )

    ( Call CFA and EXECUTE to execute the routine the pointer in the storage space )
    ( points to, to simulate the "EXECUTE"ion of a routine for the definitions.    )
```

Some examples of routines that use parameter data structures are given in Appendix 2.

Note that neither the number of kinds of used input data structures, nor the amount of code for the specification of the various parameter data structures, afflict the amount of generated executable code of routines that use parameter datastructures.

Pointers to parameter datastructures:

Parameter data structures consist of a pointer within the storage space. This pointer points to the data structure itself. In order to execute the specification of the parameter data structure, we have to place the location of the data structure on the stack. This can be done by means of the routines CALC and @. Therefore these two routines are automatically compiled in front of the code associated with the specification of the data structure.

When parameter data structures are used as input for another definition, we have to compile a routine placing the run time location of the data structure on the stack. Whenever a parameter data structure is used within the source code, the specification routine is compiled in the object code. To avoid this, we use the new, immediate, routine REF. It compiles the routine CALC+@ and the relative address within the storage space associated with the parameter following REF in the input stream.

We could use the routine ′ (tick) to obtain pointers to data structures with an associated *permanent* header in the dictionary, but, for sake of uniformity, we extend the actions of the routine REF for these cases. A possible implementation of REF is given.

Use of the routine REF is the only way to pass a parameter data structure to another definition. Note that REF X, where X corresponds to a parameter data structure, pushes at run time the contents of the location associated with X within the storage space on the stack of the calling definition. That is, it pushes the address of the data structure that X denotes on the stack, not the location of X within the storage space.

Conclusions

We have formalized the stack usage by defining formal stack entries for values and user-defineable parameter data structures. This resulted in a description of a syntax that offers extreme clarity of source code. We can make the following remarks about the new syntax and implementation:

— The input and scratch values we introduced offer a very elegant and structured way of number handling. FORTH code using these parameters can be made independent of the arithmetic stack.
— Parameter datastructures allow the abstraction from the physical implementation of the used data structures.
— The possible independence and abstraction enable the programmer to work at a higher, easier level. This will reduce the time required for coding and debugging.
— This higher level implies that parameters are particularly useful to encode elaborate definitions, and to test the correctness of algorithms quickly. Small and simple routines can still be written in parameter-less FORTH. Both kinds of levels are available and can be used together in one program.
— We have chosen to implement the input and scratch values as FYS-FORTH VALUE's, but they can be implemented as Forth-79 VARIABLE's.
— A preliminary test with our implementation shows an increase of the execution time and amount of generated code in the order of 10–30%.
— Our implementation can still be improved. Assuming that most routines will use at most four input and scratch values, we can compile four permanent definitions in the dictionary to be compiled instead of CALC+EXECTO and its inline argument. In this way we could reduce the amount of generated code by two bytes instead of three for every occurrence of one of these parameters, and the execution time (no need anymore to fetch the inline argument).
— To optimize the execution time of the parameter data structures, we can implement a routine INCODE that allows the use of assembly language only to specify the runtime actions of a new parameter data structure.
— Once a program has been developed, it may be optimized by partly rewriting the time-critical routines in parameter-less FORTH, in machine language or in micro-code. The version of the program that uses parameters can then still be used as documentation.

Acknowledgements

We want to thank prof. dr. J. v. Leeuwen of the department of Computer Science of the State University of Utrecht for giving one of us the opportunity to work a year at the Observatory for the final phase of his study.

We also want to thank Rieks Joosten and Frans Cornelis for the time they spent discussing and analyzing this concept and its use. We also thank Hans v. Koppen for the use of his room and his Apple II computer.

References

[BART 79] P. Bartholdi, "The "TO" Solution", *FORTH Dimensions*, vol. 1 no. 4, 1979, San Carlos, Calif., pp. 38–41.

[BART 81] P. Bartholdi, "Another Aid For Stack Manipulation And Parameter Passing In FORTH", *Proceedings Of The 1981 Rochester FORTH Conference On Data Bases And Process Control*, Rochester, New York, pp. 217–226.

[JOOS 82] R. Joosten, Working Group Report, *Proceedings Of The 1982 Rochester FORTH Conference On Data Bases And Process Control*, Rochester, New York, pp. 267–277.

[JOOS 83] R. Joosten, *FYS FORTH 0.2/0.3 Preliminary User Manual*, State University Of Utrecht, Utrecht, the Netherlands, 1983.

[McKI 81] D. McKibbin, "Parameter Passing To DOES>", *FORTH Dimensions* vol. 3 no. 1, 1981, San Carlos, Calif. p 14.

To obtain a copy of the FYS-FORTH manual of FYS-FORTH 0.3 please contact: H. Nieuwenhuijzen, Sterrewacht "Sonnenborgh", Zonneburg 2, 3512 NL Utrecht, the Netherlands.

Siem Korteweg studied mathematics and computer science at the University of Utrecht, Utrecht, Holland. He formalized some concepts of FYS.FORTH, the local FORTH dialect.

Dr. Hans Nieuwenhuÿzen received a Ph.D. in astronomy and physics from the State University of Utrecht in Utrecht, Holland in 1970. He is a senior research fellow at the University and promotes Forth for interactive environments. Dr. Nieuwenhuÿzen is currently teaching courses in digital data processing including data transport, data base access and processing for applications in remote sensing, astronomy, high energy physics and medicine.

Manuscript received November 1983.

Appendix 1: Highlevel implementation

This code provides a highlevel implementation of parameters. This code is not optimized, nor does this implementation offer stack security. The latter is not present as it has to be implemented within the FORTH-kernel.

