A Formal Language Processor Implemented in Forth

Sergey N. Baranov

***Abstract*—**The structure of a Forth program is described which implements a language processor for an ALGOL-like programming language with its context-free component belonging to the class LL(1). It allows to check that a program in the given formal language is syntactically correct as well as to convert a correct program into a pseudo-code for a simple interpreter to interpret it and thus simulate the program behavior in a certain environment. The ultimate goal of this work is to build a tool for running experiments with programs in the Yard language which formally describes the behavior of multi-layer artificial neural networks on the principles of a machine with dynamic architecture (MDA) and due to that has a number of specific language constructs. The tool is assumed to run on a PC under MS Windows and is based on the system VFX Forth for Windows IA32 which implements the Forth standard Forth 200x of November 2014 (the so called Forth 2014).

***Keywords*—**formal languages, language processor, parser, regular expressions, Forth.

I. Introduction

AUTOMATED analysis of formal languages started in the 1960s. Various tools were developed for processing both context free and context dependent formal languages to be studied with computer machinery. To-day, data processing technologies are widely used in translation systems for a variety of computer devices with many applications to support them. E.g., the Flex/Bison parsers ([1], [2], [3]) are often used to quickly obtain a particular language processor from a formal definition of a language. The tool ANTLR [4] is the next step in developing language processors and is based on principles close to those of this paper. However, these mentioned tools constrain the type of the input formal grammar, and the employed algorithm often requires enormous memory for storing intermediate data. Moreover, their usage requires understanding their special language for representing a formal grammar of the considered programming language and code generation of the recognized program is out of scope of those useful tools.

This paper is aimed at developing a flexible and "pocket-like" inexpensive tool based on the current Forth 2014 standard [5] for experimenting with new formal languages. Usage of Forth provides the necessary flexibility and unlimited freedom [6] in designing the respective definitions while modern computing machinery eliminates a lot of memory and speed limitations of early 1960s. The tool should also allow for experiments with code generation for correct programs and simulation of their execution on some hardware platform (code generation and simulation will be the scope of future work based on the already obtained results).

Any ALGOL-like programming language may be considered as a set of chains composed from the lexical units of this language (the so called language lexems or terminals) according to the rules of the language grammar which splits into its context-free component and a number of context dependent rules or constraints [7]. To use the proposed technique, the context-free component of the language under consideration should be specified in the formalism of regular expressions [8], [9] built from the language terms (terminals, non-terminals, and expressions in them) using three classical operations: concatenation (sequencing), alternative choice (branching), and recursion (cycling). Initial terms are language lexems (terminals) directly recognizable in a program text, non-terminals denoting language constructs built from terminals, and an empty string denoting an absence of a lexem. Context dependencies are specified informally, they are checked through a mechanism of semantics – special procedures invoked by the language processor in the process of input text parsing and which exchange their data via a stack or global variables.

The text in the given programming language to be processed is contained in a simple text file considered as a sequence of symbols (characters) in the 8 bit ASCII coding. The following 4 classes of characters are distinguished:

1) 96 skip characters (space, tabulation and other "invisible" symbols);

2) 10 digits – characters which represent decimal digits , from "0" to "9";

3) 119 letters – letters of the Latin (52) and Russian (66) alphabets, both in the upper and lower case, as well as the underscore symbol "_";

4) 31 special characters – "!", """, "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";", "<", "=", ">", "?", "@", "[", "\", "]", "^", "`", "{", "|", "}", "~".

Thus a complete set of 96+10+119+31=256 8-bit character codes is obtained.

The text under processing may contain comments (see section IV) which are skipped and treated as a single space.

II. Sample Programming Language

Further narrative will be illustrated with examples in the OCC language – a subset of C for developing programs to run on a special kind of hardware [10].

There are a number of lexems (terminals) in any programming language, denoted in this paper with symbols in single quotes (apostrophes). Though their particular nomenclature may be different in different languages, one can always divide it in 3 groups:

S. N. Baranov for over 10 years was with Motorola ZAO, St. Petersburg, Russia. He is now with the St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, 14 linia 39, St. Petersburg, 199178, Russia (phone: 812-328-0887; fax: 812-328-4450; e-mail: SergeyBaranov@gmail.com).

