PYTHON as Pseudo Language for Formal Language Theory

Zdravko DOVEDAN HAN, Kristina KOCIJAN, Vjera LOPINA

Department of Information and Communication Sciences, Faculty of Humanities and Social Sciences, University of Zagreb

Abstract
In this paper we use selected topics from formal language theory to show that programming language Python, with its syntax and semantics of primitive and structured statements (strings, tuples, lists and dictionaries) and functions, is suitable for use as a pseudo language and description of structures and algorithms in the formal language theory.

Keywords
formal language theory, pseudo language, pseudo code, Python, language defining, syntax analysis, translation.

I. INTRODUCTION

Formal language theory (FLT) appeared in the fifties of the twentieth century in the early works of Noam Chomsky. It was based on an appropriate natural language (e. g. English) model. Soon afterwards the use of context-free grammar and what is today called "Backus-Naur Form" (BNF) gave a major boost to the further development of FLT. It was used for the first time in the definition of ALGOL 60 language, in 1963.

Since then a strong feedback between formal language theory and programming languages was created: formal language theory found its application in defining programming languages and their translators, while programming languages have helped in the study of formal language theory. Many FLT books published from the end of the seventies to the present day have used Pascal and C for a description of certain structures and algorithms of formal language theory.

However, we argue that the right tool for use in FLT was obtained only by the emergence of Python language, which is now at the top of popularity, and combines all programming paradigms: structured, object-oriented, logical, dynamic and functional and which has a wide range of applications in many branches: mathematics, physics, chemistry, biology, natural language processing, databases, web applications, information technology, computer science, etc..

Using the examples of selected topics in formal language theory, this paper is structured as follows: In Section II. we introduce some basic definitions of formal language theory and show how the definition of grammar and generating of sentential form can be directly translated in Python. In Section III. we show how a pushdown recognizer for syntactic analysis of one simple language can be written in Python. We conclude by identifying some directions for future work.

II. LANGUAGE DEFINING

In order to define formal language, let's start from some basic definitions, [1] and[3].

Character is a unique (undivided or atomic) element. For example, upper- or lower-case Roman letters and digits are characters. *Alphabet* is a finite set of characters. *String* is a sequence of characters. *Empty string* is a string that has no symbols. It will be denoted by ε. The *length* of a string x, denoted $d(x)$ or $|x|$, is the number of characters in the string. The length of empty string is zero. If A is an alphabet, A^* denotes the set of all strings over A, including empty string ε. A^+ denotes a set of $A^*\setminus\{\varepsilon\}$. A *language* over an alphabet A is a set of strings over A, that is, $L \subseteq A^*$. It is often written $L(A)$ to show that some language L is defined over an alphabet A.

Sets of strings that make up elements of a language are called *sentences*. So, a language is a set of sentences.
Strings of finite-length that can be seen as a unique, undivided whole are often observed. Such strings are called *symbols* or *words*. Set of all symbols defined over an alphabet A will be marked with V and called *vocabulary*. Since $V \subseteq A^*$, we conclude that V is a language. For example, vocabulary

$$V = \{i,iv,v,ix,x,xl,l,xc,c,cd,d,cm,m\}$$

can be defined over the alphabet

$$A = \{i,v,x,l,c,d,m\}.$$

According to the Chomsky hierarchy (Chomsky, 1957) languages are classified into four groups (or types), as follows:

type	language
0	*unrestricted*
1	*context-sensitive*
2	*context-free*
3	*linear*

There are several methods for specifying the set that makes language. One method uses the formalism of *regular sets* and *regular expressions*. It is applicable only to the description of the type 3 language.

Second method uses a system called generative grammar. Every sentence of a language can be derived using grammar rules (called "productions").

Third method belongs to a class of *automata*. *Generators* are automata that can generate sentences of languages, but automata are more often used in the role of *recognizers* (in syntax language analysis).

Grammars

In general, a language is a finite set of sentences where sentences are of finite length. However, for many languages it is not possible to put an upper bound on the length of the longest sentence in that language and/or to the number of sentences. Except for that, it would be very unpractical to list all the sentences of a language even if a language consists of a finite number of sentences (for example 100 or 5000!). There are two principal methods of defining languages: by grammar and by automaton.

