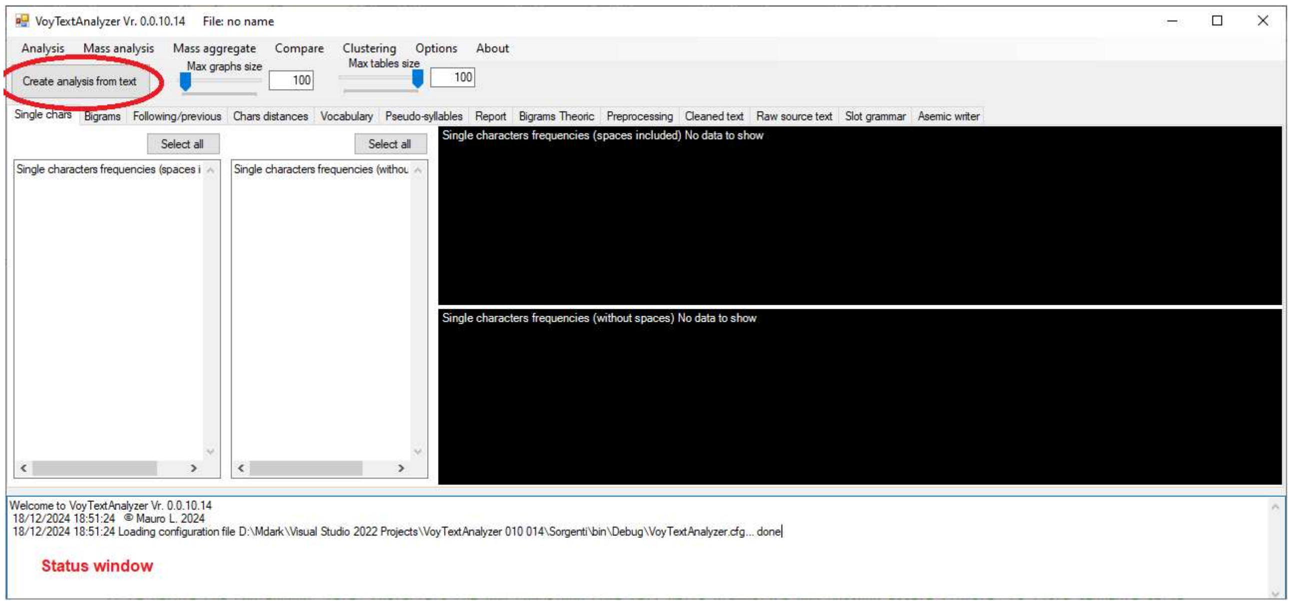
VoyTextAnalyzer adds the slot grammars functions to a previous program, TextAnalyzer, which is unpublished and was meant as a statistical tool for texts in any language[1]. All the original statistical functions of TextAnalyzer are available in VoyTextAnalyzer but are not described here. A manual is available: “Manuale Text Analyzer”, unfortunately only in Italian at the moment.

Installation

No installer is provided. If you are not interested in the source files just copy VoyTextAnalyzer.exe in any directory and run it by double-clicking on it. The first time VoyTextAnlyzer is run it will create a configuraton file in the same directory (it’s just an XML file with .cfg extension).

Use

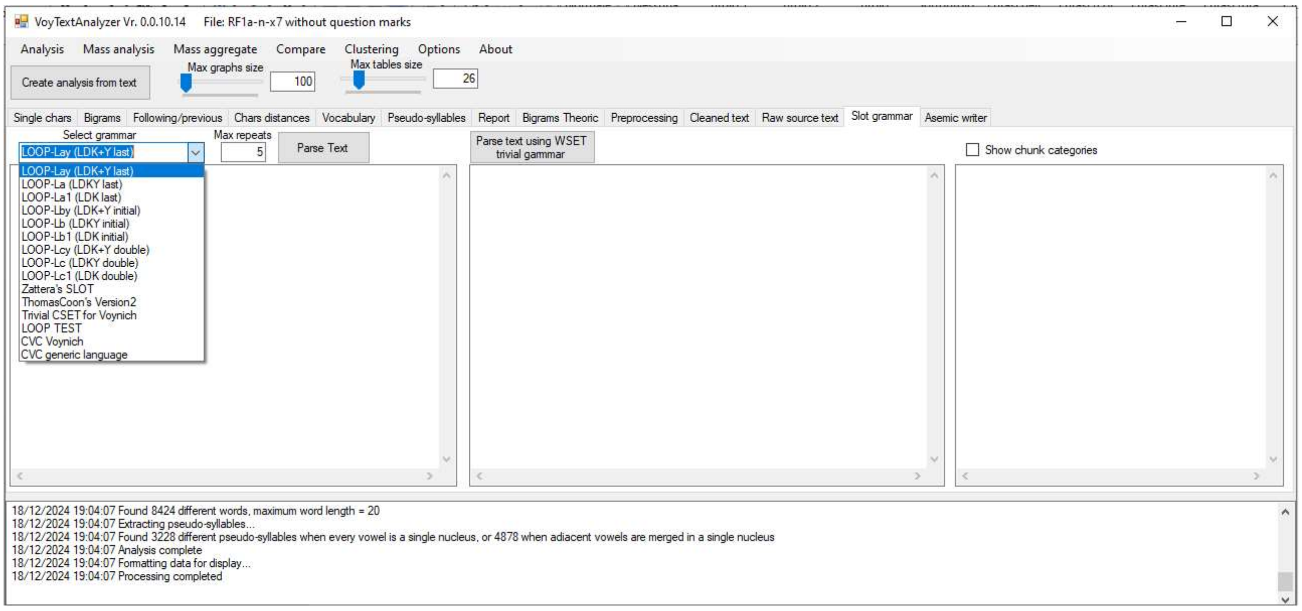


```
Welcome to VoyTextAnalyzer Vr. 0.0.10.14
18/12/2024 18:51:24  © Mauro L. 2024
18/12/2024 18:51:24 Loading configuration file D:\Mdark\Visual Studio 2022 Projects\VoyTextAnalyzer 010 014\Sorgenti\bin\Debug\VoyTextAnalyzer.cfg... done
```