```
BASE HEX
( *** BUFFER, SENTRY STACK AND HELPVALUES *** )
CREATE BUFF 51 ALLOT                    ( max. 1 line of 80 characters )
VARIABLE U.STOP HERE U.STOP ! 10 ALLOT                  ( sentry stack )
U.STOP @      VALUE BOTTOM
U.STOP @ F + VALUE TOP
0 VALUE #INPUT            0 VALUE #LOCAL          0 VALUE #OUTPUT
0 VALUE BEGIN.PAR.CODE 0 VALUE END.PAR.CODE
0 VALUE HOLD.IN           0 VALUE HOLD.TIB         0 VALUE MAX.IN
0 VALUE SAVE.SP           0 VALUE #SCRATCH

( sentry and arithmetic stack handling )

: SS!                     ( -- -> -- )
  BOTTOM U.STOP ! ;                                  ( empty sentry stack )
: >S                      ( N -> -- )
  U.STOP @ TOP = IF SS! 33 ERROR THEN                     ( stack overflow )
  1 U.STOP +! U.STOP @ C! ;                     ( transfer to sentry stack )
: S>                      ( -- -> N )
  U.STOP @ BOTTOM = 34 ?ERROR                  ( stack underflow; no SS! )
  U.STOP @ C@ -1 U.STOP +! ;                   ( transfer to arithm. stack )
: S                       ( -- -> N )
  U.STOP @ C@ ;                                ( copy to arithmetic stack )
CODE SP                   ( -- -> N )                ( value of stackpointer )
                TXA,                        ( stackpointer into accumulator )
   PUSH         JMP,
ENDCODE
CODE !SP                  ( N -> -- )
    1 ,X        LDA,                                  ( N into accumulator )
                TAX,                          ( assign N to stack pointer )
    NEXT        JMP,
ENDCODE
```

```
( *** RUNTIME ROUTINES *** )
: INLINE1                    ( -- -> N )
  R>
  ARG.ADDR C@ 1 SKIP.ARG
  SWAP >R ;
: INLINE2                    ( -- -> N )
  R>
  ARG.ADDR @ 2 SKIP.ARG
  SWAP >R ;
: CALC+EXECTO                    ( effects depend upon actions of EXECTO )
   S INLINE1 -                            ( -: stack grows to low memory )
   1- EXECTO ;                         ( 1-: compensate reversed byte order )
: CALC+@                     ( -- -> N )
   S INLINE1 -                            ( -: stack grows to low memory )
   1- @ ;                              ( 1-: compensate reversed byte order )