1) lexems of the general mode (distinguished be angle brackets "<" and ">") in their denotation):

• '<finish>' – a special terminus lexem with no external representation;

• '<float>' – denotation of a floating point number in the decimal system in accorance with the ANSI/IEEE 754-1985 standard in the format: <d><d>*.<d>*[{E|e}[+|-]<d><d>*] – a number starts with a digit; there is always a period as a delimiter between the mandatory integer and a possible fraction in the significand; if present, the exponent begins with the Latin character "E" or "e", which may be followed by plus ("+") or minus ("-"), followed by at least one digit – examples are 3.14, 10.E-6, 123.456e3, 5.E+1;

• '<number>' – denotation of an integer in binary, octal, decimal (by default), or hexadecimal, the radix being denoted by letters "B" or "b" (binary), "C" or "c" (octal), "D" or "d" (decimal), and "H" or "h" (hexadecimal); in the latter case the first six letters of the Latin alphabet A..F in any case are used as digits after 0..9 (if the first significant digits is one of these letters, then it should be preceded by the digit 0) – examples are 101B, 111111B (binary), 777C, 100C (octal), 0, 125, 126D (decimal), 0ffH, 0AH, 0DH, 1000H (hexadecimal);

• '<operation>' – an operation sign composed by one or two special symbols: '-', '#', '*', '/', '/\', '/=', '\/', '^', '~', '+', '<', '<=', '=', '=<', '=>', '>', '>='; each of these 17 operations (except for '~') is dyadic, two of them ('-' and '+') are both dyadic and monadic; and '~' is monadic; their order of execution in compound expressions is determined by their priorities (all monadic operations have the highest priority 10):

Operation	Name	Priority		Operation	Name	Priority
'~'	Negation	10		'/='	Not equal	5
'^'	Power	9		'<'	Less than	5
'=<'	Left shift	8		'<='	Less than or equal	5
'=>'	Right shift	8		'='	Equal	5
'*'	Multiply	7		'>'	Greater than	5
'/'	Divide	7		'/\'	And	4
'-'	Subtract	6		'#'	Exclusive Or	3
'+'	Add	6		'\/'	Or	3
'>='	Greater than or equal	5				

• '<start>' – a special initial lexem with no external representation;

• '<string>' – a denotation of a character string in double quotes ("), the string length – the number of contained characters – may be from 0 (empty string) up to 80 (maximal length); a double quote itself as a string character is denoted by two double adjacent double quotes; examples are "abc", "" (empty string), "a""bc" (a quote inside the string);

• '<tag>' – an identifier; i.e., a sequence of letters (including underscore) and digits beginning with a letter. the total number of characterd not exceeding 80; examples are abcd, x1, _great;

2) lexems – reserved words: 'auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double', 'else', 'enum', 'extern', 'float', 'for', 'goto', 'if', 'int', 'long', 'register', 'return', 'short', 'signed', 'static', 'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while' – these lexems correspond to words contained in their denotations;

3) indicants denoted by one two adjacent special characters: '(', ')', ',', '.', ':=', ';', '[', ']', '{', '}' – brackets of three kinds (round, square, and curly), assignation sign composed by a colon and an equal sign, comma, period, and semicolon.

Each lexem, except for '<start>' и '<finish>', enjoy an external representation in the source program text; some lexems may have several representations (e.g., the lexem '<=' may be represented with an indicant "<=" and the word "le"), and some may have an unlimited number of representations (like '<number>', '<string>' or '<tag>').

Three lexems of the general type are characterized with additional parameters. The lexem '<float>' (denotation of a floating point number) and '<number>' (denotation of an integer) have the respective value as their parameters, while '<operation>' (operation) has two additional parameters – operation priority and its name (add, multiply, etc.)

The following non-terminals are distinguished in the language description to denote particular language structures: abstr_decl, abstr_decl1, compound, declaration, declarator, declarator1, enumm, expression, formula, init, operand, param, paramlist, pointer, program, specif, statement, struct_decl. The non-terminal named program is usually the initial non-terminal which any syntactically correct text is generated from. The following semantics whose names begin with "$" occur in the grammar rules to take into account non-formal context dependencies: $array1, $array2, $arrow, $aster, $brk, $call1, $call2, $call3, $case1, $case2, $char, $comp1, $comp2, $cond, $cond1, $cond2, $cont, $decl, $default, $do1, $do2, $dot, $else, $expr, $field, $finish, $for1, $for2, $for3, $for4, $for5, $goto, $ident, $if, $incr1, $incr2, $init, $label, $mem, $null, $number, $op1, $op2, $opcode1, $opcode2, $operand, $qual, $ret1, $ret2, $start, $stmnt, $string, $sw1, $sw2, $sw3, $tag, $then, $type, $wl1, $wl2. Each semantic usually denotes a certain procedure which is executes if in the process of input text parsing the current recognized lexem follows this semantic the in a grammar rule, the semantic enjoying access to all lexem parameters.

