

Eren Alpar

This is the report of Enadream ChatBot version 0.71. The source code of the project can be found in <https://github.com/enadream/ChatBot/> (The repository can be private to protect the source code in the future). The binary of the application can be found in <https://github.com/enadream/ChatBot/releases>.

Which language has been used for the application ?

C++ has been used for the whole project and will be continued with c++ in the future. C++ is an object-oriented programming (OOP) language that is viewed by many as the best language for creating large-scale applications. C++ is a superset of the C language.

Advantages of C++ Over Other Languages

1. **Object-Oriented:** C++ is an object-oriented programming language which means that the main focus is on objects and manipulations around these objects. This makes it much easier to manipulate code, unlike procedural or structured programming which requires a series of computational steps to be carried out.
2. **Speed:** When speed is a critical metric, C++ is the most preferred choice. The compilation and execution time of a C++ program is much faster than most general-purpose programming languages. And the Chatbot requires high speed to make sense of sentences as fast as possible.
3. **Compiled:** Unlike other programming languages where no compilation is required, every C++ code has to be first compiled to a low-level language and then executed. So the application can perform the program tasks only instead of dealing with parsing the source code and translate it into machine code in the run time.
4. **Pointer Support:** C++ also supports pointers which are often not available in other programming languages. So that we can control the application more detailed and can make better optimization.
5. **Closer to Hardware:** C++ is closer to hardware than most general-purpose programming languages. This makes it very useful in those areas where hardware and software are closely coupled together, and low-level support is needed at the software level.

So the main point of using c++ in this project is better optimization and faster application.

Disadvantages of C++ Over Other Languages

The main disadvantage of using C++ is writing a lot of code for everything compared to other interpretive languages. The programmer has to control when to free a memory, when to create a new memory space, and all other things. And these things increase program complexity. And it might be difficult to deal with it.

Analysis of the code

The main file of the project is located in “src” file. Which contains .cpp and .hpp files of the project. The version 0.71 contains 27 files, 16 of which are header files and 11 are source files. These files contain a total of 4660 lines of code, 475 of which are comment lines, 697 of which are blank lines. (More details in the image 1.1).

```
D:\Projects\C++ Projects\ChatBot\ChatBot\src>cloc .
  27 text files.
  27 unique files.
   0 files ignored.

github.com/AlDanial/cloc v 1.94  T=0.20 s (136.8 files/s, 29557.1 lines/s)
-----
Language                     files      blank     comment      code
-----
C++                          11         524         419         3797
C/C++ Header                 16         173          56          863
-----
SUM:                          27         697         475         4660
-----

D:\Projects\C++ Projects\ChatBot\ChatBot\src>
```

Image 1.1. Shows the number of files and codes.

How does the chatbot work?

Chatbot consists of 10 different objects to parse 10 different types of parts of speech (Image 1.2). Each different type of the object has different parsing abilities. The verb object can parse verbs and their inflectional forms -ing, -ed, -s. The noun object can parse nouns and their inflectional form -s. The aux_verb object can parse auxiliary verbs and their negative forms of it. And other objects just make a search in their dictionaries to find the input.

The chatbot can parse 11 different part of speech. These are Noun, Verb, AuxiliaryVerb, Pronoun, Adverb, Adjective, Preposition, Conjunction, Interjection, Determiner, Punctuation. (Image 1.3)

```
struct Verb {    // 24 byte
    char chars[VERB_CHAR_SIZE];
    uint8 length;
    SuffixGroup suffixes;
};
```

Image 1.4. Shows the verb struct.

```
namespace handle {
    class MainHandler {
    private: // variables
        noun::NounHandler noun;
        basic::UnindexedList pronoun;
        basic::IndexedList adv;
        basic::IndexedList adj;
        basic::UnindexedList prepos;
        basic::UnindexedList conj;
        basic::UnindexedList interj;
        basic::AuxiliaryVerb aux_verb;
        basic::UnindexedList det;

    public: // variables
        verb::VerbHandler verb;
        tkn::Tokenizer tokenize;
    };
}
```

Image 1.2. Shows the objects.

```
enum WordType : uint8 {
    6     Undefined,
    7
    8
    9     Noun,
    10    Verb,
    11    AuxiliaryVerb,
    12    Pronoun,
    13    Adverb,
    14    Adjective,
    15    Preposition,
    16    Conjunction,
    17    Interjection,
    18    Determiner,
    19    Punctuation,
    20
};
```

Image 1.3. Shows the word types.