( creation and deletion of stacks and storage spaces )

: SETUP                      ( -- -> ? )
        ( allocate new storage space and stack, initialize scratch values to zero. )
   SP DUP TO SAVE.SP                     ( save start address scratch values )
   DUP  INLINE1 +                                ( SP+#INPUT=new sentry )
   >S                                       ( new sentry on sentry stack )
   INLINE1 DUP TO #SCRATCH                  ( save number scratch values )
   -                                        ( SP-#SCRATCH=new BOS )
   !SP                                            ( initialize new stack )
   SAVE.SP #SCRATCH - 1+ #SCRATCH ERASE ;
                                            ( initialize the scratch values to 0 )

: TRANSFER                   ( -- -> ? )
   ( transfer output values to stack of calling routine, and update the Sentry )
   ( stack and stack pointer.                                              )
   INLINE1 DUP TO #OUTPUT                   ( save number of output values )
   2/ 0                                     ( #allocated.bytes/2=#output.values )
   ?DO S INLINE1 - 1- @ LOOP                        ( output values on stack )
   SP 1+ S #OUTPUT - 1+ #OUTPUT CMOVE                    ( transfer output )
   S> #OUTPUT - SP!                   ( S-#OUTPUT=new SP; s-stack updated )
   0 TO #OUTPUT ;

: CLEANUP                    ( -- -> -- )
   END.PAR.CODE                         ( any code compiled for the params? )
   IF BEGIN.PAR.CODE @ CURRENT !  b                  ( restore CURRENT )
      END.PAR.CODE                                   ( erase the generator )
      BEGIN.PAR.CODE                          ( routines of the parameters. )
      HERE END.PAR.CODE - CMOVE
      BEGIN.PAR.CODE END.PAR.CODE - ALLOT                     ( adjust DP )
      COMPILE TRANSFER  0 C,                                ( 0: no output )
      0 TO #INPUT          0 TO #LOCAL          0 TO #OUTPUT
      0 TO BEGIN.PAR.CODE 0 TO END.PAR.CODE
   THEN ;

( *** COMPILETIME ROUTINES *** )
```

```
: :                          ( -- -> -- )
  [COMPILE] :                ( the new : and ; only have new effects when )
                             ( parameters are used.                       )
  HERE TO BEGIN.PAR.CODE ; IMMEDIATE

: ;                          ( -- -> -- )
  CLEANUP [COMPILE] ; ; IMMEDIATE

( security routines )

: ?ILLEGAL.ORDER             ( ?illegal.declaration  -> -- )
   5 ?ERROR ;

: ?TOO.MUCH.CODE             ( -- -> -- )
   ( checks against compilation of code between declarations )

  END.PAR.CODE                                    ( SETUP compiled? )
  IF    -4 ALLOT                     ( remove SETUP and inline arguments )
        END.PAR.CODE
  ELSE BEGIN.PAR.CODE
  THEN
  HERE <>                                         ( code compiled yet? )
  5 ?ERROR ;                                            ( signal error )

: SECURITY.INPUT             ( -- -> -- )
  #LOCAL ?ILLEGAL.ORDER                ( INPUT: must precede LOCAL: )
  ?TOO.MUCH.CODE ;                    ( no code should be compiled yet )

: SECURITY.LOCAL             ( -- -> -- )
  ( 0 ?ILLEGAL.ORDER )                          ( LOCAL: should be last )
  ?TOO.MUCH.CODE ;                    ( no code should be compiled yet )

( building routines )

: PRELUDE
  ( transfers a list of identifiers to a new input buffer )
  -WORD
  IN TO HOLD.IN TIB TO HOLD.TIB                       ( save IN and TIB )
  COUNT
  2DUP + &, SWAP C!                          ( add extra ',' to the list )
  DUP 1+ TO MAX.IN                     ( max. offset in new input buffer )
  BUFF SWAP 1+ CMOVE                     ( move list to new input buffer )
  BUFF TO TIB 0 TO IN ;                          ( assign new TIB and IN )