Grammar rules are formulated in the following way:

```
Non-terminal : Regular_expression .
```

Non-terminal being one of the non-terminal denotations enlisted above, and being specified in accordance with the following syntax of the Naur-Backus formalism:

Regular expression ::= {	
Empty | Lexem | Non-terminal | Semantic |	Basic elements (1)
Regular_expression Regular_expression |	Concatenation (2)
Regular_expression ';' Regular_expression |	Alternatives (3)
Regular_expression '*' Regular_expression |	Recursion (4)
'(' Regular_expression ')' }.	Expression in brackets (5)

Alternatives are enumerated in curly brackets separated with a vertical bar. In the section marked (1) basic elements of a regular expression are enumerated: `Empty` (an empty expression denoting absence of anything), `Lexem`, `Non-terminal`, and `Semantic`. Section (2) represents a concatenation which has no particular denotation, section (3) is for alternative choice with a semicolon as the operation sign, section (4) stands for recursion with an asterisk as its operation sign, and the last section (5) allows to embrace a regular expression in round brackets and thus to consider it as one operand in operations of concatenation, alternative choice, and recursion.

For better visibility of an alternative choice between a regular expression `A` and an empty alternative: `( A ; )` is denoted as `A` in square brackets: `[ A ]` (pronounced as "possible `A`") , as well as for a recursions with an empty left or right operand `*A` and `A*` are denoted as `*( A )` and `( A )*` respectively, while a recursion with both non-empty operands `A*B` is denoted as `( A )*( B )`.

With the above denotations a grammar of the OCC language may be specified through the following regular expressions:

```
abstr_decl : pointer [ abstr_decl1 ] ;
             abstr_decl1 .
abstr_decl1 : ( '(' abstr_decl ')' )*(
              '[' [ formula ] ']' ;
                 '(' [ paramlist ] ')' )  .
compound : $comp1 '{' *( declaration $decl )
       *( statement $stmnt ) $comp2 '}' .
declaration : ( specif )*  ( declarator
      [ $init ':=' init ] )*( ',' ) ';' .
declarator : [ pointer ] declarator1 .
declarator1 : ( $tag '<tag>' ;
         $tag '<label>' ; '(' declarator
        ')' )*( ( '[' [ formula ] ']' )* ;
          '(' (  ( $tag '<tag>' )*( ',' ) ;
                 paramlist ;
               ) ')'         ) .
enumm : '<tag>' [ ':=' expression ] .
expression : ( formula )*( ',' ) .
formula : ( *( $opcode1 '<operation>' )
    operand $operand )*(
    $opcode2 ':=' ; $opcode2 '<operation>' ) .
init : formula ; '{' ( formula ; '.' '.' '.'
               )*( ',' ) '}' .
operand : ( ( $incr1 '<increment>'
       $tag ( '<tag>' ; '<label>' ) ;
       $tag '<tag>' ( $incr2 '<increment>' ;
       ( $array1 '[' formula $array2 ']' )* ;
       $call1 '(' [ expression $call2 ]
              $call3 ')' ;  $ident ) ;
       $op1 '(' expression $op2 ')' )
   [ ( $dot '.' ; $arrow '->' ) $field (
      '<tag>' ; '<label>' ) ] ;
    $tag '<label>' ;
    $number '<number>' ;  $char '<char>' ;
    $string '<string>'
   )*( $cond1 '?' expression $cond2 ':' ) .
param : specif ( declarator ;
         '[' [ formula ] ']' ) .
paramlist : ( param ; '.' '.' '.' )*( ',' ) .
pointer : ( $aster '<operation>'
         *( $qual 'const' ;
            $qual 'volatile' ) )* .
program : $start '<start>'
  ( ( specif )*( ';' ;
     declarator [ compound ]
       ( ';' ;
     ':=' init *( ',' declarator
          [ ':=' init ] ) ';' ;
       ',' ( declarator [ ':=' init ]
         )*( ',' ) ';'
        *( declaration ) compound
            ) ) ;
      declarator *( declaration ) compound
    )* $finish '<finish>' .
specif : ( $mem 'auto' ; $mem 'register' ;
         $mem 'static' ;
         $mem 'extern' ; $mem 'typedef' ;
         $type 'void' ; $type 'char' ;
         $type 'short' ;
         $type 'int' ; $type 'long' ;
         $type 'float' ;
         $type 'double' ; $type 'signed' ;
         $type 'unsigned' ;
    ( 'struct' ; 'union' ) [ '<tag>' ] [ '{' (
          struct_decl )*
     '}' ] ; 'enum' ( '<tag>' [ '{' ( enumm
       )*( ',' ) '}' ] ;
      '{' ( enumm )*( ',' ) '}' ) ;
      $qual 'const' ; $qual 'volatile'  )* .
statement : *( $label '<label>' ':' ;
      $case1 'case' expression
      $case2 ':' ; $default 'default' ':' )
      ( compound ;
        'if' $cond '(' expression $then ')'
               statement
         [ $else 'else' statement ] $if ;
        'switch' $sw1 '(' expression $sw2 ')'
             statement $sw3 ;
        'while' $cond '(' expression $wl1 ')'
                  statement $wl2 ;
   $do1 'do' statement 'while' $cond '('
               expression ')' $do2 ;
        'for' $for1 '(' [ expression ]
              $for2 ';' [ expression ]
         $for3 ';' [ expression ] $for4 ')'
            statement $for5 ;( 'goto' $goto
'<tag>' ; $cont 'continue' ; $brk 'break' ;
  $ret1 'return' [ expression ] $ret2  ;
         expression $expr ; $null ) ';' ) .
struct_decl : specif ( declarator [ ':'
        expression ] )*( ',' ) ';' .
```

