Context-Oriented Programming: Evolution of Vocabularies

M.L.Gassanenko
Russia

Abstract

Context-Oriented Programming (COP) came into being as a result of simplification and adaptation to Forth of Object-Oriented Programming (OOP) ideas. Like OOP, it allows the use of late binding and run-time inheritance. Unlike OOP,

1) class information is separated from data;
2) late binding methods are supported even for primitive data elements (such as values on the stack, addresses, etc.) that are not objects in an OOP sense);
3) the class of object may be determined at run time by conventional means of the programming language;
4) no special encapsulation support is provided.

Through the use of COP idea, the mechanism of adding new definitions to vocabularies was generalized and extended to support operation tables and other vocabulary-like structures. A wordset for controlling the search order is also presented.

Introduction

Existing attempts to introduce OOP to Forth were not very successful and typically led to creation of another reversal polish language [2,4]. The OOP came into being as a result of evolution of a record (structure), but Forth does not support records. The two Forth random access data structures are memory and a vocabulary. In this paper we will not build records from memory but extend the vocabulary concept to suport various radom access structures.

Among the various object-oriented systems the language SELF [9,3] should be mentioned. In this prototype-based object-oriented language objects are pure contexts.

The implementation technique uses multiple code fields words that are described in [1,5,6].

To facilitate translation of this paper back into Russian the original Russian terms are given in quotes. The Russian alphabet encoding is given before the references list.

Useful Non-Standard Words

RUSH (cfa --> ; exit) works as EXECUTE EXIT except that it loads new IP from return stack before executing cfa.

RUSH> (--> ; control transfer) exits the current colon definition and executes the word compiled after it. In F-PC it is called GOTO.

```
: RUSH> R> @ RUSH ;
```

IT (--> cfa) last name code field.

```
: IT LATEST NAME> ;
```

or

```
: IT LAST @ NAME> ;
```

TO and IS Aliases. State-smart prefixes that allow to compile/execute the first additional code field of a multi-cfa word. Typically this code field is used to assign a value.

AT State-smart prefix that allows to compile/execute the second additional code field of a word. Typically this code field returns the parameter field address.

HAS (? -->) execute the 1st additional (assignment) code field of the last word.

E.g.:

```
VECT X    ' DUP HAS
```

is equivalent to

```
VECT X    ' DUP TO X
```

or

```
VOCABULARY XXX  HAS
<definitions>
```

is equivalent to

```
VOCABULARY XXX
TO XXX
<definitions>
```

```
IT IS           An idiom:
                VECT X
                . . . .
                : <smth> ... ; IT IS X

CF!         ( cfc cfa --> )     write code field contents cfc into the
                code field at cfa.

CF@         ( cfa --> cfc )     fetch cfc from the code field at cfa

CF,         ( cfc --> )         add a code field into the dictionary
                and fill it with cfc.
```

Mechanisms

A mechanism is a set of tools intended for work in the same application area ("predmetnaja oblast'"). In a particular case it may be a set of functions. In a broad sense this term also may refer to the application area objects (e.g. memory, etc.). It should be noted that a Forth system is a set of mechanisms, e.g.: memory management one (@ ! C@ C! CMOVE), stack (DUP SWAP , etc.), dictionary (HERE ALLOT , C,), etc.

Mechanisms may be nested one into another (i.e. implemented one by means of another), e.g. the dictionary resides in the memory and the dictionary management words are defined via the memory management ones. Taking this into account, the cases of mechanisms misnesting should be considered as design errors. For example, in F-PC the words @ , ! , etc. work not with all the memory but only with the code segment (i.e. with a memory subset). The dictionary is housed not in the @-addressable memory but in the "long" memory, outside the code segment. This results in importability of any code that uses the misnested mechanisms. The ANSI standard team follows the path of ignoring some of Forth mechanisms and proclaiming that all what has left is Forth --- yet another misdesign example.

So, a mechanism is a set of tools. The next step would be to implement the interchangeable mechanisms to work in different application areas by similar methods. Since application areas cannot be exchangeg or removed, we have only to implement interchangeable sets of functions that will work in these areas and will be addressed via the same "messages".

An actual mechanism is a set of functions that work in an application area. These functions are called actions ("faktichеskoe dejstvie"). They are analogous to the OOP methods and we will use this term as well as the first one. An envelope ("obolochka") of mechanism is a set of words (messages) that invoke the actual mechanism actions. Depending on which actual mechanism is active, envelope messages address to different actions.