Verb Handler Class

The verb handler class handles the verb read, write, parse operation (Image 1.5). Firstly, the class has to read a verb from a dictionary this operation can be done with MultipleVerbAdder function. This function takes a char pointer and the size of the array and reads it line by line. Each line has to be a verb or verb group if it's irregular verb. And the line be sent to the AddNewVerb function. This function parses the line and converts the line lowercase and remove spaces and tabs. Then the function sends data to the AddNewVerb function. This function checks the ending of the verb for exceptions -s, -ed, -ing suffixes. And the function saves the verb chars, length suffixes into a 24 byte data area if it's unique. (Image 1.4) The function saves the record with 2 characters index. For example if the verb start with "pl"(play) the function puts the verb struct into "pl" index.

```
class VerbHandler { // 32 byte size
private: // Variables
    VerbIndexList* buffer;
    IrrVerbsList irrVerbCollection;

private: // Functions
    uint8 ED_Parser(const char* verb_chars, const uint8& lenght, std::vector<Verb*>& out_verbs) const;
    uint8 ING_Parser(const char* verb_chars, const uint8& lenght, std::vector<Verb*>& out_verbs) const;
    uint8 S_Parser(const char* verb_chars, const uint8& lenght, std::vector<Verb*>& out_verbs) const;

    uint8 FindVerb(const char* verb_chars, const int& length, std::vector<Verb*>* out_verbs = nullptr) const;
    uint8 FindWithException(const char* verb_chars, const int& length, SuffixGroup& exception_p, std::vector<Verb*>& out_verbs) const;

    Verb& CreateVerb(const char* verb_chars, const int& str_lenght, const SuffixGroup& exception_p);
    void CreateIrregularVerb(Verb& verb);
    int8 UpdateIrrVerbAddress(Verb& new_address, const Verb& old_address);
    int8 AddNewVerb(const char* line, const uint16& size);

    void CheckException_S(Verb& verb) const;
    void CheckException_ING(Verb& verb) const;
    void CheckException_ED(Verb& verb) const;

    void ExceptionToStr(const int8 type, const Suffix_t& ex_type, String& out_string) const;
    void VerbToStr(const Verb& verb, String& out_string) const;
public: // Functions
    VerbHandler();
    ~VerbHandler();
    void GetVerbsWithIndex(const char* index_couple, String& out_string);
    uint16 GetAllIrregularVerbs(String& out_string) const;
    int8 ParseVerb(const String& raw_string, TypeAndSuffixes& word, String& out_string, const bool write_result) const;
    void MultipleVerbAdder(const char* file, const uint64& size);
    int8 Free();
};
```

Image 1.5. Shows the verb handler class.

Noun Handler Class

The noun handler class contains functions related with nouns. First step of this function is reading the dictionary. Dictionary format of the nouns is simply a list that contains words line by line. This operation can be done with MultipleAdder function. This function reads a char array address and the size of it. And then reads each line one by one and add the word to the indexed list. All other functions related with the nouns has been shown in the picture 1.7. The parse function, search words in the ram with both native forms and inflections.

```
struct Noun // 24 byte
{
    char chars[NOUN_CHAR_SIZE];
    uint8 length;
    Exception s;
};
```