The initial non-terminal in this grammar is `program`.

This grammar representation is nothing more than a linear record of a syntactic graph of this language for recognizing correctly constructed chains composed from its lexems, and it may be considered as a text in Forth (that's why its elements are separated with spaces) which in its turn may represent a Forth program if the respective word definitions are provided. Therefore, the development of a language processor which recognizes this programming language consists in development of these word definitions.

III. Language Processor Structure

The language processor works as follows.

1. An instrumental Forth system compliant with the Forth 2014 standard (e.g., VFX Forth for Windows IA32 [11] or gFORTH for Windows [12]) is launched on a working PC under MS Windows, and the respective Forth text with word definitions is compiled.
2. After successful compilation of this Forth text which establishes the necessary context, another Forth text with the language grammar is compiled which thus is transformed into a parser program. Successful compilation of the grammar is terminated with the message "`Lexical Analyzer successfully compiled!`" from the instrumental Forth system.
3. Upon successful completion of the step 2, the source text in the programming language is submitted as input to the parser for analysis and upon successful completion of parsing a pseudo-code of the submitted program is built in an output file for subsequent execution. If a syntax error in the input text were found or any exception occurred (like file system error, buffer overflow etc.) then a respective error message is produces and the parser terminates processing.

The language processor has two major components: the lexical analyzer (scanner) and the syntactic analyzer (parser), each of them may be considered as a separate software product.

In the current version, the scanner consists of 62 Forth definitions with the total size of 368 LOCs (source lines of code in Forth, excluding empty lines and lines with comments only) and the parser has 73 definitions of the total size 431 LOCs; thus the total size of the scanner and parser is 799 LOCs including the OCC grammar of 135 LOCs. The grammar graph of the OCC language in its internal representation occupies 1391 cells (machine words).

For scanner and parser testing, the respective test wrappers and test suites were developed. Exceptions which occur when running these programs are processed through the Forth interruption mechanism of `CATCH-THROW`, where the word `CATCH` receives an address of the message string formed by the scanner or parser when the exception is recognized and passed by `THROW`. Exceptions recognized while compiling these two components terminate compilation through the word `ABORT"` with a respective error message.