The (vocabulary-like) structures that contain functions will be called contexts. The examples of such structures are: wordlists, operation tables, a sequence of actions that should be performed on ABORT and are specified in different modules, a definition with several code fields, etc.

Operation Tables

Operation tables allow to implement the interchangeable function sets that work in similar application areas. They are not a something new in the Forth community.

The most general defining word for messages is defined as:

```
: FMSG ( n cfa --> )      \ n is a message number and cfa calculates
                          \ the current active operation table address
     CREATE
               SWAP CELLS ( offset ) , ( function ) ,
     DOES>   >R
               R@ CELL+ @ EXECUTE R> @ + @ RUSH ;
\ RUSH works as EXECUTE EXIT
```

If the word F leaves the active operation table address on the stack, messages to this table may be defined as:

```
1 ' F FMSG msg1
2 ' F FMSG msg2
  . . . .
```

(In current implementation the 0-th cell in a table is reserved for extentions and has not yet been used.)

In most cases the table address calculation boils down to fetching a variable value. In these cases we may use a simple version:

```
: @MSG  ( n addr --> )
     CREATE
               SWAP CELLS ( offset ) , ( address ) ,
     DOES>
               2@ SWAP @ + @ RUSH ;
```

(Of course, it should be better done in assembler.)

In practice, for the cases that require more complex address calculation it is advisable to define special defining words.

Managing Vocabularies via Operation Tables

The words : , CREATE , etc. are defined as messages that execute an action from the active table. This enables managing diverse vocabulary-like structures by means of these words, and not just the wordlists. The structure of resulting system is quite recursive: the operation tables and the messages that work with them are elements of a wordlist; wordlists are managed via operation tables and the messages.

CURRENT is implemented as a stack rather than a variable. It contains an operation table address (top element), an address of target object (i.e. the object being filled) (the 2nd element), and probably some other information (e.g. it is convenient to keep the object length as the 3rd element). The operation table that contains the target object filling methods is called parent operation table (of the target).

The Context Filling Mechanism Messages Specification

Here we describe the set of messages that allow to add new definitions to a variety of vocabulary-like structures. Some of these words are best explained in Forth, so the code will be given. The question sign (?) in stack diagrams shows that parameters are not specified. We also do not specify what is an identifier --- a number, an address, or a something else.

This set of messages is a framework that may be customized for many kinds of vocabulary-like structures by means of operation tables (that it supports). These messages do not necessarily have a meaning for all possible vocabulary-like structures (contexts). For example, the actions HIDE , REVEAL , IMMEDIATE are defined as NOOP for operation tables. To customize this framework for a new context structure one needs to define a new operation table and fill it with proper functions.

1. Building an entry

ID (? --> id) Leaves an identifier on the stack. Different id sources are possible.

ID, (id -->) Compile the identifier into the dictionary.

ROOM (id --> room) Room is the address where a reference to a new object named id should be placed.

PROPER (id --> id flag) Flag is true if id is allowable.

?PROPER (id --> id) An error message if id is impermissible.
```
: ?PROPER PROPER 0= ABORT" impermissible identifier" ;
```

?WORD (id --> val flag) Flag is true if the context contains an object named id. Val is the same as id when flag is false, and is the object address if flag is true.

?WARNING (id --> id) Prints a warning if the object is already defined, or does nothing.

LATEST (--> nfa) Last definition name field address.

IT (--> cfa) Last definition code field address. In the 1st version it was called LASTCF.

!LATEST (nfa -->) Sets LATEST to nfa.

!IT (cfa -->) Sets IT to cfa.

HEADER (id -->) Builds a header for id.

DEFINE (? -->) Adds a new entry to the context. Does not build code fields.
```
: DEFINE ID ?PROPER ?WARNING HEADER ;
```

(ALIAS) (cfa -->) Make the last DEFINEd entry an alias of the given cfa. Implies that no code fields were built.

```
CREATE    ( ? --> )
              : CREATE DEFINE HERE !IT (VAR) CF, ;

:         ( ? --> ? )          Start a : definition.
              : : ?EXEC CREATE (NEST) !CODE !CSP HIDE ] ;

;         ( ? --> )            End the colon definition.
              : ; ?COMP ?CSP COMPILE EXIT REVEAL [COMPILE] [ ;
              The message words that call these ( : and ; ) actions
              are immediate, but they may not be immediate for some
              operation tables and non-immediate for others.
              Immediacy is a message attribute, not an action
              one, and should therefore be specified in message
              definitions rather than in actions ones.
```