Grammar is a formalism to generate all four types of languages. A grammar is a 4-tuple, $G = (N,T,P,S)$, where:

- N is a finite set of *nonterminal symbols*,
- T is a finite set of *terminal symbols*, different from N,
- P is a finite set of pairs (α,β),where:
  $\alpha = \alpha_1\gamma\alpha_2$; $\alpha_1, \alpha_2, \beta \in (N \cup T)^*$, $\gamma \in N$

  An element (α,β) in P will be written $\alpha \rightarrow \beta$ and called *production.*
- S is a special symbol in N, $S \in N$, called *start symbol*.

If in some grammar, P contains following productions

$\alpha \rightarrow \beta_1 ... \alpha \rightarrow \beta_n$

this is written

$\alpha \rightarrow \beta_1 | \beta_2 | ... | \beta_n$.

The sign ' | ' is read 'or'. β_i are alternatives for α.

A grammar defines a language in a recursive manner. If $G = (N,T,P,S)$ is a grammar, *sentential form* of G is defined recursively as follows:

(1) S is a sentential form.

(2) If $\alpha\delta\gamma$ is a sentential form, where $\alpha,\gamma \in (N \cup T)^*$, and $\delta \rightarrow \beta$ is in P, then $\alpha\beta\gamma$ is also a sentential form.

A sentential form of G containing no nonterminal symbols is called a *sentence* generated by G.

Let $G = (N,T,P,S)$ be a grammar. It is defined as a relation $\Rightarrow$, to be read as *directly derives*, on $(N \cup T)^*$ as follows: If $\alpha\delta\gamma$ is a string in $(N \cup T)^*$ and $\delta \rightarrow \beta$ is a production in P, then $\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$.

If $\alpha_0, \alpha_1, ..., \alpha_n, \alpha_i \in (N \cup T)^*$, $n \geq 1$, such that

$\alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n$

then $\alpha_0 \overset{n}{\Rightarrow} \alpha_n$ is a *derivation of length n*. Generally, it is written

$\alpha_0 \overset{*}{\Rightarrow} \alpha_n$, $n \geq 0$, $\alpha_0 \overset{+}{\Rightarrow} \alpha_n$, $n > 0$

and said that α_0 *derives* α_n. Thus, a language L generated by a grammar G is:

$L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$

It is said that two grammars, G_1 and G_2, are *equivalent* if $L(G_1) = L(G_2)$.

Grammar classification

Grammars can be classified according to the format of their productions. If $G = (N,T,P,S)$ is a grammar, it is said that G is:

1) Right-linear or type 3 if each production in P is of the form

   $A \rightarrow xB$ or $A \rightarrow x$ $A,B \in N$, $x \in T^*$

   Left-linear if each production in P is of the form

   $A \rightarrow Bx$ or $A \rightarrow x$ $A,B \in N$, $x \in T^*$

   A right-linear grammar is called a regular grammar when

   (a) All productions, with the possible exception of $S \rightarrow \varepsilon$, are of the form $A \rightarrow aB$ or $A \rightarrow a$, where A, $B \in N$, $a \in T$.

   (b) If $S \rightarrow \varepsilon$ is in P, then S does not appear on the right side of any production.

2) Context-free or type 2 if each production in P is of the form

   $A \rightarrow \alpha$ $A \in N$, $\alpha \in (N \cup T)^*$

3) Context-sensitive or type 1 if each production in P is of the form $\alpha \rightarrow \beta$
   where $|\alpha| \leq |\beta|$.

4) Unrestricted or type 0 if there are no restrictions as the ones above.