Image 1.6. Shows the noun struct.

```

class NounHandler {
private:
    NounList* nounLists;
    IrregularNounList irrNounList;

private:
    void CheckException_S(Noun& noun) const;
    uint8 S_Parser(const char* noun_chars, const uint8& length, std::vector<Noun*>& out_nouns) const;
    uint8 Pos_Parser(const char* noun_chars, const uint8& length, std::vector<Noun*>& out_nouns) const;
    void CreateIrrNoun(Noun& noun);
    int8 UpdateIrrNounAdress(Noun& new_address, const Noun& old_address);

public:
    NounHandler();
    ~NounHandler();

    int8 AddNoun(const char* word_chars, const uint32& length);
    Noun& CreateNewNoun(const char* noun_chars, const uint32& str_length, const Exception& exception_p);
    void MultipleAdder(const char* file, const uint64& line_length);
    uint8 FindNoun(const char* word_chars, const uint8& length, std::vector<Noun*>* out_nouns = nullptr) const;
    uint8 FindWithException(const char* noun_chars, const int& length, Exception ex_type, std::vector<Noun*>& out_nouns) const;
    int8 ParseNoun(const String& raw_string, TypeAndSuffixes& word, String& out_string, const bool write_result) const;
    void ExceptionToStr(const Exception ex_type, String& out_string) const;
    void Free();
};

```

Image 1.7. Shows the NounHandler class.

Indexed and Unindexed Classes

There are two main types of classes in the project, those are indexed and unindexed classes. Indexed class basically used for open class dictionaries such as Nouns, Verbs, Adverbs, Adjectives. Unindexed classes used for closed class dictionaries such as prepositions, conjunctions etc. (Image 1.2).

Unindexed list objects simply used to make a search in a short dictionary list (Image 1.8). They are capable of reading dictionary from disk and make a search in it. It also checks uniqueness control for the words. There is also an inherited unindexed class called AuxiliaryVerb. This class inherited UnindexedList's functions. AuxiliaryVerb class is specialized for parsing negative forms of Auxiliary verbs such as can't wouldn't etc.

Indexed list objects used to create complex dictionary with 2 characters indexes (Image 1.9). They are capable of reading dictionary from disk and make a search in it with a faster way. All classes control uniqueness of the words to make sure that's not any duplication of the words.

```

class UnindexedList {
protected:
    SimpleWord* words = nullptr;
    uint32 amount = 0;
    uint32 capacity = 0;

protected: // Functions
    void IncreaseSpace();

public:
    UnindexedList();
    ~UnindexedList();

    int8 AddWord(const char* word_chars, const uint32& length);
    void MultipleAdder(const char* file, const uint64& line_length, const char* type);
    int32 FindWord(const char* word_chars, const uint8& length) const;
    int8 ParseWord(const String& raw_string, String& out_string, const bool write_result);
    void Free();
};

```

Image 1.8. Shows the UnindexedList class.

```

class IndexedList {
protected:
    WordList* wordLists;

protected: // Functions
    SimpleWord* CreateWord(const char* word_chars, const uint32& str_length);

public:
    IndexedList();
    ~IndexedList();

    int8 AddWord(const char* word_chars, const uint32& length);
    void MultipleAdder(const char* file, const uint64& line_length, const char* type);
    SimpleWord* FindWord(const char* word_chars, const uint8& str_length);
    int8 ParseWord(const String& raw_string, String& out_string, const bool write_result);
    void Free();
};

```

Image 1.9. Shows the IndexedList class.

What's indexing and why it has been used?

Indexing, broadly, refers to the use of some benchmark indicator or measure as a reference or yardstick. In this project indexing used in open class dictionaries to make faster search in the dictionary. All indexed dictionaries use 2 char index such as aa, ab, ac... etc. There is one buffer which holds addresses of index blocks (Image 2.1). Index blocks contain words with starting index. There is 26×27 address in the main block. 26 chars for English letters and one more block for hyphen(-) char. For example to find a word x-ray in the dictionary, the program has to find the address of 'x-' first and time consuming of this operation is constant. The formula to find the exact location of any index is;

$f(x) = \text{row} * 27 + \text{col}$. Row is the index of first char and col is index of second char.