2. Modification of the last entry

```
!CODE     ( cfc --> )          Write the code field contents (cfc)
              into the last word code field.
              : !CODE IT CF! ;

#!CODE    ( n cfc --> )         Write the code field contents (cfc)
              into the last word n-th code field.
              : #!CODE SWAP CFL * IT + CF! ;

HIDE      ( --> )              Make the last defined entry invisible,
              if it is possible.

REVEAL    ( --> )              Make the last defined entry visible,
              does nothing if HIDE does so.

IMMEDIATE ( --> )              Make the last defined entry immediate,
              if it is possble.
```

3. System demands

```
PARENTP   ( --> addr )         Addr is the address (within the CURRENT
              stack) of a pointer to the active operation table.

PARENT    ( --> addr )         Active operation table address.
              : PARENT PARENTP @ ;

OBJECT    ( --> addr )         The address of the target object being
              filled. It may be a wordlist, operation table, etc.
              : OBJECT PARENTP CELL+ @ ;

OBJECTL   ( --> len )          The object length.
              : OBJECTL PARENTP 2CELLS+ @ ;
              or calculates the length.
```

4. CURRENT stack manipulations

```
RECUR     ( --> )              Discard the information about the
              current target object from the CURRENT stack and recur
              to the previous one.

STRAIGHTEN ( --> )             Discard an inflector (see below).
```

RESUME (-->) Set the context necessary for the current active operation table to work. This message is executed every time the current stack changes, and prevents the operation tables from misbehaviour in a wrong context. This supports mechanism nesting.

5. Affecting the whole target object

INHERIT (addr -->) Copy the object housed at addr into the current target one. Useful for operation tables.
: INHERIT OBJECT OBJECTL CMOVE ;

(FORGET) (addr -->) Forget the definitions above the addr.

Since the words PROPER , ?PROPER , etc. do not consume id, it is possible to define them as:

```
DEF ?PROPER NOOP
```

and

```
TRUE CONSTANT PROPER
```

(remeber that the defining words are redefined to add new functions to different targets, in particular, to operation tables).

Probably, the word RESUME should be subdivided into three —— INITIALIZE , REMOVE and RESUME. The first would be executed on adding a new table onto the current stack the second would work on its removal, and the third --- on RECURring to it. Yet another candidate to be included here is the word DOES>.

The word INHERIT is used in the form:

```
TO <object1>
AT <object2> INHERIT
<definitions of <object1> methods that override inherited ones>
```

and copies the contents of <object2> into <object1>. By this means the methods that may be reused are inherited.

By way of illustration how the use of messages allows to define words what can work for different contexts let us consider the following definitions:

ITS' (? --> cfa) Fetches an identifier from its source, searches for it in the current object and returns the object address. Analogous to ' (tick), except that tick searches the search order.

```
: ITS' ID ?WORD 0= ABORT" -?" ;

: CONSTANT CREATE , (CONST) !CODE ;

: (DOES>) R> !CODE ;
: DOES> ?COMP COMPILE (DOES>) COMPILE-INSIR ; IMMEDIATE
```

DEF and ALIAS are two words that define aliases.

```
: DEF DEFINE ' (ALIAS) ;
: ALIAS DEFINE (ALIAS) ;

\ E.g.
\ DEF <BUILDS CREATE
\ ' CREATE ALIAS <BUILDS
```

OPERATION TABLES MANAGING METHODS

In current implementation these functions are defined via a special defining word (::) since the standard colon that doesn't support operation tables cannot be used for creation the first operation table. Here these definitions are given as if this mechanism was always present in system.