: NEW.BUILD.HEADER           ( -- -> -- )
  ( when optimized in assembler, watch COMP. COLONDEF )
  HERE
  CURRENT @ ,                                       ( create a linkfield )
  CURRENT !                                       ( update dictionary link )
  &, WORD COUNT                                     ( read new identifier )
  DUP 1+ ALLOT                                ( add identifier to dictionary )
  OVER + 1- 80 TOGGLE                          ( give last character parity )
  1- C0 TOGGLE ;          ( give new routine precedence, and countbyte parity )
```

```
20 CONSTANT JSR,

: COMP.COLONDEF          ( -- -> -- )
   JSR, C,
   ' NEW.BUILD.HEADER CFA 1+ @ , ;                    ( 1+: skip JSR )

: GENERATOR              ( -- -> -- )
   INLINE1 INLINE2                        ( -- -> offset,addr.runtime-part )
   , C, ;

: SINGLE.BUILD           ( addr.runtime-part,offset -> -- )
   NEW.BUILD.HEADER                       ( create new entry in dictionary )
   COMP.COLONDEF
  ( when optimizing the runtime code of the parameters, we must decide here )
  ( whether we compile the usual three bytes or the optimized two bytes. The )
  ( routines to be compiled:                                                 )
  ( : INPO S 1- EXECTO ; : INP1 S  3 - EXECTO ;                              )
  COMPILE GENERATOR
  C, ,                          ( compile inline arguments of GENERATOR )
  COMPILE EXIT ;                          ( complete the generator routine )

: PARAMETER.BUILDER                ( addr.runtime-part,beg.offset -> #alloc. )
        ( this way of parameter passing to PARAMETER.BUILDING allows consecutive )
                                        ( occurrences of INPUT: and LOCAL:. )
  SWAP OVER                                     ( save beg.offset on stack )
  PRELUDE                                ( transfer idents. to new input buffer )
  BEGIN MAX.IN IN >                                ( more idents. in buffer? )
  WHILE 2DUP SINGLE.BUILD                         ( create generator routine )
        2+                                                  ( increment offset )
  REPEAT
  HOLD.IN TO IN HOLD.TIB TO TIB                          ( restore TIB and IN )
  UNDER SWAP - ;                         ( end offset - beg.offset =#allocations )

: START.CODE              ( complete temporary structure produced by the )
  HERE TO END.PAR.CODE ( declarations, and start with the "real" code of the )
  COMPILE SETUP        ( definition, i.e. SETUP with inline arguments      )
  #INPUT C,
  #LOCAL C, ;
```

```
( general input and scratch building routines )

: TRANSFER.ERROR          ( <> -> <> )
  HOLD.TIB TO TIB HOLD.IN TO IN                    ( restore TIB and IN )
  CLEANUP                          ( restore usual FYS-FORTH dict. struct. )
  ERR ERROR ;                                 ( transfer error to INTERPRET )

: INPUT.BUILDER           ( addr.runtime-part -> -- )
  SECURITY.INPUT
  #INPUT PARAMETER.BUILDER          ( create a generator routine for every )
                                              ( identifier in the input buffer )
  +TO #INPUT
  START.CODE
  ONERR> TRANSFER.ERROR ;

: SCRATCH.BUILDER         ( addr.runtime-part -> -- )
  SECURITY.LOCAL
  #INPUT #LOCAL + PARAMETER.BUILDER       ( create a generator routine )
                                        ( for every identifier in input buffer )
  +TO #LOCAL
  START.CODE
  ONERR> TRANSFER.ERROR ;

( specific input and scratch building routines )

: INPUT:

  ' CALC+EXECTO CFA INPUT.BUILDER ; IMMEDIATE

: LOCAL:
  ' CALC+EXECTO CFA SCRATCH.BUILDER ; IMMEDIATE
```

```
( *** THE OUTPUT: ROUTINE *** )
: OUTPUT:                    ( -- -> -- )
   ( this version requires all the identifiers of the output values to fit in one list. )
   PRELUDE                                  ( transfer idents. to new input buffer )
   BEGIN.PAR.CODE @                                      ( old value of CURRENT )
   0 BEGIN.PAR.CODE !                                  ( make temporary end of chain )
   ( Limit searched vocabulary to parameters only. In this way we can only output )
   ( the parameters.                                                              )
   0 TO #OUTPUT                                                      ( initialization )
   COMPILE TRANSFER
   HERE 1 ALLOT                          ( prepare backpatching of inline argument )
   BEGIN MAX.IN IN >                                     ( more idents. in buffer? )
   WHILE &, WORD CURRENT FIND 0=                          ( non-existing param? )
         IF BEGIN.PAR.CODE !                                          ( fix the chain )
            35 ERROR                                       ( illegal output parameter )
         THEN
         2+                                                   ( 2+: skip GENERATOR )
         C@ C,                          ( offset is inline argument for TRANSFER )
         2 +TO #OUTPUT
   REPEAT
   #OUTPUT SWAP C!                          ( backpatch inline arg. of TRANSFER )
   BEGIN.PAR.CODE !                                   ( fix the chain of vocabulary )
   HOLD.IN TO IN HOLD.TIB TO TIB
   ONERR> TRANSFER.ERROR ; IMMEDIATE

( *** GENERAL PARAMETER DATA STRUCTURES *** )

: INPAR                      ( -- -> -- )
  [COMPILE] :                                        ( create a new routine: PARAM )
  HERE NFA 40 TOGGLE                                   ( give new routine precedence )
  COMPILE LIT
  HERE 2 ALLOT                          ( prepare backpatching of arg. of INP.BLD )
  COMPILE INPUT.BUILDER
  COMPILE EXIT                                                  ( finish actions of PARAM )
  HERE SWAP !                                          ( backpatch arg. of INP.BLD )
  ( the remaining code must be adapted to implement INCODE )
  COMP.COLONDEF
  COMPILE S
  COMPILE INLINE1
  COMPILE -
  COMPILE 1-
  COMPILE @ ; IMMEDIATE

: REF                        ( -- -> parameter field address )
  -FIND 0= 4 ?ERROR DUP
  BEGIN.PAR.CODE U> END.PAR.CODE 0= 0= AND                          ( parameter? )
  IF     COMPILE CALC+@
         2+                                                   ( 2+: skip GENERATOR )
         C@ C,                          ( offset is inline argument for CALC+@ )
  ELSE [COMPILE] LITERAL
  THEN ; IMMEDIATE

TO BASE
```

Note: To use the parameters as implemented above, the headers of the following routines must reside in the dictionary:

INPUT:, LOCAL:, OUTPUT:, INPAR, REF, SS!, : and ;

Use of the routine SS! is recommended whenever an error occurs during the execution of a routine using parameters.

Appendix 2: Some examples

Factorial

In usual implementations a routine calculating the factorial of a given number could look like:

```
: FACTORIAL                                        ( N -> FAC N )
  ?DUP                                             ( recursion? )
  IF    DUP 1- MYSELF                              ( recursively calculate FAC N-1 )
        *                                          ( FAC N = N * FAC N-1 )
  ELSE  1                                          ( FAC 0 = 1 )
  THEN ;
```

This source is not very difficult to understand, and therefore it is not necessary to use parameters for this definition. To show some possibilities of our implementation we will give some versions of FACTORIAL using parameters.

```
: FACTORIAL.1                                      ( NUMBER -> FAC NUMBER )
  INPUT: NUMBER        ( take one number from the stack and label it locally )
  NUMBER                                           ( recursion? )
  IF    NUMBER 1- MYSELF        ( recursively calculate FAC NUMBER-1 )
        NUMBER *        ( FAC NUMBER = FAC NUMBER-1 * NUMBER )
  ELSE 1                                           ( FAC 0 = 1 )
  THEN
  TO NUMBER                          ( assign FAC NUMBER to NUMBER )
  OUTPUT: NUMBER ;
  ( use input value to leave FAC NUMBER on stack )
```

To calculate FAC 5 use: 5 FACTORIAL.1 . ==> 120 OK