The char a is accepted as 0 the char '-' accepted as 26. So that to find x-ray in the program substitute formula as:

$f(x) = 23 * 27 + 26 = 647$. buffer[647] contains the address of 'x-' index. If the word is exist in the dictionary, it has to be in this block. So that search performance is enhanced.

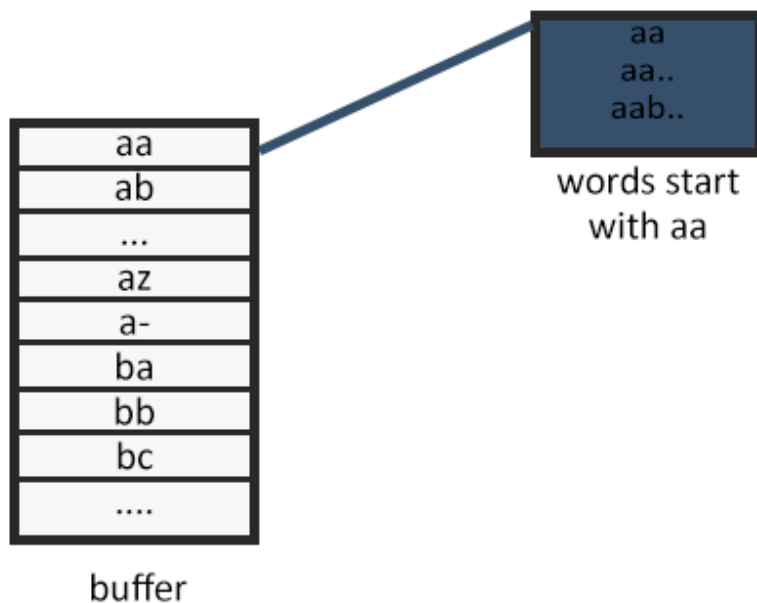


Image 2.1. Shows the data blocks.

```
#define WORD_ROW 26
#define WORD_COLUMN 27
```

Image 2.2. Shows the row and column size.

The time complexity of the indexing shown in the Image 2.3. The first equation in the image shows a linear search in a list. And the second equation in the image shows indexing. There is 702 subblock (26×27) in the memory. Because of the starting bigrams of the words are not evenly distributed in English the real time complexity of the function wouldn't be like in the picture. But it enhances the performance of searching.