A name field in operation tables is a cell that points to the action routine; an identifier is an offset from the operation table beginning.

```
TO OPTAB        \ OPTAB is the operation table
                \ that will contain the following functions

: ID ' >BODY @ ;     \ It is important to know
                     \ where the message number is stored
DEF ID, DROP
: ROOM OBJECT + ;
: PROPER  DUP 0>
          OVER OBJECTL < AND
          OVER DUP ALIGNED = AND ;
: ?PROPER PROPER 0= ABORT" impermissible identifier" ;
: ?WORD   PROPER IF      ROOM @ TRUE
                 ELSE    0              THEN ;
DEF ?WARNING NOOP

: LATEST  PARENT [ CURRENT_MSG# 1+ ]CELLS+ @ ;
: IT      PARENT [ CURRENT_MSG# 2+ ]CELLS+ @ ;
: !LATEST PARENT [ CURRENT_MSG# 1+ ]CELLS+ ! ;
: !IT     PARENT [ CURRENT_MSG# 2+ ]CELLS+ ! ;

: HEADER ALIGN ROOM DUP !LATEST HERE SWAP ! ;
: DEFINE ID ?PROPER ?WARNING HEADER ;
: (ALIAS) LATEST ! ;
: CREATE DEFINE HERE !IT (VAR) CF, ;

: !CODE IT CF! ;
: #!CODE SWAP CFL * IT + CF! ;
DEF HIDE NOOP
DEF REVEAL NOOP
DEF IMMEDIATE NOOP
: PARENT PARENTP @ ;
: OBJECT PARENTP CELL+ @ ;
: OBJECTL PARENTP 2CELLS+ @ ;
```

```
: RECUR PARENTP [ 3 ]CELLS+ CURP! RESUME ;
DEF RESUME NOOP          \ No additional settings
DEF STRAIGHTEN NOOP      \ Inflectors discard themselves, parent
                         \ tables do nothing (see about weak
                         \ inflectors below)
: INHERIT OBJECT OBJECTL CMOVE ;
DEF STRAIGHTEN NOOP

RECUR     \ End of OPTAB definitions


          VOCABULARY MANAGING METHODS

TO WORDLIST    \ WORDLIST is the operation table
               \ that will contain the following functions
AT OPTAB INHERIT    \ The methods that are not redefined are the same
            \ as for operatiion tables  (they are shown in comments)
: ID BL WORD ;
: ID, ", ;
: ROOM HASH THREADS-MASK AND OBJECT >THREADS + ;
TRUE CONSTANT PROPER
DEF ?PROPER NOOP
: ?WORD DUP ROOM SEARCH-THREAD ;
: ?WARNING DUP ?WORD
          IF OVER .NAME ." already defined" CR THEN DROP ;

: LATEST  OBJECT [ 4 ]CELLS+ @ ;
: IT      OBJECT [ 5 ]CELLS+ @ ;
: !LATEST OBJECT [ 4 ]CELLS+ ! ;
: !IT     OBJECT [ 5 ]CELLS+ ! ;

: HEADER ( id ) DUP ROOM ALIGN
    ( id room ) HERE CELL+ +LINK
         ( id ) HERE !LATEST
         ( id ) ID, ALIGN ;
\ : DEFINE ID ?PROPER ?WARNING HEADER ;
: (ALIAS) ( cfa --> )
\ A cludge that builds 5 code fields that are aliases of the 5 code
\ fields starting from cfa. Had the headers reside in a separate
\ segment, only a pointer assignment would be needed.
;
\ : CREATE DEFINE HERE !IT (VAR) CF, ;
\ : !CODE IT CF! ;
\ : #!CODE SWAP CFL * IT + CF! ;
: HIDE   LATEST N>FLAG SMUDGE-FLAG TUCK !BITS ;
: REVEAL 0 LATEST N>FLAG SMUDGE-FLAG !BITS ;
: IMMEDIATE LATEST N>FLAG IMM-FLAG TOGGLE ;

\ PARENTP PARENT OBJECT
: OBJECTL THREADS-MASK CELL+ [ 7 ]CELLS+ ;
DEF INHERIT WRONG-OPER

RECUR     \ End of WORDLIST definitions
```

Syntax for Changing the Target Object

All the contexts are implemented as multi-cfa words. The main (0-th) one executes the default action: a wordlist adds to the search order, an operation table takes parameters (the target and probably its length) from the stack and adds them and itself onto the CURRENT stack. The 1st additional code field invokes the context parent operation table to add the context onto the CURRENT stack. The 2nd additional code field returns the parameter field address. The word RECUR allows to recur to the previous target object.

Adding new definitions to different target object looks like this:

```
TO FORTH
<FORTH definitions>
OP-TABLE PROCSEQ          \ We define a new operation table
   HAS                    \ HAS here is equivalent to TO PROCSEQ
   <PROCSEQ definitions>
   RECUR
<FORTH definitions>
   TO ASSEMBLER
   <ASSEMBLER definitions>
   RECUR
<FORTH definitions>
RECUR
<the CURRENT stack contains what it contained before>
```

BUMPER is an operation table a pointer to which always resides below the CURRENT stack bottom. Its methods print an error message and prevent the system from crush if the CURRENT stack is empty. RESUME is its only action defined as NOOP.