IV. Lexical Analyzer

The lexical analyzer (scanner) converts the input source text into a sequence of numeric values denoting lexems (terminals) of the programming language, recognized in the input text in the order of their occurrences in this text. The scanner is implemented as two co-programs: the low-level `GetChar` and the upper-level `GetLex`. The former sequentially consumes and filters characters from the input text skipping all irrelevant characters and returning the next significant character upon a request, while the latter forms the next lexem from characters obtained at the low level. Both co-programs work with "looking ahead" at 1 element – a character in case of `GetChar` and a lexem in case of `GetLex`. Results of each invocation of these co-programs are returned in a respective pair of variables: `{CurrChar, NextChar}` in case of `GetChar`, and `{CurrLex, NextLex}` in case of `GetLex`. The first variable in the pair contains the value (character or lexem) returned upon the given request to respective co-program, and the second variable in the pair contains the value to be returned upon the next request addressed to this co-program.

A list of lexems is specified by the defining word `LexClasses` created to build definitions of all lexems in the word list `Lexems`. A lexem is represented with its execution token which never executed during compilation of the lexical analyzer, while during parsing this code checks whether the current value of the variable `CurrLex` is the execution token of this lexem and if this is the case, the current lexem is "accepted" and the parser proceeds to considering the next lexem from its input stream; otherwise, this lexem signals the parser about its failure to accept the current lexem.

Along with lexems, the scanner recognizes two kinds of comments in the input: a) from two adjacent slashes ("`//`") up to the end of line; and b) from a slash and an asterisk ("`/*`") up to an asterisk and a slash ("`*/`"), as is common in many C realizations, unless these two character combinations marking the beginning of a comment are not inside a string denotation. At the formal level, a comment is treated as a space character.

The initial value of the variable `NextChar` is a space, and that of the variable `NextLex` is the lexem `'<start>'`. Upon reaching the end of the input file, the co-program `GetChar` returns a special character `<eof>` with no external representation and denoting the end-of-file. From that moment this character is returned by `GetChar` in response to all further requests for the next character. When consumed by the co-program `GetLex` this character transforms in a special lexem `'<finish>'` returned by `GetLex` in response to all further requests for the next lexem.

A test wrapper for the scanner provides three kinds of testing: getting text lines from the input file, getting characters from the input file, and getting lexems. They are specified by test words `TestGetLine"`, `TestGetChar"`, and `TestGetLex"` which get the name of the input file from the input stream and subsequently extract lines, characters, and lexems from it until the input file is exhausted. The word `Test"` is an extension of `TestGetLex"` – it establishes an interrupt handler through `CATCH` and executes `TestGetLex"` in this context.

```
 : Test" ( "<chars>name<quote>"--)
   -1 ?Echo !
```

```
['] TestGetLex" CATCH ?DUP
IF
    CR ." Exception: " COUNT TYPE
    CR CloseInFile
THEN ;
```

Fig. 1 displays excerpts from a log of running GetLex in the test wrapper LA_TestWrapper which demonstrates the work of the scanner. Excerpts are separated by dotted lines.

```
include
c:\yard\LA_TestWrapper.fth
Including
c:\yard\LA_TestWrapper.fth
Including C:\yard\LA34.fth
ok
Test" c:\yard\words31.txt
Yard version:
C:\yard\LA34.fth
Test run on 30.06.2017 at
11:21:30
001 // Identifiers<eol>
002 abcd  x1 _great<eol>
'<START>'
'<TAG>'  repr=abcd
'<TAG>'  repr=x1
003 <eol>
004 // Integers<eol>
005 0 101B  111111B /*
Binary */<eol>
'<TAG>'  repr=_great
'<NUMBER>'  repr=0 value=0
'<NUMBER>'  repr=101B
value=5
006 777C 100C /* Octal
*/<eol>
'<NUMBER>'  repr=111111B
value=63
'<NUMBER>'  repr=777C
value=511
007 125 126D /* Decimal
*/<eol>
'<NUMBER>'  repr=100C
value=64
'<NUMBER>'  repr=125
value=125
008 0ffH 0AH 0DH 1000H /*
Hexadecimal */<eol>
'<NUMBER>'  repr=126D
value=126
...........................
```

```
..........................
014 static struct switch
typedef union<eol>
'SIGNED'  repr=signed
'STATIC'  repr=static
'STRUCT'  repr=struct
'SWITCH'  repr=switch
'TYPEDEF'  repr=typedef
015 unsigned void volatile
while <eol>
'UNION'  repr=union
'UNSIGNED'  repr=unsigned
..........................
020 ~ not ^ ** * / + - = < >
# xor /\ & \/ | or >= /= <=
=> =< shl shr le ge <eol>
'WHILE'  repr=while
'<OPERATION>'  repr=~ op=~
prio=10
...........................
025 <eol>
026 // Strings<eol>
027 "abc"<eol>
'--'  repr=-
028 "" /* Empty string
*/<eol>
'<STRING>'  repr=abc
length=3
029 """abc" "a""bc" "abc"""
/* Quote in various places
*/<eol>
'<STRING>'  repr= length=0
'<STRING>'  repr="abc
length=4
'<STRING>'  repr=a"bc
length=4
030 <eol><eof>
'<STRING>'  repr=abc"
length=4
'<FINISH>'
 ok
```