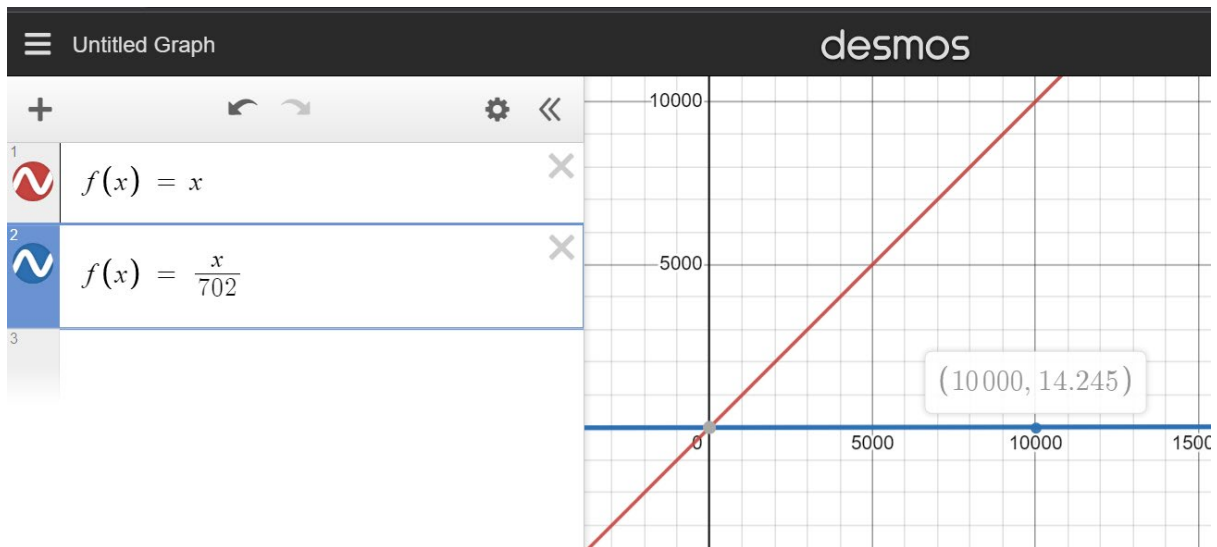


Image 2.2. Shows the time complexities of linear search vs indexed search with 702 different indices.

Word amounts of dictionaries

There is 57202 nouns, 8760 verbs, 20744 adjectives, 3752 adverbs, 66 pronouns, 68 prepositions, 12 conjunctions, 12 interjections, 26 auxiliary verbs, 20 determiners in the dictionaries. (Image 2.3)

```
ECLI $ read -dir "dict/nouns.txt" -noun
[INFO]: The file has readed successfully.
[INFO]: 57202 nouns added successfully.

ECLI $ read -dir "dict/verbs.txt" -verb
[INFO]: The file has readed successfully.
[INFO]: 8760 verbs added successfully.

ECLI $ read -dir "dict/adjs.txt" -adjective
[INFO]: The file has readed successfully.
[INFO]: 20744 adjectives added successfully.

ECLI $ read -dir "dict/advs.txt" -adverb
[INFO]: The file has readed successfully.
[INFO]: 3752 adverbs added successfully.

ECLI $ read -dir "dict/pronouns.txt" -pronoun
[INFO]: The file has readed successfully.
[INFO]: 66 pronouns added successfully.

ECLI $ read -dir "dict/prepos.txt" -preposition
[INFO]: The file has readed successfully.
[INFO]: 68 prepositions added successfully.

ECLI $ read -dir "dict/conjs.txt" -conjunction
[INFO]: The file has readed successfully.
[INFO]: 12 conjunctions added successfully.

ECLI $ read -dir "dict/interjs.txt" -interjection
[INFO]: The file has readed successfully.
[INFO]: 12 interjections added successfully.

ECLI $ read -dir "dict/verbs_aux.txt" -aux_verb
[INFO]: The file has readed successfully.
[INFO]: 26 auxiliary verbs added successfully.

ECLI $ read -dir "dict/dets.txt" -determiner
[INFO]: The file has readed successfully.
[INFO]: 20 determiners added successfully.
```

Image 2.3. Shows dictionary Info logs.

How to use ChatBot ?

When the program is started there is no dictionary data exist on the ram. To read dictionaries user has to use ECLI (Enadream command line interface). Ecli commands are similar to the Linux bash. All existing commands in Ecli can be found by typing "help" (Image 2.4). To get more information about a command can be found by typing "help xxx"(xxx is command that you want to use.) (Image 2.5)

```
ECLI $ help
[INFO]: All the existing commands are;
  clear : Use this command to clear console.
  delete : Use this command to delete data.
  exit : Use this command to exit the program.
  help : Use this command to get information about commands.
  parse : Use this command to parse verbs, nouns, etc.
  sr_parse : Use this command to parse a sentence.
  print : Use this command to print some data.
  read : Use this command to read data from disk.
  tester : Use this command to run morphological tester.
  tokenize : Use this command to tokenize a string.
  -command : Use this parameter to see parsed command text.
For the more information about spesific command or usage type "help read". i.e
```