Grammar implementation in Python

The grammar $G = (N,T,P,S)$ can be directly translated in the Python 4-tuple in which the elements N and T can be implemented as lists, element P as dict and start symbol S as symbol from the list N. Initially we define grammar as text writing only the production in the form of:

```
alfa -> beta
```

Nonterminals are upper-case letters, `#` is empty string and symbol `->` is production sign. Here are two examples of productions, context-free grammar of the Roman numerals language and context-sensitive grammar of language $\{a^n b^n c^n d^n, n>0\}$:

```
Roman = """
R -> MA|CB|XD|I
M -> m|mm| mmm
A -> #|CB| XD I
C -> c|cc|ccc|cd|d|dc|dcc|dccc|cm
B -> #|XD|I
X -> x|xx|xxx|xl|l|lx|lxx|lxxx|xc
D -> #|I
I -> i|ii|iii|iv|v|vi|vii|viii|ix
"""
abcd = """
S  -> aBCSd| abcd
Ba -> aB
Bb -> bb
Ca -> aC
Cb -> bC
Cc -> cc """
```

Procedure GRM() from the productions of grammar, given as a text (or text file), returns the definition of grammar $G=(N,T,P,S)$ that can be displayed by calling the printing procedure, Write_GRM():

```
from random import *
def GRM (X) : # Grammar definition
  N = T = ''
  A = (X.replace (' ', '')).split('\n')
  S = A[1][0];  Y = { 'alfa' : [], 'start' : S}
  for a in A :
     if not a : continue
     [x, y] = a.split('->')
     y      = tuple (y.split('|'))
     Y[x], Y['alfa'] = y, Y['alfa'] +[x]
     for c in y :
       T += c *(not c.isupper() and c not in T)
     for c in x :
       N += c *(c.isupper() and c not in N)
  return list(N), list(T), Y, S
def Write_GRM (G) :
  N, T, P, S = G
  print P['name'], '= (N, T, P, S)'
  print 'N = { ' +('%s, '*(len(N)-1) \
              % tuple(N[:-1])) +N[-1] +' }'
  print 'T = { ' +('%s, '*(len(T)-1) \
              % tuple(T[:-1])) +T[-1] +' }'
  print 'S =', P['start']
  print 'P :'
  for x in P['alfa'] :
      print x, '->', P[x][0],
      for y in P[x][1:] : print '|', y,
      print
  print
```

For example, the grammar of the Roman numerals language is:

```
>>> G = (N, T, P, S) = GRM (Roman); Write_GRM (G)
G = ( N, T, P, S )
N = { R, M, A, C, B, X, D, I }
T = { m, #, c, d, x, l, i, v }
S = R
P :
R -> MA | CB | XD | I
M -> m | mm | mmm
A -> # | CB | XD | I
C -> c | cc | ccc | cd | d | dc | dcc | dccc | cm
B -> # | XD | I
X -> x | xx | xxx | xl | l | lx | lxx | lxxx | xc
D -> # | I
I -> i | ii | iii | iv | v | vi | vii | viii | ix
```

and grammar of language $\{a^n b^n c^n d^n, n>0\}$ is:

```
>>> G = (N, T, P, S) = GRM (abcd);  Write_GRM (G)
G = ( N, T, P, S )
N = { S, B, C }
T = { a, d, b, c }
S = S
P :
S -> aBCSd | abcd
Ba -> aB
Bb -> bb
Ca -> aC
Cb -> bC
Cc -> cc
```

Procedure DER(P) generates sentences of the language defined by productions P of grammar G. If called by DER(P,True), a sequence of sentential forms (SF) will be displayed.

```
def DER (P, DSP = False) :
  SF = P['start'];  print SF,
  if not DSP : print '*=>',
  while True :
    for a in P['alfa'] :
      if a in SF :
        x  = (''.join(sample (P[a], 1)))
        i  = SF.find(a);  x *= (x != '#')
        SF = SF[:i] +x +SF[i+len(a):]
        if DSP : print '=>', SF,
        break
    else :
      print SF if not DSP else ''; return SF
```

```
>>> G = (N, T, P, S) = GRM (Roman)
>>> for i in range(3) :
       sf = DER (P, True);  print

R => CB => cB => cXD => cxD => cxI => cxiii
R => XD => xxxD => xxxI => xxxi
R => MA => mA => mXD => mxcD => mxcI => mxciii

>>> G = (N, T, P, S) = GRM (abcd)
>>> for i in range(3) :
       sf = DER (P, True);  print

S => aBCSd => aBCabcdd => aBaCbcdd => aaBCbcdd
=> aaBbCcdd => aabbCcdd => aabbccdd
```

III. SYNTACTIC ANALYSIS

In practice we often encounter the problem that a grammar or a generator of the language is known and a character string is given, and the question is asked whether this is a sentence of the language generated by a given grammar or generator. This process is called *syntactic analysis*.