Important: before you can use slot grammars, you must first load and ‘analyze’ the source text. Use the ‘Create analysis from text’ button to open a file dialog and load the desired source text file. The file must be in .txt format. VoyTextAnalyzer will automatically remove all ‘non-word characters’ (according to Regex definition) and redundant spaces. The loading time is very fast (a couple of seconds with a text the size of the Voynich manuscripts).

Once the source text has been loaded you can proceed to the ‘Slot grammar’ tab.

[1] And indeed the slot grammars functions can be applied to any language, not only to Voynichese. The pre-defined grammar “CVC generic language” is a rough example of a grammar usable on any language.



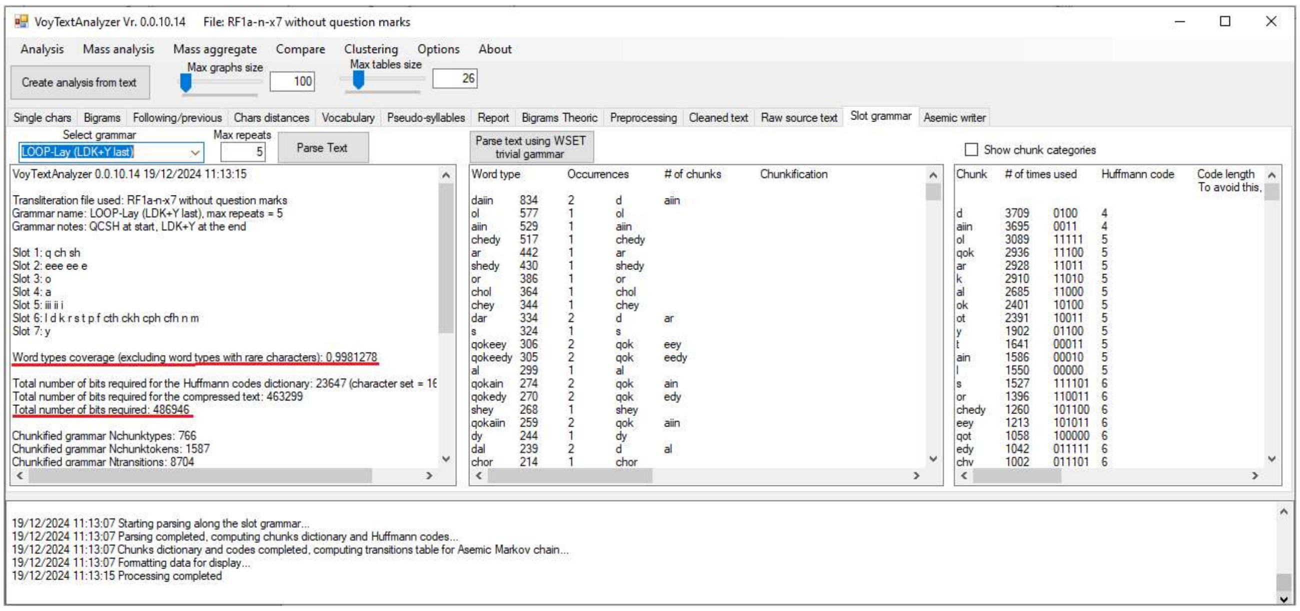
```
18/12/2024 19:04:07 Found 8424 different words, maximum word length = 20
18/12/2024 19:04:07 Extracting pseudo-syllables...
18/12/2024 19:04:07 Found 3228 different pseudo-syllables when every vowel is a single nucleus, or 4878 when adiacent vowels are merged in a single nucleus
18/12/2024 19:04:07 Analysis complete
18/12/2024 19:04:07 Formatting data for display...
18/12/2024 19:04:07 Processing completed
```

The drop-down menu allows to choose which of the pre-defined grammars to use. Notice you cannot add or modify grammars without recompiling the software, this will be explained later.