Image 2.4. Shows commands.

```
ECLI $ help read
[INFO]: read : This command reads data from disk and and can have these parameters;
  -dir : This parameter indicates the directory of the file.
  -noun : This parameter specifies the file as noun.
  -verb : This parameter specifies the file as verb.
  -adverb : This parameter specifies the file as adverb.
  -pronoun : This parameter specifies the file as pronoun.
  -adjective : This parameter specifies the file as adjective.
  -preposition : This parameter specifies the file as preposition.
  -conjunction : This parameter specifies the file as conjunction.
  -interjection : This parameter specifies the file as interjection.
  -aux_verb : This parameter specifies the file as auxiliary verb.
  -determiner : This parameter specifies the file as determiner.
Example usages;
ECLI $ read -dir "dict/nouns.txt" -noun
ECLI $ read -dir "dict/verbs.txt" -verb
```

Image 2.5. Shows output of help read.

First steps of the program is reading all dictionaries from the disk. And these commands has to be entered before starting parsing process.

```
read -dir "dict/nouns.txt" -noun
read -dir "dict/verbs.txt" -verb
read -dir "dict/adjs.txt" -adjective
read -dir "dict/advs.txt" -adverb
read -dir "dict/pronouns.txt" -pronoun
read -dir "dict/prepos.txt" -preposition
read -dir "dict/conjs.txt" -conjunction
read -dir "dict/interjs.txt" -interjection
read -dir "dict/verbs_aux.txt" -aux_verb
read -dir "dict/dets.txt" -determiner
```


To parse a word there is a command called “parse”. Usage of the parse command shown below. (Image 2.6)

```
ECLI $ help parse
[INFO]: parse : This command parses text from console and can have these parameters;
      "" : (With no parameter) You can parse an input by looking all types.
      -noun : This parameter parse the input by just looking to nouns.
      -verb : This parameter parse the input by just looking to verbs.
      -pronoun : This parameter parse the input by just looking to pronouns.
      -adverb : This parameter parse the input by just looking to adverbs.
      -adjective : This parameter parse the input by just looking to adjectives.
      -preposition : This parameter parse the input by just looking to prepositions.
      -conjunction : This parameter parse the input by just looking to conjunctions.
      -interjection : This parameter parse the input by just looking to interjections.
      -aux_verb : This parameter parse the input by just looking to auxiliary verb.
      -determiner : This parameter parse the input by just looking to determiner.
Example usages;
ECLI $ parse "speaking"
ECLI $ parse -adverb "speaking"
```

Image 2.6. Shows parse options.

The output of an input shown in the below (Image 2.7). Each different result comes from different dictionaries, if the result is found as inflectional word, the result shown as purple color with inflectional indicator shown as Result 2.

```
ECLI $ parse "speaking"

[RESULT 1]:
[INFO]: Noun has been found:
1. speaking      [Sfx_S]: -s

[RESULT 2]:
[INFO]: Verb has been found:
[Inflectional Verb (-ing)]: speak + ing
1. speak        [Sfx_ED]: Irr verb V1, [Sfx_S]: -s, [Sfx_ING]: -ing

[RESULT 3]:
[INFO]: Adjective has been found:
1. speaking
```

Image 2.7. Shows parsing of “speaking”.

All inputs are converted into lowercase before parsing operation. So that “SpEakIng” and “SPEAKING” can be parsed.

To use tester there is a command called “tester”. Usage of the tester command shown below. (Image 2.8)

```
ECLI $ help tester
[INFO]: tester : This command runs the morphological tester.
Example usage;
ECLI $ tester -dir "test.txt"
```