Inflectors and the Delegation Interpreter

Inflectors are operation tables that modify the behaviour of the parent operation table when are placed onto the CURRENT stack. A message word invokes an action from the top operation table (i.e. the table which address is on the top of CURRENT stack). This action may be the PASS routine that causes execution of the action corresponding to the message from the next operation table (i.e. the table which address is the next element of the CURRENT stack). Thus an inflector intercepts some messages while delegating others to the next operation tables.

Delegation Interpreter Implementation

Implementation of the delegation interpreter ("interpretator soobsh'enij", message interpreter) requires two processor registers whose values will not be destroyed by other Forth routines such as NEST and NEXT.

The first version of the delegation interpreter was written in Forth rather than assembler and looks as follows:

```
\ CURP@ ( --> addr ) returns the address of CURRENT stack top.
3 QUANS R1 R2 R3
\ The values of R1 (delegation pointer) and R3 (message offset)
\ should not be destroyed by the Forth interpreter.
: CMSG ( n --> )    \ define a message word
    CREATE    CELLS ,
    DOES>       @ IS R3
              CURP@ IS R1
               R1 @ IS R2
          R1 R2 + @ RUSH ;

: PASS         CELL  AT R1 +!
                R1 @ IS R2
           R3 R2 + @ RUSH ;
```

If we define words DELEGATE and MP> :

```
: DELEGATE ( message_offset delegation_pointer --> )
             IS R1
             IS R3
       R1 @ IS R2
  R1 R2 + @ RUSH ;

\ Message Pointer --- leaves the values of message offset and
\ delegation pointer on the stack. Should be the first word in
\ a colon definition, otherwise these values may be destroyed!

: MP>      ( --> message_offset delegation_pointer )
           R3 R1 ;
```

we can define the above words as:

```
: CMSG ( n --> )    \ define a message word
    CREATE    CELLS ,
    DOES>     @ CURP@ DELEGATE ;

: PASS    MP> CELL+ RUSH> DELEGATE ;
```

Inflectors Implementation

When an inflector is created, all its methods are defined as PASS , except STRAIGHTEN that is defined as

```
: STRAIGHTEN R1 CELL+ CURP! ;
```

(or : STRAIGHTEN MP> CELL+ CURP! DROP ;)

An useful example of inflectors is GIVEN that causes the active operation table to take id from the data stack and not from the input stream or any other source:

```
INFLECTOR GIVEN
TO GIVEN
DEF ID NOOP
RECUR
```

If we had defined STRAIGHTEN for parent operation tables as

```
\ Discard anything above
: STRAIGHTEN MP> CUPR! DROP ;
```

we could implement weak inflectors. A weak inflector is an inflector that does not intercept the STRAIGHTEN message and therefore gets discarded. The word STRAIGHTEN would discard all the weak inflectors and the first non-weak one, if it occurs. To make an inflector weak one can simply define its STRAIGHTEN action as NOOP , althought there's a questionable reason for weak inflectors.

Search Order Control Wordset

The current implementation uses executable vocabulary stack (ORDER stack) that allows the use of vocabularies of diverse structures.

The executable vocabulary stack may contain objects of several types:

1. Vocabularies (to be more precise, their search code field addresses).
2. BOTTOM , the ORDER stack end mark.
3. LIMIT , the end-of-search mark.
4. MARK , the group end mark. The provided tools allow elimination of a group rather than of one element from the ORDER.

In searching, the vocabularies are searched, the group end marks are ignored, LIMIT and BOTTOM cause the end of search.

The word HEREAFTER adds the group end mark onto the ORDER stack. The word JUST adds the end-of-search mark onto the ORDER stack. This mark may be removed by means of the word THUS that scans the ORDER from the top and removes all the elements until it encounters a group end mark (that also gets removed). It is ill advised to use the word JUST in the interpretation mode because the system will cease to know any words.

The pseudo-vocabularies NUMBERS and VOC-NAMES recognize the numbers and the vocabularies. Both vocabularies and pseudo-vocabularies add themselves to the search order when executed (i.e. when their default code fields are executed).

VOCABULARY is a defining word for vocabularies and the word INIT> (or INHERITS> , they are aliases) allows to specify additional actions that are taken before adding the vocabulary to the search order.