Fig. 1. A log of a scanner test run

Three digit numbers in the beginning of a line are the input text line numbers; the text lines are included in the log as they are read-in by the co-program GetChar. There are 30 such lines in this example. Each line terminates with the symbol <eol> which marks its end-of-line, while the symbol <eof> marks the end of the input file.

The log contains denotations of the recognized lexems followed be their external representation in the input file (after the key word "repr=") and additional parameters of this lexem if any with appropriate key words.

V. Syntactic Analyzer

The syntactic analyzer (parser) is built on-top of the lexical analyzer. Its main (starting) word is OCC" which obtains the name of the input file with the program text to be analyzed and checks whether this text complies with the grammar of the programming language OCC. As with the scanner, the test wrapper of the parser contains the word Test" which establishes an interrupt handler and initiates execution of the main parser word in this context.

The parser main word is created through the defining word Grammar. It starts grammar definition of the considered programming language in form of a series of generating rules for its non-terminals. The grammar ends with the closing word EndGrammar which identifies the initial non-terminal:

```
: Grammar    ( "<spaces>name"--123)
  CREATE  ALIGN HERE ( pfa)
    DUP GrammarGraph ! \ Grammar graph start
    ['] (CallNT) , 0 ,  HERE CELL+ CELL+ ,
    ['] (Success) , ['] (Fail) ,
.....................................
: EndGrammar ( "<spaces>name" addr 123 --)
  ( pfa) ' ( pfa xt-inital) >BODY @
  SWAP  CELL+ ! 0 ,
  CR ." Lexical Analyzer successfully
compiled!" ;
```

The Forth interpreter of the underlying Forth system ensures execution of the source grammar text as a text in Forth resulting in construction of a grammar graph for the given formal language which consists of elements of several kinds. The parser main word created by the defining word Grammar provides traversal of this graph controlled by the variable Pnt, pointing to its next element. The return stack Return with operations Push and Pop and the queue SemanticsQueue of semantics whose execution is delayed till accepting the current terminal, are used as auxiliary data structures. Executable codes are denoted with words in brackets. In total, 9 kinds of elements are provisioned for a grammar graph:

1. Jump at the address addr – 2 cells: $\boxed{\texttt{(Jump)|addr}}$
```
  : (Jump) Pnt @ @ Pnt ! ;
```
2. Call of a non-terminal at the address addr – 3 cells: $\boxed{\texttt{(CallNT)| addr |failaddr}}$
```
     : (CallNT) Pnt @ Return Push (Jump) ;
```
3. Starting a non-terminal xt – 2 cells: $\boxed{\texttt{(StartNT)| xt}}$
```
  : (StartNT) CELL Pnt +! ;
```
if zero is specified instead of xt then this is an auxiliary non-terminal created automatically with no name.
4. Successful completion of a non-terminal – 1 cell: $\boxed{\texttt{(ExitNT)}}$
```
  : (ExitNT) Return Pop CELL+ CELL+ Pnt ! ;
```
5. Unsuccessful completion of a non-terminal – 1 cell: $\boxed{\texttt{(FailNT)}}$
```
  : (FailNT) Return Pop CELL+ Pnt ! (Jump) ;
```
6. Passing a semantic xt – 2 cells: $\boxed{\texttt{(Semantic)| xt}}$
```
  : (Semantic) Pnt @
       SemanticsQueue Push  CELL Pnt +! ;
```
7. Passing a lexem xt – 3 cells: $\boxed{\texttt{(Lexem)|xt |failaddr}}$
```
  : (Lexem) CurrLex @ pnt @ @ =
   IF \ accept the current lexem:
     Pnt @ CELL+ CELL+ Pnt !
   ELSE \ reject the current lexem:
     CELL Pnt +!  (Jump) THEN ;
```
8. Successful completion of parsing – 1 cell: $\boxed{\texttt{(Success)}}$
```
 : (Success) ( --) CR ." Success!" 1 THROW ;
```
9. Unsuccessful completion of parsing – 1 cell: $\boxed{\texttt{(Fail)}}$
```
  : (Fail) ( --) CR ." Compilation Failed!"
 ........... \ Form a message in MessageBuf
              MessageBuf THROW ;
```