If the language is defined by a grammar, problem reduces to finding a sequence of derivations (sentential forms), starting from S, which would result in this string (sentence). Such a procedure of syntax analysis is called *parsing*. Parsing process structure on the computer (the program in some selected programming language) is called *parser*, [4].

If the language is defined by an automaton, we ask the question: can a given string be generated by a given generator? Then the automaton is in the role of language recognizer, which analyzes the input string, and after a finate number of changes in their configuration, starting from an initial state reaches a final state if the string is in the language and answers "yes", or the process interrupts and answers "no" if the input string is not in the language. Such a syntax analysis procedure is called *recognizing*, and an automaton that does it is called *recognizer*, [4].

Recognizing

Third language defining method is by an *automaton*. It is a device which consists of a combination of the following parts: an input tape with an input head (reader), an output tape with an output head (writer), an auxiliary memory, and a finite set of rules which controls or regulates the information flow.

Depending on the type of language that automaton recognizes, there are following types of recognizers:

Name	*Definition*	*Language*
final	$M = (Q,\Sigma,\delta,q_0,F)$	type 3
pushdown	$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \mathcal{F})$	type 2
double- pushdown	$Pt = (Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, \mathcal{F})$	type 1
Turing machine	$Tg = (Q, \Sigma, \Gamma, \delta, q_0, F)$	type 0

where:

Q	final set of states
$P(Q)$	power set of Q
Σ	alphabet
Γ	alphabet of stack
Γ_1, Γ_2	alphabet of first and second stack
Z_0	the initial character of stack, $Z_0 \in \Gamma$
δ	*transition function,*
q_0	*the initial state,* $q_0 \in Q$
F	*set of final states,* $F \subseteq Q$

Depending on the type of language, transition function is defined as:

δ: $Q \times \Sigma$	$\rightarrow P(Q)$	type 3
δ: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$	$\rightarrow Q \times \Gamma^*$	type 2
δ: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma_1 \times \Gamma_2$	$\rightarrow Q \times \Gamma_1^* \times \Gamma_2^*$	type 1
δ: $Q \times \Sigma \cup \{\#\}$	$\rightarrow Q \times \{\Gamma \setminus \{\#\}\} \times \{-1,0,1\}$	type 0

To show the syntax analysis of context-free languages using pushdown recognizer P, first we introduce the definitions, [1]:

1) A *configuration* of P is a tuple (q, w, α) from $Q \times \Sigma^* \times \Gamma^*$, where:

q	current state
w	remaining portion of the input
α	content of the pushdown list; the leftmost symbol of α is the topmost pushdown symbol.

2) An *inital configuration* of P is (q_0, w, Z_0),
3) A *final configuration* of P is (q, ε, α), $q \in F$, $\alpha \in \Gamma^*$,
4) A *move* by P is binary relation $\vdash$. We write

   $(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$

   if $\delta(q,a,Z)$ contains (q',γ) for any $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $Z \in \Gamma$. We say that a input string w is *accepted* by P if

   $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$

The language defined by P, denoted $L(P)$, is set of strings w accepted by P. It is generally a context-free language:

$$L(P) = \{w: \ w \in \Sigma^* \wedge (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha), \ q \in F, \ \alpha \in \Gamma^*\}$$