Every grammar has a default number of repetitions, which can be changed with the ‘Max repeats’ box, from 1 up to a maximum determined by the grammar.

Use the button ‘Parse text’ to parse all the word types into chunks and calculate the Huffmann codes and N_{bits}. The Trivial WSET grammar cannot be selected in the drop-down menu: use the “Parse text using WSET trivial grammar” button instead. The elaboration takes about 10 seconds (most of which is writing the output to the text boxes)[2].

[2] Notice the elaboration is not inside a System.Thread (sorry), this mean it could timeout and crash the program. This may be corrected in a future version (the rest of TextAnalyzer functions are threaded), however the processing time is rather fast and a text such as the VMS should not cause any problems (no timeouts should occur when writing to the textboxes).


VoyTextAnalyzer Vr. 0.0.10.14 File: RF1a-n-x7 without question marks
Analysis Mass analysis Mass aggregate Compare Clustering Options About
Create analysis from text
Single chars Bigrams Following/previous Chars distances Vocabulary Pseudo-syllables Report Bigrams Theoric Preprocessing Cleaned text Raw source text Slot grammar Asemic writer
Parse Text
Parse text using WSET trivial gammar
Show chunk categories

Max graphs size: 100
Max tables size: 26
Select grammar: LOOP-Lay (LDK+Y last)
Max repeats: 5

```
VoyTextAnalyzer 0.0.10.14 19/12/2024 11:13:15

Transliteration file used: RF1a-n-x7 without question marks
Grammar name: LOOP-Lay (LDK+Y last), max repeats = 5
Grammar notes: QCSH at start, LDK+Y at the end

Slot 1: q ch sh
Slot 2: eee ee e
Slot 3: o
Slot 4: a
Slot 5: iii ii i
Slot 6: l d k r s t p f cth ckh cph cfh n m
Slot 7: y

Word types coverage (excluding word types with rare characters): 0,9981278

Total number of bits required for the Huffmann codes dictionary: 23647 (character set = 1€
Total number of bits required for the compressed text: 463299
Total number of bits required: 486946

Chunkified grammar Nchunktypes: 766
Chunkified grammar Nchunktokens: 1587
Chunkified grammar Ntransitions: 8704
```

```
Word type    Occurences    # of chunks    Chunkification

daiin     834    2    d       aiin
ol        577    1    ol
aiin      529    1    aiin
chedy     517    1    chedy
ar        442    1    ar
shedy     430    1    shedy
or        386    1    or
chol      364    1    chol
chey      344    1    chey
dar       334    2    d       ar
s         324    1    s
qokeey    306    2    qok     eey
qokeedy   305    2    qok     eedy
al        299    1    al
qokain    274    2    qok     ain
qokedy    270    2    qok     edy
shey      268    1    shey
qokaiin   259    2    qok     aiin
dy        244    1    dy
dal       239    2    d       al
chor      214    1    chor
```

```
Chunk    # of times used    Huffmann code    Code length
                                             To avoid this,

d        3709    0100      4
aiin     3695    0011      4
ol       3089    11111     5
qok      2936    11100     5
ar       2928    11011     5
k        2910    11010     5
al       2685    11000     5
ok       2401    10100     5
ot       2391    10011     5
y        1902    01100     5
t        1641    00011     5
ain      1586    00010     5
l        1550    00000     5
s        1527    111101    6
or       1396    110011    6
chedy    1260    101100    6
eey      1213    101011    6
qot      1058    100000    6
edy      1042    011111    6
chv      1002    011101    6
```

```
19/12/2024 11:13:07 Starting parsing along the slot grammar...
19/12/2024 11:13:07 Parsing completed, computing chunks dictionary and Huffmann codes...
19/12/2024 11:13:07 Chunks dictionary and codes completed, computing transitions table for Asemic Markov chain...
19/12/2024 11:13:07 Formatting data for display...
19/12/2024 11:13:15 Processing completed
```

You can click on any textbox, press Ctrl-A to select everything inside and then Ctrl-C to paste all the data in any Windows program, for instance Excel (the data may seem weirdly aligned in the textboxes but they will be neatly divided in columns once copied to Excel).