the scanner reports through `MessageBuf` which lexem was the current one and what other lexems were checked for it in form of the message: "`Lexem` <name> `is unexpected; possible options are:` <list of lexem names>".

The above list of 9 element kinds is complete for the following reasons. Elements `(Success)` and `(Fail)` are necessary because these are all possible outcomes of the parsing process (excluding its abnormal terminations through `ABORT` or exceptions). Elements `(CallNT)`, `(Semantic)`, and `(Lexem)` are inevitable as they match all grammar basic elements. Execution of a non-terminal may terminate either successfully or with a failure; therefore, two different exits `(ExitNT)` and `(FailNT)` should be provisioned as well. And finally, `(StartNT)` and `(Jump)` are needed to start a non-terminal body and to jump around it in a linear code of a grammar graph. Thus, totally 2+3+2+2=9 different kinds of elements are needed and this seems to be a sufficient minimum (as the number of Muses[1] is).

An initial value – the address of a five cell element "invoking the initial non-terminal"

```
(CallNT)| addr |failaddr|(Success)|(Fail)
```

of the given grammar is assigned to the variable `Pnt`, `addr` being the starting address of a series of elements for the initial non-terminal of the grammar, and `failaddr` being the address of the next cell but one which contains a reference to the code `(Fail)` while the previous cell contains a reference to the code `(Success)` (see items 8 and 9 above which occur in the grammar graph only once in this five cell element).

The parser provides an option to print-out the grammar graph with by the word `.DisplayCode` – see Fig. 2 below. Similar to Fig. 1, excerpts from the OCC grammar graph representing its beginning and end are separated by a dotted line. One can see that the whole graph occupies only 5564/4=1391 cells.

VI. Conclusion

The described implementation of a scanner and a parser in Forth turned out to be quite flexible and powerful. Its main part was borrowed from earlier author's development [10] and ported from Forth-83 to the Forth 2014 standard [5] with minor changes. However, it required reworking and redeveloping the grammar interpreter in order to avoid direct references to the internal structure of the definitions prohibited by Forth 2014. The tool runs on a PC under MS Windows and was developed using the system VFX Forth [11] which supports Forth 2014.

Another problem yet to be solved with the proposed analyzer is checking the input grammar for its correctness; i.e., that it really belongs to the class LL(1) and contains no undesirable recursions. This problem was successfully overcome in [13], so the tool may reuse the found solution.