Python's *dict* is the most appropriate structure for the implementation of the transition function because it represents its copy. If D is the transition function of any type of recognizer, its elements will generally have the structure:

$$D = \{ x_0 : y_0, \ x_1 : y_1, \ \ldots, \ x_n : y_n \}$$

where x_i is domain tuple and y_i is codomain whose structure is dependent on automaton type. For example, pushdown recognizer of language `Exp` generated by a grammar:

```
E -> E+E| E*E| (E)| a| b
```

written in Python is given below:

```
# -PUSHDOWN RECOGNIZER
Exp = """
Q  = [0, 1];  A = ['a', 'b', '+', '*', '(', ')']
St = ['$', '('];  _1 = '$';  s = 0;  F = [1]
D  = {
  (0,'a','$'): (1, '$'),  (0,'a','('): (1, '('),
  (0,'b','$'): (1, '$'),  (0,'b','('): (1, '('),
  (0,'(','$'): (0, '($'), (0,'(','('): (0, '(('),
  (1,'+','$'): (0, '$'),  (1,'*','$'): (0, '$'),
  (1,'+','('): (0, '('),  (1,'*','('): (0, '('),
  (1,')','('): (1, ''),   (1,'','$') : (1,'')  }
Name = 'Exp'
DSP  = (Q, A, St, _1, D, s, F) """
NL = '\n'
def Input_W ():
  return (raw_input ('Enter input string: ')).\
         replace (' ', '')

def Write_SP (Name): # pushdown recognizer
  print Name
  print NL, 'SP = (Q, A, St, _1, D, s, F)', NL
  print 'Q  =', Q, NL, ' A =', A, NL, \
        'St =', St, NL, '_1 =', _1, NL, \
        ' s =', s, NL, ' F =', F
  print NL, 'D:'
  S = D.keys();  S.sort()
  for d in S: print ' ', d, '=', D[d]
  print

def Write_C (y, C): print y, C

def SP (x):
  global Q, A, St, _1, D, s, F

  Ok = True; End = False
  q = s; alfa = '$'
  C = (q, x, alfa);
  Write_C ('', C)
  while len(x)>=0 and Ok and not End:
    X = ''; a = ''
    if len(x) > 0    : X = x[0]; x = x[1:]
    if len(alfa) > 0 : a = alfa[0]
    Ok = False
    d = (q, X, a)
    if d in D :
      q, g = D[d]
      if g == '' and a != '': alfa = alfa[1:]
      if g != ''            : alfa = g +alfa[1:]
      Ok = True
    else : Ok = False

    if Ok:
      C = (q, x, alfa);
      Write_C ('   |--', C)
      if q in F and alfa == '' : End = True
      if End and x != ''       : Ok  = False

  Ok = Ok and End
  return Ok

exec Exp;  Write_SP (Name)

w = Input_W(); print
while len(w) > 0:
  Ok = SP (w)
  if Ok: Write_C ('   |--', 'accept')
  else : Write_C ('   |--', 'error')
  print
  w = Input_W(); print
```

```
>>>
Enter input string: a*(a+b)
 (0, 'a*(a+b)', '$')
   |-- (1, '*(a+b)', '$')
   |-- (0, '(a+b)', '$')
   |-- (0, 'a+b)', '($')
   |-- (1, '+b)', '($')
   |-- (0, 'b)', '($')
   |-- (1, ')', '($')
   |-- (1, '', '$')
   |-- (1, '', '')
   |-- accept
```

IV. TRANSLATION

If Σ is input alphabet and Δ is output alphabet, *translation* from the language L_1, $L_1 \subseteq \Sigma^*$, to the language L_2, $L_2 \subseteq \Delta^*$, is a relation T from $\Sigma^* \times \Delta^*$ so that L_1 is the domain and L_2 is the codomain of T. The sentence y, such that (x, y) is in T, is called *the output* of x.

Syntax-directed translation

One of formalisms for defining translation is *the syntax-directed translation scheme*. Intuitively, syntax-directed translation scheme is simply a grammar in which translation elements are related to each production.