Leftmost textbox: Report

It shows the data which have been used in the , particularly the coverage along word types and the total number of bits required (N_{bits}).

$N_{chunktypes}$ is the total number of unique chunks found. See ‘Asemic writer’ for $N_{chunktokens}$ and $N_{transitions}$.

More informations follow in the report (use the scroll bars or copy and paste).

Center textbox: all the text word types as divided in chunks by the grammar

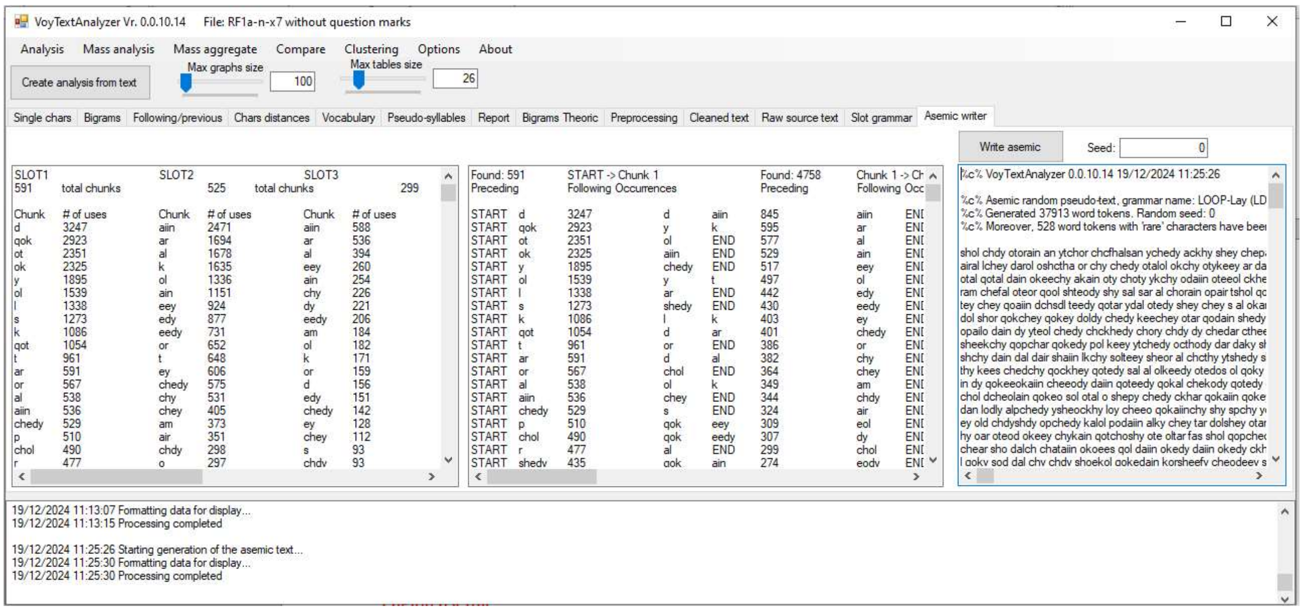
Self-explicative. There are more columns on the right side: the list of all the word types not found by the grammar and the list of the word types discarded because they include ‘rare’ characters.

Rightmost textbox: chunks dictionary

Self-explicative. Clicking on ‘Show chunk categories’ will change the visualization to one where chunks have been divided in ‘categories’ (not used in the paper [1] and not further developed).