[1] 'The thrice three Muses mourning for the death
Of Learning late deceas'd in beggary'
That is some satire, keen and critical. (W. Shakespeare, "A Midsummer Night's Dream", Act 5, Scene 1, 52-54).

```
GrammarGraph @ .DisplayCode
0000 (CALLNT)  2848 0016
0012 (SUCCESS)
0016 (FAIL)
0020 (STARTNT)  ABSTR_DECL
0028 (CALLNT)  2700 0060
0040 (CALLNT)  0080 0052
0052 (JUMP)  0072
0060 (CALLNT)  0080 0076
0072 (EXITNT)
0076 (FAILNT)
0080 (STARTNT)  ABSTR_DECL1
0088 (JUMP)  0148
0096 (STARTNT)  Noname
0104 (LEXEM)  '('  0144
0116 (CALLNT)  0020 0144
0128 (LEXEM)  ')'  0144
0140 (EXITNT)
0144 (FAILNT)
0148 (CALLNT)  0096 0264
0160 (LEXEM)  '['  0204
0172 (CALLNT)  1216 0184
0184 (LEXEM)  ']'  0204
0196 (JUMP)  0240
0204 (LEXEM)  '('  0260
0216 (CALLNT)  2560 0228
0228 (LEXEM)  ')'  0260
0240 (CALLNT)  0096 0260
0252 (JUMP)  0160
0260 (EXITNT)
0264 (FAILNT)
0268 (STARTNT)  COMPOUND
0276 (SEMANTIC)  $COMP1
0284 (LEXEM)  '{'  0376
0296 (CALLNT)  0380 0324
0308 (SEMANTIC)  $DECL
0316 (JUMP)  0296
0324 (CALLNT)  4416 0352
0336 (SEMANTIC)  $STMNT
0344 (JUMP)  0324
0352 (SEMANTIC)  $COMP2
0360 (LEXEM)  '}'  0376
0372 (EXITNT)
0376 (FAILNT)
0380 (STARTNT)  DECLARATION
0388 (JUMP)  0424
0324 (CALLNT)  4416 0352
0336 (SEMANTIC)  $STMNT
0344 (JUMP)  0324
0352 (SEMANTIC)  $COMP2
0360 (LEXEM)  '}'  0376
0372 (EXITNT)
0376 (FAILNT)
0380 (STARTNT)  DECLARATION
0388 (JUMP)  0424
0396 (STARTNT)  Noname
0404 (CALLNT)  3440 0420
0416 (EXITNT)
0420 (FAILNT)
0424 (CALLNT)  0396 0584
0436 (CALLNT)  0396 0456
0448 (JUMP)  0436
0456 (JUMP)  0524
0464 (STARTNT)  Noname
0472 (CALLNT)  0588 0520
0484 (SEMANTIC)  $INIT
0492 (LEXEM)  ':='  0516
0504 (CALLNT)  1384 0516
0516 (EXITNT)
0520 (FAILNT)
0524 (CALLNT)  0464 0584
0536 (LEXEM)  ','  0568
0548 (CALLNT)  0464 0568
............................
5424 (STARTNT)  STRUCT_DECL
5432 (CALLNT)  3440 5564
5444 (JUMP)  5504
5452 (STARTNT)  Noname
5460 (CALLNT)  0588 5500
5472 (LEXEM)  ':'  5496
5484 (CALLNT)  1120 5496
5496 (EXITNT)
5500 (FAILNT)
5504 (CALLNT)  5452 5564
5516 (LEXEM)  ','  5548
5528 (CALLNT)  5452 5548
5540 (JUMP)  5516
5548 (LEXEM)  ';'  5564
5560 (EXITNT)
5564 (FAILNT)   ok
```

Fig. 2. The beginning and the end of the OCC grammar graph

Future work will consist in developing a pseudo-code generator and an its interpreter to simulate execution of programs in the considered programming language.

References

[1] M. E. Lesk, E. Schmidt, "Lex – A Lexical Analyzer Generator", web: http://dinosaur.compilertools.net/lex/ (2017).

[2] "Win flex-bison", web: http://sourceforge.net/projects/winflexbison/ (2017).

[3] "GNU Bison", web: http://www.gnu.org/software/bison/ (2017).

[4] Terence Parr, "ANTLR (ANother Tool for Language Recognition)", web: http://www.antlr.org/ (2017).

[5] "Forth 200x", web: http://www.forth200x.org/forth200x.html (2016)

[6] L. Brodie, *Thinking Forth*. Punchy Pub, 2004.

[7] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley (1986).

[8] Jeffrey E.F. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2002.

[9] B. K. Martynenko, "Regular Languages and CF Grammars". In *Computer Tools in Education. 1, pp.14–20*, 2012. (In Russian).

[10] S. Baranov, Ch. Lavarenne, *Open C Compiler in Forth*. In: *EuroForth'95, 27-29 Oct. Schloss Dagstuhl*, 1995.

[11] "VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014". – Southampton: MPE Ltd, 2014. – 429 p., web: http://www.mpeforth.com/ (2014)

[12] "gForth", web: https://www.gnu.org/software/gforth/ (2017)

[13] S.N. Baranov, L.N. Fedorchenko, "Equivalent Transformations and Regularization in Context-Free Grammars" Cybernetics and Information Technologies. Bulgarian Academy of Sciences, Sofia. 2014. Volume 14, no 4, p.30-45. web: http://www.degruyter.com/view/j/cait.2014.14.issue-4/cait-2014-0003/cait-2014-0003.xml (2017)