Whenever a production was used in the derivation of an input sentence, the translation element is used to help compute a portion of the output sentence related to the portion of the input sentence generated by that production. *Translational form* of *T* is defined as follows:

1) (S, S) is a translational form and the first S is related to the second S.
2) If $(\alpha A\beta, \alpha' A\beta')$ is a translational form and if $A \to \gamma, \gamma'$ is a rule in R, then $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ is new translational form. Nonterminals from γ and γ' are exactly related, same as in the rule. Nonterminals from α and β are related to such nonterminals from α' and β' in the new translational form exactly as in the old. We write:

   $(\alpha A\beta, \alpha' A\beta') \rightarrow (\alpha\gamma\beta, \alpha'\gamma'\beta')$

   and read (αAβ, α'Aβ ') "directly derives" ($\alpha\gamma\beta$, α'γ'β'). Similarly to the derivation of sentential form, a series of deriving *k* translational form, where k≥0, will be denoted by $*\rightarrow$, so the translation defined by *T*, denoted as τ (T), is a set of pairs:

   $\tau(T) = \{ (x, y) \mid (S, S) *\rightarrow (x, y), x \in \Sigma^*, y \in \Delta^* \}$

The implementation of SDT in Python is given in the procedure SDT ():

```
def SDT (X) : # Syntax-directed translation
              grammars
  A = (X.replace (' ', '')).split('\n')
  Y = {'start' : A[1][0]}
  for a in A :
     if not a : continue
     b = a.split('->'); N = b[0]
     b = b[1].split(',');
     Y [N] = (tuple (b[0].split('|')),
              tuple (b[1].split('|')))
  return Y
```

where X is an input-output grammar of languages to be translated, with productions of form

```
A -> I, O
```

where *I* are the alternatives for input and *O* for output language. For example, for translating Roman into Arabic numerals, a grammar RA can be defined:

```
#    Input grammar,  Output grammar
RA = """
R -> MA| CB| XD| I,  MA| CB| XD| I
M -> m| mm| mmm,     1| 2| 3
A -> #| CB| XD| I,   000| CB| 0XD| 00I
C -> c| cc| ccc| cd| d| dc| dcc| dccc| cm, \
     1| 2| 3| 4| 5| 6| 7| 8| 9
B -> #| XD| I,       00| XD| 0I
X -> x| xx| xxx| xl| l| lx| lxx| lxxx| xc, \
     1| 2| 3| 4| 5| 6| 7| 8| 9
D -> #| I,           0| I
I -> i| ii| iii| iv| v| vi| vii| viii| ix, \
     1| 2| 3| 4| 5| 6| 7| 8| 9 """
```

To translate the input string, the Roman numeral, into Arabic, it is necessary to derive syntax analysis tree (sequence of derivations) by some parsing procedure. It is left out here, so let's show the scheme of translating Roman numerals into Arabic using the examples of three generated sentential forms:

```
T = SDT (RA);  frm  = "(%s, %s)"
for i in range (3) :
  x = y = T['start']
  print frm % (x, y),
  while not x.islower() :
    for s in x :
      if s.isupper() :
        a, b = T[s]
        z = ''.join(sample (a, 1))
        i = a.index(z);  z = z *(z != '#')
        x = x.replace (s, z)
        a = b[i];  y = y.replace (s, a)
        print '\t-->', frm % (x, y)
        break
  print
```

```
>>>
(R, R) --> (XD, XD)        (R, R) --> (CB, CB)
       --> (xD, 1D)               --> (dB, 5B)
       --> (x, 10)                --> (dXD, 5XD)
(R, R) --> (MA, MA)               --> (dxcD, 59D)
       --> (mmA, 2A)              --> (dxc, 590)
       --> (mmI, 200I)
       --> (mmvi, 2006)
```

Finite transducer

Finite transducer is defined as a 6-tuple $M = (Q,\Sigma,\Delta,\delta,q_0,F)$. It is a finite automaton where Δ is *output alphabet* and δ is a mapping from $Q\times(\Sigma\cup\{\varepsilon\})$ to finite subsets of $Q\times\Delta^*$. A *configuration* of finite transcudes M is a tuple (q, x, y), where

- $q \in Q$ is the current state,
- $x \in \Sigma^*$ is the input string remaining on the input tape, with the leftmost symbol of x under the input head,
- $y \in \Delta^*$ is the output string emitted up to this point.