Asemic writer tab

All the data in this tab have not been used in the [1] and may be the subject of a future paper.



```
SLOT1                   SLOT2                   SLOT3
591     total chunks            525     total chunks            299

Chunk   # of uses       Chunk   # of uses       Chunk   # of uses
d       3247            aiin    2471            aiin    588
qok     2923            ar      1694            ar      536
ot      2351            al      1678            al      394
ok      2325            k       1635            eey     260
y       1895            ol      1336            ain     254
ol      1539            ain     1151            chy     226
l       1338            eey     924             dy      221
s       1273            edy     877             eedy    206
k       1086            eedy    731             am      184
qot     1054            or      652             ol      182
t       961             t       648             k       171
ar      591             ey      606             or      159
or      567             chedy   575             d       156
al      538             chy     531             edy     151
aiin    536             chey    405             chedy   142
chedy   529             am      373             ey      128
p       510             air     351             chey    112
chol    490             chdy    298             s       93
r       477             o       297             chdv    93
```

```
Found: 591      START -> Chunk 1                Found: 4758     Chunk 1 -> Ch
Preceding       Following Occurrences           Preceding       Following Occ

START   d       3247            d       aiin    845             aiin    ENI
START   qok     2923            y       k       595             ar      ENI
START   ot      2351            ol      END     577             al      ENI
START   ok      2325            aiin    END     529             ain     ENI
START   y       1895            chedy   END     517             eey     ENI
START   ol      1539            y       t       497             ol      ENI
START   l       1338            ar      END     442             edy     ENI
START   s       1273            shedy   END     430             eedy    ENI
START   k       1086            l       k       403             ey      ENI
START   qot     1054            d       ar      401             chedy   ENI
START   t       961             or      END     386             or      ENI
START   ar      591             d       al      382             chy     ENI
START   or      567             chol    END     364             chey    ENI
START   al      538             ol      k       349             am      ENI
START   aiin    536             chey    END     344             chdy    ENI
START   chedy   529             s       END     324             air     ENI
START   p       510             qok     eey     309             eol     ENI
START   chol    490             qok     eedy    307             dy      ENI
START   r       477             al      END     299             chol    ENI
START   shedv   435             qok     ain     274             eodv    ENI
```

```
%c% VoyTextAnalyzer 0.0.10.14 19/12/2024 11:25:26

%c% Asemic random pseudo-text, grammar name: LOOP-Lay (LD
%c% Generated 37913 word tokens. Random seed: 0
%c% Moreover, 528 word tokens with 'rare' characters have bee

shol chdy otorain an ytchor chcfhalsan ychedy ackhy shey chep
airal lchey darol oshctha or chy chedy otalol okchy otykeey ar da
otal qotal dain okeechy akain oty choty ykchy odaiin oteeol ckhe
ram chefal oteor qool shteody shy sal sar al chorain opair tshol qc
tey chey qoaiin dchsdl teedy qotar ydal otedy shey chey s al oka
dol shor qokchey qokey doldy chedy keechey otar qodain shedy
opailo dain dy yteol chedy chckhedy chory chdy dy chedar cthee
sheekchy qopchar qokedy pol keey ytchedy octhody dar daky sl
shchy dain dal dair shaiin lkchy solteey sheor al chcthy ytshedy s
thy kees chedchy qockhey qotedy sal al olkeedy otedos ol qoky
in dy qokeeokaiin cheeody daiin qoteedy qokal chekody qotedy
chol dcheolain qokeo sol otal o shepy chedy ckhar qokaiin qoke
dan lodly alpchedy ysheockhy loy cheeo qokaiinchy shy spchy y
ey old chdyshdy opchedy kalol podaiin alky chey tar dolshey otar
hy oar oteod okeey chykain qotchoshy ote oltar fas shol qopche
chear sho dalch chataiin okoees qol daiin okedy daiin okedy ckh
l gokv sod dal chv chdv shoekol gokedain korsheefv cheodeev s
```

```
19/12/2024 11:13:07 Formatting data for display...
19/12/2024 11:13:15 Processing completed

19/12/2024 11:25:26 Starting generation of the asemic text...
19/12/2024 11:25:30 Formatting data for display...
19/12/2024 11:25:30 Processing completed
```

The first two textboxes are written by the ‘Parse text’ function of the previous tab. You need instead to click the “Write asemic” button to generate the random text in the last box.

Leftmost textbox: chunkified grammar

Briefly, the chunkified grammar is generated from the word types divided in chunks (see previous ‘Slot grammar’ tab, central textbox). It’s a derived slot grammar [chunkified grammar = CHUNK(original grammar)] where the n-th slot is built by taking all the chunks which appear in the n-th position of a word type. For example: ‘daiin’ = [d] + [aiin], the [d] goes in the first slot of the chunkified grammar, the [aiin] goes in the second.

The total number of chunks used in the slots of the chunkified grammar is the $N_{\text{chunktokens}}$ value displayed in the report.

Center textbox: Markov chain transitions table

It’s computed from the chunkified slot grammar *and* the frequency with which each word token appears in the text. It gives the probability of transitioning, say, form START to a [d] as 1st chunk, then from [d] in the 1st chunk to [aiin] as 2nd chunk and so on.

The total number of transitions is the $N_{\text{transitions}}$ value displayed in the report.