For example, we may define

```
VOCABULARY ASMMACROS
INHERITS> FORTH HIDDEN ASSEMBLER ;
```

When we will execute ASMMACROS, the code from the INIT> part will be executed first, i.e. the vocabularies FORTH, HIDDEN and ASSEMBLER will add themselves to the search order and only then ASMMACROS will add itself to it.

The word ONLY is defined as

```
VOCABULARY ONLY
INIT>  JUST VOC-NAMES NUMBERS ;
```

and contains the most necessary words: HEREAFTER , JUST , ONLY , THUS , TO , AT , IS and some others.

Note that because of VOC-NAMES is coded in Forth, not in assembler, and works slowly, a name is first tried to be a number and only then it is tested on being a vocabulary. This results in that VOC-NAMES is rarely tried and does not slow down the search.

The word !!ORDER (reset-order) is defined as:

```
: !!ORDER ['] BOTTOM ORDER0 @ DUP OPDERP! ! ONLY ;
```

An example of the search order manipulations is given below. The marks that are present on the search order stack are not shown.

```
ONLY FORTH
\ The search order is: FORTH ONLY NUMBERS VOC-NAMES
   HEREAFTER HIDDEN EDITOR
   \ The search order is: EDITOR HIDDEN FORTH ONLY NUMBERS VOC-NAMES
      HEREAFTER ONLY FORTH ASSEMBLER
      \ The search order is: ASSEMBLER FORTH ONLY NUMBERS VOC-NAMES
      THUS
   \ The search order is: EDITOR HIDDEN FORTH ONLY NUMBERS VOC-NAMES
   THUS
\ The search order is: FORTH ONLY NUMBERS VOC-NAMES
THUS
\ The search order is the same as before the execution of the first
\ line of this example
```

WHAT IS DIFFERENT FROM OOP? FORTH, OOP AND ESOTERIC TEACHINGS

An object in Yang.

An environment (a context) is Yin.

Encapsulated Yin is Yang. Yang inside is Yin.

A set of objects forms an environment (sometimes).

This is Yin-Yang cycle.

Yin becomes Yang on the next layer of a system.

Many Yang constitute Yin on the next layer of a system.

OOP supports the environment ---> object layer transition.

COP supports the objects ---> environment layer transition.

From this point of view it is interesting to consider the process-oriented programming [7,8] that appears to work at the third level (an active object acts in an environment).

CONCLUSION

The benefits of this technique are:

flexibility;

code reuse;

late binding;

applicability of late binding methods to primitive objects that have no information about their structure (the words ID, and ROOM serve as good examples).

Given these tools, it is easy to implement an object-oriented system: we need just to associate data structures with their operation tables.

References

[1] Baranoff S.N., Nozdrunoff N.R.
Jazyk Fort i ego realizatsii. - L:Mashinostroenie, 1988. (The Forth Language and Its Implementations, in Russian.)

[2] Dahm M., "OOF, an Object Oriented Forth", 1991 FORML Conference Proceedings, p.338-352 , FIG, Oakland, USA, 1992.

[3] Craig Chambers, David Undar, "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented programming Language". Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, in SIGPLAN Notices, v.24 n.7, July 1989.

[4] Pliss O.A.
Ob'ektno-orientirovannaja sistema MEDIUM./Instrumentalnye sredstva podderzhki programmirovanija. - L:LIIAN, 1988. (MEDIUM: an Object-Oriented System, in Russian)

[5] Rosen, Evan. "High Speed, Low Memory Consumption Structures (QUAN and VECT)." Proc. of the 1982 FORML Conf., p191-196.

[6] Schleisiek, Klaus. Multiple Code Fields Data Types and Prefix Operators. The Journal of Forth Application and Research, Vol.1,No.2, Dec. 1983, pp.55-68.

[7] Tuzov V.A.
Funktsional'nye metody programmirovanija./ Instrumentalnye sredstva programmirovanija - L:LIIAN, 1988. - p.129-143. (The Functional Methods of Programmming [not functional programming; subsequently this technique was called the process-oriented programming], in Russian)

[8] Tuzov V.A.
Jazyki predstavlenija znanij. - L:LGU, 1990. (The Languages of Knowledge Representation [what an ideal language of knowledge representation should be], in Russian.)

[9] David Ungar, Randall B. Smith, "SELF: the Power of Simplicity", OOPSLA'87 Conference Proceedings, SIGPLAN Notices, v.22, n.12, December 1987.