The *initial configuration* is (q_0, x, ε) and the *final configuration* is (q, ε, y), $q \in F$, $y \in \Delta^*$. A *move* by M is the binary relation $\vdash$. We write

$(q, ax, y) \vdash (r, x, yz)$

if $\delta(q,a)$ contains (r,z) for any $q \in Q$, $a \in \Sigma\cup\{\varepsilon\}$, $z \in \Delta^*$. We say that y is an *output* for x if

$(q_0, x, \varepsilon) \vdash^* (q, \varepsilon, y)$

The *translation defined by M*, denoted $\tau(M)$, is

$\tau(M) = \{(x,y) \mid (q_0, x, \varepsilon) \vdash^* (q,\varepsilon,y),\ x \in \Sigma^*,\ y \in \Delta^*\}$

Here is an example of transducing Roman numerals into Arabic. D is transition function with structure of elements

```
q : (Tm, Td, Tc, Tl, Tx, Tv, Ti, Tf)
```

where q is state, T*a*=D[q]['mdclxvi$'.find(*a*)].

```
# TRANSDUCER OF ROMAN NUMERALS INTO ARABIC
# Transition function ("table")
e = '';
# q    m      d      c      l       x
#         v        i        $
D = { 'q0' : 0, 'tr' : 'mdclxvi$',
       'Q' : range (31), 'F' : range (1, 31),
  0: ((1,e), (8,e), (4,e), (17, e),(13, e),
         (26,e),  (22, e),  e      ),
  1: ((2,e), (8,1), (4,1), (17,10),(13,10),
         (26,100),(22,100),(e,1000)),    # m
  2: ((3,e), (8,2), (4,2), (17,20),(13,20),
         (26,200),(22,200),(e,2000)),    # mm
```

```
  3: (    e, (8,3), (4,3), (17,30),(13,30),
         (26,300),(22,300),(e,3000)),    # mmm
  4: ((12,e),(7,e), (5,e), (17, 1),(13, 1),
         (26, 10),(22, 10),(e, 100)),    # c
  5: (    e,     e, (6,e), (17, 2),(13, 2),
         (26, 20),(22, 20),(e, 200)),    # cc
  6: (    e,     e,     e, (17, 3),(13, 3),
         (26, 30),(22, 30),(e, 300)),    # ccc
  7: (    e,     e,     e, (17, 4),(13, 4),
         (26, 40),(22, 40),(e, 400)),    # cd
  8: (    e,     e, (9,e), (17, 5),(13, 5),
         (26, 50),(22, 50),(e, 500)),    # d
  9: (    e,     e,(10,e), (17, 6),(13, 6),
         (26, 60),(22, 60),(e, 600)),    # dc
 10: (    e,     e,(11,e), (17, 7),(13, 7),
         (26, 70),(22, 70),(e, 700)),    # dcc
 11: (    e,     e,     e, (17, 8),(13, 8),
         (26, 80),(22, 80),(e, 800)),    # dccc
 12: (    e,     e,     e, (17, 9),(13, 9),
         (26, 90),(22, 90),(e, 900)),    # cm
 13: (    e,     e,(21,e), (16, e),(14, e),
         (26,  1),(22,  1),(e,  10)),    # x
 14: (    e,     e,     e,      (),(15, e),
         (26,  2),(22,  2),(e,  20)),    # xx
 15: (    e,     e,     e,      (),     (),
         (26,  3),(22,  3),(e,  30)),    # xxx
 16: (    e,     e,     e,      (),     (),
         (26,  4),(22,  4),(e,  40)),    # xl
 17: (    e,     e,     e,      (),(18, e),
         (26,  5),(22,  5),(e,  50)),    # l
 18: (    e,     e,     e,      (),(19, e),
         (26,  6),(22,  6),(e,  60)),    # lx
 19: (    e,     e,     e,      (),(20, e),
         (26,  7),(22,  7),(e,  70)),    # lxx
 20: (    e,     e,     e,      (),     (),
         (26,  8),(22,  8),(e,  80)),    # lxxx
 21: (    e,     e,     e,      (),     (),
         (26,  9),(22,  9),(e,  90)),    # xc
 22: (    e,     e,     e,      (),(30, e),
         (25,  e),(23,  e),(e,   1)),    # i
 23: (    e,     e,     e,       e,      e,
          e,(24,  e),(e,   2)),          # ii
 24: (    e,     e,     e,       e,      e,
          e,       e,(e,   3)),          # iii
 25: (    e,     e,     e,       e,      e,
          e,       e,(e,   4)),          # iv
 26: (    e,     e,     e,       e,      e,
          e,(27,  e),(e,   5)),          # v
 27: (    e,     e,     e,       e,      e,
          e,(28,  e),(e,   6)),          # vi
 28: (    e,     e,     e,       e,      e,
          e,(29,  e),(e,   7)),          # vii
 29: (    e,     e,     e,       e,      e,
          e,       e,(e,   8)),          # viii
 30: (    e,     e,     e,       e,      e,
          e,       e,(e,   9)) }         # ix
```

```
p = lambda x : x if str(x) else '#'
TR = D['tr']
def FT (w, a=1):
  A = a;  w += '$';  q = 0;  i = 1;  y = ''
  C = (q, p(w), p(y));  print "(%2s,%s,%s)" % C
  while 'Ok' :
    a = w[0]; j = TR.find(a)
    if j >= 0:
      Q = D[q][j]
      if Q != () :
        q, z = Q; y += str(z); w = w[1:]
        C = (p(q), p(w), p(y))
        print " |-- (%2s,%s,%s)" % C
        if not w : return True
      else : print 'syntax error';  return False
    else:
      print 'illegal character';  return False
w  = raw_input ("input Roman ").lower()
Ok = FT (w)
```

```
>>>
input Roman X
( 0,x$,#)
 |-- (13 $,#)
 |-- ( #,#,10)
```

```
>>>
input Roman mmvi
( 0,mmvi$,#)
 |-- ( 1,mvi$,#)
 |-- ( 2,vi$,#)
 |-- (26,i$,200)
 |-- (27,$,200)
 |-- ( #,#,2006)
```

```
>>>
input Roman dxc
( 0,dxc$,#)
 |-- ( 8,xc$,#)
 |-- (13,c$,5)
 |-- (21,$,5)
 |-- ( #,#,590)
```