Rightmost textbox: asemic random text

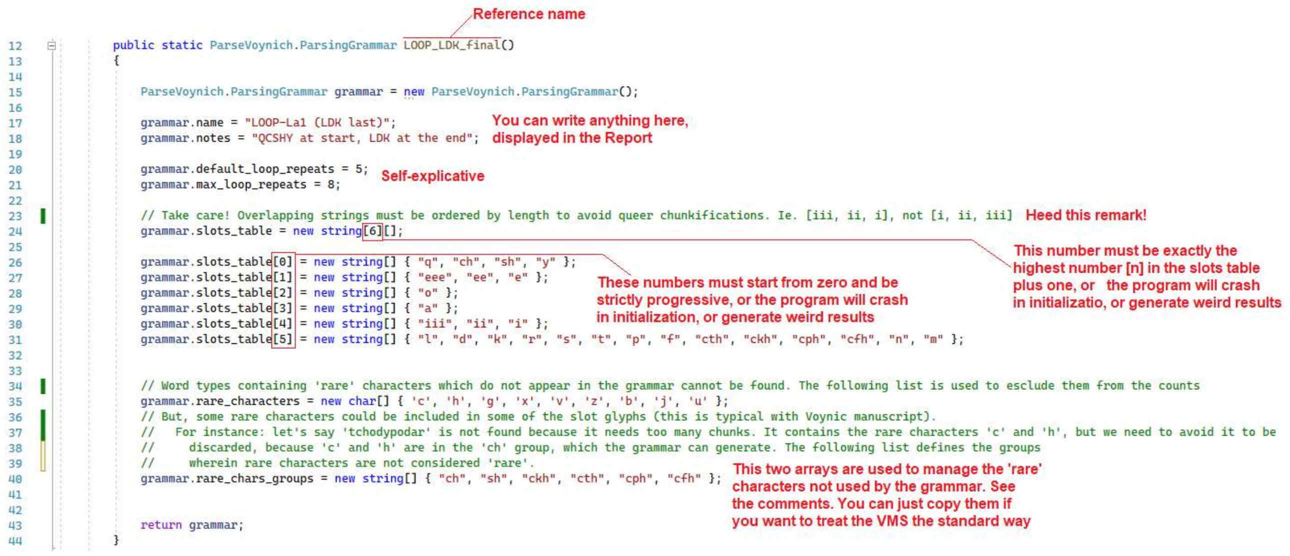
The asemic text is automatically generated with as many words as the original text to make comparisons easier.

VoyTextAnalyzer generates a very good pseudo-Voynich text, indistingushable from the original on many statistical properties (data no shown, you can check yourself by copying the text and using your favourite analysis tool, or the built-in statistical function of VoyTextAnalyzer). The main statistical difference is the asemic writer generates less hapax legomena and rare words than found in the original Voynich. This may be the subject of a future paper.

Adding a grammar using source files

VoyTextAnalyzer has no grammar editor. Grammars can be added/modified only in source files. The environment in which VoyTextAnalyzer has been developed is Microsoft Visulat Studio 2022 free edition (.NET framework 4.5.2).

All the grammars are in ParseGrammarsLibrary.cs



```
public static ParseVoynich.ParsingGrammar LOOP_LDK_final()
{

    ParseVoynich.ParsingGrammar grammar = new ParseVoynich.ParsingGrammar();

    grammar.name = "LOOP-La1 (LDK last)";
    grammar.notes = "QCSHY at start, LDK at the end";

    grammar.default_loop_repeats = 5;
    grammar.max_loop_repeats = 8;

    // Take care! Overlapping strings must be ordered by length to avoid queer chunkifications. Ie. [iii, ii, i], not [i, ii, iii]
    grammar.slots_table = new string[6][];

    grammar.slots_table[0] = new string[] { "q", "ch", "sh", "y" };
    grammar.slots_table[1] = new string[] { "eee", "ee", "e" };
    grammar.slots_table[2] = new string[] { "o" };
    grammar.slots_table[3] = new string[] { "a" };
    grammar.slots_table[4] = new string[] { "iii", "ii", "i" };
    grammar.slots_table[5] = new string[] { "l", "d", "k", "r", "s", "t", "p", "f", "cth", "ckh", "cph", "cfh", "n", "m" };


    // Word types containing 'rare' characters which do not appear in the grammar cannot be found. The following list is used to esclude them from the counts
    grammar.rare_characters = new char[] { 'c', 'h', 'g', 'x', 'v', 'z', 'b', 'j', 'u' };
    // But, some rare characters could be included in some of the slot glyphs (this is typical with Voynic manuscript).
    //   For instance: let's say 'tchodypodar' is not found because it needs too many chunks. It contains the rare characters 'c' and 'h', but we need to avoid it to be
    //     discarded, because 'c' and 'h' are in the 'ch' group, which the grammar can generate. The following list defines the groups
    //     wherein rare characters are not considered 'rare'.
    grammar.rare_chars_groups = new string[] { "ch", "sh", "ckh", "cth", "cph", "cfh" };


    return grammar;
}
```