```
: FACTORIAL.2                                      ( NUMBER -> FAC NUMBER )
  INPUT: NUMBER
  LOCAL: RESULT                  ( introduce a locally labelled scratch value )
  NUMBER                                           ( recursion? )
  IF    NUMBER 1- MYSELF        ( recursively calculate FAC NUMBER-1 )
        NUMBER *        ( FAC NUMBER = FAC NUMBER-1 * NUMBER )
  ELSE 1                                           ( FAC 0 = 1 )
  THEN
  TO RESULT                          ( assign FAC NUMBER to RESULT )
  OUTPUT: RESULT ;
  ( use scratch values to leave FAC NUMBER on stack )
```

To calculate FAC 5 use: 5 FACTORIAL.2 . ==> 120 OK

Binary Search

We will now encode a more complicated algorithm, making the following assumptions:

— an array contains objects consisting of 2 bytes.
— the objects in the array are in a particular order.
— ORDERING compares 2 objects.

```
: BINSEARCH.1
   INVARRAY: ROW
   INPUT:      UPPERBOUND,OBJECT
   LOCAL:      FOUND,LEFT,MIDDLE,RIGHT
   ( FOUND and LEFT are properly initialized: 0 )
   UPPERBOUND TO RIGHT
   BEGIN RIGHT LEFT <>
   WHILE RIGHT LEFT + 2/ TO MIDDLE
         MIDDLE ROW OBJECT ORDERING
         DOCASE
          -1 CASE MIDDLE LEFT =
                  IF  LEFT TO RIGHT                    ( not present )
                  ELSE MIDDLE TO LEFT
                  THEN                        ELSE
           0 CASE 1 TO FOUND
                  LEFT TO RIGHT               ELSE
           1 CASE MIDDLE TO RIGHT
         ENDCASE
   REPEAT
   FOUND
   IF     OUTPUT:MIDDLE,FOUND
   ELSE   OUTPUT:FOUND
   THEN ;

: BINSEARCH.2
   ROT ROT 0
   BEGIN 2DUP <>
   WHILE 2DUP + 2/ DUP
         2* 5 PICK + @
         6 PICK ORDERING
         DOCASE
          -1 CASE 2DUP =
                  IF    2DROP DUP
                  ELSE  UNDER
                  THEN                ELSE
           0 CASE UNDER UNDER UNDER
                  UNDER 1 EXIT        ELSE
           1 CASE ROT DROP SWAP
         ENDCASE
   REPEAT
   2DROP 2DROP 0 ;
```

Note the differences in selfdocumenting power.

Parameter Data Structures

Another version of FACTORIAL, using a pointer to a VALUE for input and output looks like:

```
: FACTORIAL.3                                          ( N -><> )
  INPTR: IN/OUT                  ( pointer to location for input and output )
  IN/OUT                                                ( recursion? )
  IF    IN/OUT                                          ( save N on stack )
        -1 +TO IN/OUT                                   ( put N-1 in IN/OUT )
        REF IN/OUT                                      ( put input/output place on stack )
        MYSELF                                          ( recursively calculate FAC N-1 in IN/OUT )
        IN/OUT *                                        ( FAC N = FAC N-1 * N )
  ELSE 1                                                ( FAC 0 = 1 )
  THEN
  TO IN/OUT ;                                           ( place FAC N in input/output place, )
                                                        ( nothing is left on the stack.      )
```

To calculate FAC 5 use:

```
5 TO NUMBER  REF NUMBER FACTORIAL.3
             NUMBER                 ==>  120  OK
```

Parameter data structures can be used to extend a particular FORTH system with recursion, cf. the following version of FACTORIAL:

```
: FACTORIAL.4                                          ( CFA N -> FAC N )
  INROUTINE: "MYSELF"
  INPUT:     NUMBER
  NUMBER                                                ( recursion? )
  IF      REF "MYSELF"
          NUMBER 1-
          "MYSELF"                                      (recursively calculate FAC N-1 )
          NUMBER *   ( FAC NUMBER = FAC NUMBER-1 * NUMBER )
  ELSE 1                                                ( FAC 0 = 1 )
  THEN
  TO NUMBER                                             ( assign FAC NUMBER to NUMBER )
  OUTPUT: NUMBER ;
  ( use input values to leave FAC NUMBER on stack )
```

To calculate FAC 5 use: REF FACTORIAL.4 5 FACTORIAL.4 . => 120 OK

This is a complicated way to calculate FAC 5, but this is merely an example.

Appendix 3: Deviations from the '79-STANDARD:

See [JOOS 83] for Fys-Forth Deviations from Forth-79. In addition, this paper uses the following non-standard words:

The used NON '79-Standard words:

&^ IMMEDIATE
&^ -> <ASCII-VALUE.OF.^>
The ascii value of the second character in the name of this word will be pushed on the stack. In compilation mode, this ascii value is compiled as a literal. The ^-character cannot be a blank or a control character.

IMMEDIATE
' #<WORD># -> PFA
Searches the context vocabulary for <WORD>; when it is found, ' leaves the parameter field address of the definition. In compile mode, it compiles this PFA as a literal. When <WORD> is not found, an error occurs (ERROR #4: CAN'T FIND).

+TO +TO -> <>
Sets the action of the following VALUE type to add the integer on top of the stack to the data contained in the VALUE.

-FIND -FIND #<ROUTINE.NAME># -(in dict?)-> <PFA> 1
—(otherwise)-> 0
Searches the context vocabulary for the routine with name <ROUTINE.-NAME>. When it can be found, it leaves the pfa and a "true"-flag (=1), otherwise a "false"-flag is left.

?ERROR <BOOLEAN> <ERROR.NUMBER> ?ERROR -> <>
Executes ERROR with ERROR.NUMBER as argument when <BOOLEAN> =1 (i.e. TRUE).

ARG.ADDR ONLY WHILE COMPILING
ARG.ADDR -> <ADDR>
Puts the address on the stack of the inline argument of the routine that called ARG.ADDR.

C, <INTEGER> C,-> <>
Adds the lower byte of the value on top of the stack to the dictionary.

ERR ERR -> <ERR.VALUE>
User-accessible VALUE that holds the current error number.

ERROR <ERROR.NUMBER> ERROR -> <>
Issues an error message depending upon <ERROR.NUMBER>, and calls ABORT.

EXECTO <+VALUE> <ADDR> EXECTO -(%val-flag = -1) -> <>
<ADDR> EXECTO -(%val-flag = 0) -> [<ADDR>]
<VALUE > <ADDR> EXECTO -(%val-flag = 1) -> <>
Acts on the address on top of the stack depending on the value of the %val-flag. The flag is reset to zero afterwards. The following cases are implemented:
-1 :the <+VALUE> is added to the contents of <ADDR>.
0 :the contents of the address replaces the address.
1 :the <VALUE> is stored in <ADDR>.

FIND <STRING.ADDR> <LINK> FIND -(if found)-> <PFA> 1
-(otherwise)-> 0
Searches for the dictionary entry, whose name is at <STRING.ADDR> as a string, from <LINK> onwards until the entry is found. FIND leaves the Parameter Field Address of the entry and a "true"-flag (=1) when it is found, otherwise only a "false"-flag is left.

IN IN -> <IN.VALUE>
User accessible VALUE that keeps track of how far the input stream already has been processed.

MYSELF IMMEDIATE ONLY WHILE COMPILING
MYSELF -><>
Compiles the address of the routine currently being compiled. In this way recursive routines can be made.

SKIP.ARG ONLY WHILE COMPILING
<NRBYTES> SKIP.ARG -><>
Skips the inline argument of the routine that called SKIP.ARG for the address interpreter; <NRBYTES> is the length of the inline argument in bytes.

SP SP-> <SP>
Puts on the stack the value the stackpointer had just before this routine was called.

TO TO -><>
Sets the action of the following VALUE type to store the number on top of the stack in it.

UNDER <INTEGER.2> <INTEGER.1> UNDER -> <INTEGER.1>
Removes the number 2nd to the top of the stack.

VALUE IMMEDIATE
<START.VALUE> VALUE #<VALUE.NAME># -> <>
Creates a value, initializing its contents to the number on top of the stack.