V. CONCLUSION

In this paper we demonstrated how Python programming language can be employed as a pseudo-language using the example of selected structures and algorithms of formal language theory for their description and implementation. In [3], [4] and [5] there are many examples of applying Python for implementing procedures of the context-free languages syntax analysis and for recognizing the languages of all types.

Python is also suitable for use in creating the interpreters and the preprocessors, both as an encoding language and as a target language. Examples of such applications are given in [5] where it is implemented as an interpreter for language PL/0, according to [8], and preprocessor of mini language defined in [2].

It has been shown that Python is suitable for use in natural language processing (NLP). In [6] and [7] Python has been successfully applied in the implementation of the lexical and syntax analysis of English language and its translation in a special form.

Finally, we can conclude with satisfaction that the implementation of Python can significantly improve the study of the formal language theory and its applications.

References

[1] AHO, V. A.; ULLMAN, D. J.: *The Theory of Parsing, Translation, and Compiling*, vol. I: *Parsing*, Prentice-Hall, 1972.

[2] DIJKSTRA, E.W.: *A Discipline of Programming*, Prentice-Hall, 1976.

[3] DOVEDAN HAN, Z.: *FORMALNI J EZICI I PREVODIOCI*
- *regularni izrazi, gramatike, automati*, Element, Zagreb, 2012.

[4] DOVEDAN HAN, Z.: *FORMALNI JEZICI I PREVODIOCI*
- *sintaksna analiza i primjene*, Element, Zagreb, 2012.

[5] DOVEDAN HAN, Z.: *FORMALNI JEZICI I PREVODIOCI*
- *prevođenje i primjene*, Element, Zagreb, 2013.

[6] JAKUPOVIĆ, A., PAVLIĆ, M., & DOVEDAN, H. Z.: *Formalisation method for the text expressed knowledge*. *Expert systems with applications*. 41 (11). 5308-5322, 2014.

[7] PAVLIĆ, M., & DOVEDAN, H. Z., JAKUPOVIĆ, A.: *Question answering with a conceptual framework for knowledge-based system development ''Node of Knowledge''*. Expert systems with applications. 42 (2015) 5264–5286, 2015.

[8] WIRTH, N.: *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.