Reference name

You can write anything here, displayed in the Report

Self-explicative

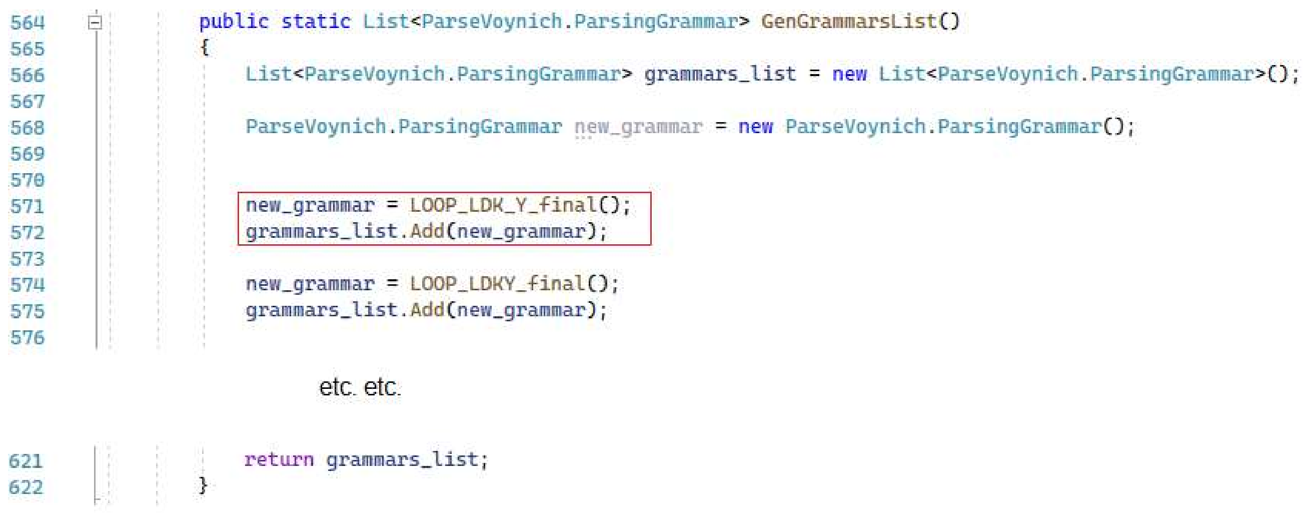
Heed this remark!

This number must be exactly the highest number [n] in the slots table plus one, or the program will crash in initializatio, or generate weird results

These numbers must start from zero and be strictly progressive, or the program will crash in initialization, or generate weird results

This two arrays are used to manage the 'rare' characters not used by the grammar. See the comments. You can just copy them if you want to treat the VMS the standard way

After creating a new grammar, it must be added in the function ParseGrammarsLibrary.GenGrammarsList() for it to appear in the drop-down menu:



```
public static List<ParseVoynich.ParsingGrammar> GenGrammarsList()
{
    List<ParseVoynich.ParsingGrammar> grammars_list = new List<ParseVoynich.ParsingGrammar>();

    ParseVoynich.ParsingGrammar new_grammar = new ParseVoynich.ParsingGrammar();


    new_grammar = LOOP_LDK_Y_final();
    grammars_list.Add(new_grammar);

    new_grammar = LOOP_LDKY_final();
    grammars_list.Add(new_grammar);


            etc. etc.

    return grammars_list;
}
```

Short description of the source files

I will only describe the classes used by the 'Slot grammars'/'Asemic writer' tabs, not the whole program. All the names in the sources are in English and as self-explicative as possible, but comments are in Italian (I'm sorry).

The launch routines of the functions from the buttons and the data visualization in the tabs are all in Form1.

Files used by the Slot grammars processing:

ParseGrammarsLibrary.cs	Definition of the grammars, see above

ParseVoynich.cs	`ParseVoynich.parse_words`
	1) Parses the word types into chunks
	`ParseVoynich.get_chunks_data`
	2) Creates of the chunks dictionary.

ParseHuffmann.cs	`ParseHuffmann.get_Huffmann_codes` 3) Adds the codes to the chunk dictionary

Files used by the Asemic writer processing:

PackGrammarAndWrite.cs	`PackGrammarAndWrite.calculate_chunkified_grammar`
	1) Compilation of the chunkified grammar
	2) Compilation of the transitions table
	`PackGrammarAndWrite.Markov_asemic_writer`
	Asemic writer

Input class for the slot grammars processing

The dictionary of all the word tokens found in the source text is used as the input of the slot grammars processing. It is computed by the original TextAnalyzer functions when 'Creating an analysis from text'.

```
public List<EValueOcc> vocabulary_words_distribution = new List<EValueOcc>();
```

An EvalueOcc is a basic class, formed by a string + a number.

```
public class EValueOcc
{
    public string element;

    public long value;
}
```

In this case `element` stores the word type, while `value` stores the number of times the word type appears in the text (occurrences).

`vocabulary_words_distribution` example:

element	value
daiin	834
ol	577
aiin	529
…	…

Slot grammar processing classes (ParseVoynich.cs and ParseHuffmann.cs)

The main class is `ParsingResults`. It stores the word types parsed into chunks and the chunks dictionary. It is computed from `vocabulary_words_distribution`.

```
public class ParsingResults
{
    public ParsingGrammar grammar;

    public int total_wordtypes_found;                                   // Nhits
    public int total_wordstypes_not_found_but_with_rare_characters;
    public float total_grammar_wordtypes;                               // Ngrammarsize

    public int total_number_of_chunktypes;     // Nchunktypes

    public List<WordParseData> data = new List<WordParseData>();    // Chunkified word types

    public Dictionary<string, Chunk> chunks_dictionary = new Dictionary<string, Chunk>();


    Two more dictionaries here for chunk 'categories'
    Not further developed and not documented here

}
```

`public ParsingGrammar grammar;`	The slot grammar used for parsing (pre-loaded by choosing a grammar from the drop-down menu).
`public List<WordParseData> data`	The list of all word types divided in chunks (provided there was a 'hit')

`public Dictionary<string, Chunk> chunks_dictionary`	The chunks dictionary

```
public class WordParseData
{
    public string word;
    public long occurrences;

    public bool word_found;
    public bool word_not_found_but_has_rare_characters;

    public int number_of_words_chunks;

    public List<List<EValue>> parsed_word = new List<List<EValue>>(); // Contains the chunks which compose a word type, divided slot by slot
}
```

Example of `parsed_word` (An EValue is like an EvalueOcc, but uses an ‘int’ rather than a ‘long’ numeric type. In parsed_word `value` stores the number of the slot which scored a ‘hit’. It’s used for the chunk ‘categories’ and is not further detailed).

The word type is ‘daiin’, the grammar is LOOP-Lay

“d”, 5		
“a”, 3	“ii”, 4	“n”, 5

```
public Dictionary<string, Chunk> chunks_dictionary
```

The key of the dictionary is the chunk string (for instance: “aiin”).

```
public class Chunk
{
    public string chunk;
    public long number_of_times_used;

    public string Huffman_code = "";
}
```

`Chunk` stores the chunk string (same as the dictionary key), the number of times the chunk is used in the text and the assigned Huffman code. Huffmann codes are added to the pre-compiled dictionary by `ParseHuffmann.get_Huffmann_codes`

Calculation of N$_{bits}$

It’s calculated at visualization time, in Form1. `display_parsing_report`

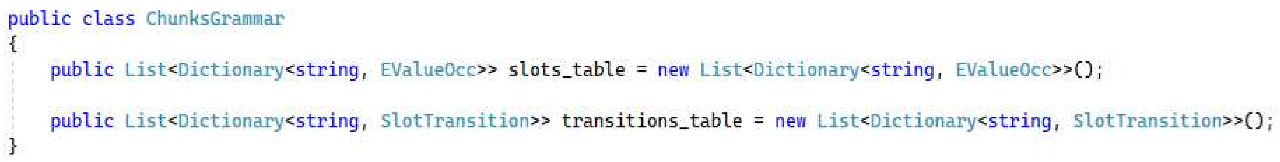
```
// Nbits metric: length of the Huffmann-compressed text (text + chunks dictionary)
int charset_size = text_analyzer.analysis_results.monograms_distribution_excluding_spaces.Count - Voynich_parsing_results.grammar.rare_characters.Length;
double bits_per_character = Math.Log(charset_size, 2);

long Huffmann_compressed_text_total_required_bits = 0;
double Huffmann_dictionary_total_required_bits = 0;
foreach (ParseVoynich.Chunk chunk in Voynich_parsing_results.chunks_dictionary.Values)
{
    Huffmann_compressed_text_total_required_bits += chunk.number_of_times_used * chunk.Huffman_code.Length;

    Huffmann_dictionary_total_required_bits += chunk.chunk.Length * bits_per_character + chunk.Huffman_code.Length;
}
```

Notice the formula used for text_bits is not the same as the one in the paper [1], but it is equivalent (implementing the formula as writte in the paper would hve resulted in an ugly software).

`charset_size` is calculated by subtracting the number of ‘rare’ characters which are not considered by the grammar from the total number of characters found in the text.

Chunkified grammar and asemic writer

The main class `ChunksGrammar`, in PackGrammarAndWrite.cs, contains the slots of the chunkified grammar and the transitions table:



```
public class ChunksGrammar
{
    public List<Dictionary<string, EValueOcc>> slots_table = new List<Dictionary<string, EValueOcc>>();

    public List<Dictionary<string, SlotTransition>> transitions_table = new List<Dictionary<string, SlotTransition>>();
}
```

Given this is out of the scope of the paper [1] , no further explanations are given here.

References

[1] Mauro Lanzini “The evaluation of slot grammars and their application in the study of the Voynich” Manuscript, 2025. https://www.academia.edu/