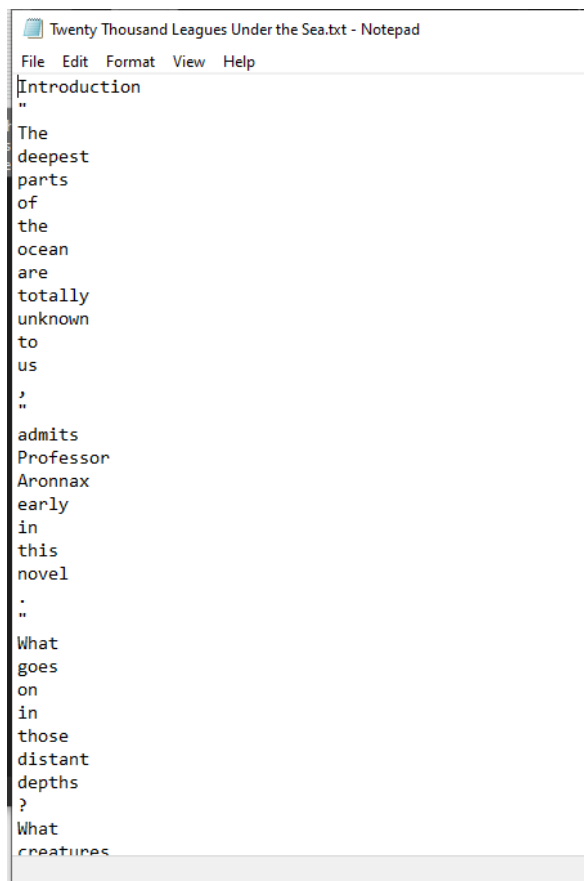
Image 2.8. Shows tester help.

The book "Twenty Thousand Leagues Under the Sea" has been converted into tokens and used tester function to analyze each token in the book (Image 2.9). The input format for the tester is putting tokens line by line. The tester operation for the book Twenty Thousand Leagues Under the Sea took 0.337196 seconds. The file contains 168144 tokens, 163222 of which are parsed successfully, 4922 words failed. Average time for a token to parse is 2 microseconds.

After the tester operation done the output file written as [FILE_NAME]_out.txt. The output file contains the parsed results of each token. Failed tokens can be found by searching "?." In the output file. (Image 2.9.2, Image 2.9.3)

```
ECLI $ tester -dir "Twenty Thousand Leagues Under the Sea.txt"
[INFO]: The file has readed successfully.
[INFO]: 168144 words parsed, 163222 words parsed successfully, 4922 words failed.
The operation completed in 0.337196 seconds. (2.0054e-06 sec / word)
[INFO]: The file has written successfully.
```

Image 2.9. Shows tester output.

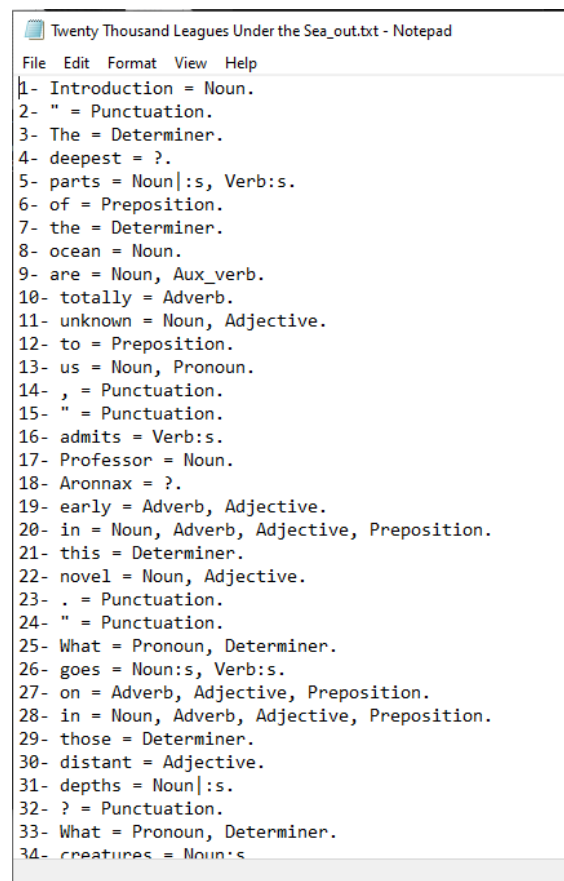


Twenty Thousand Leagues Under the Sea.txt - Notepad

File Edit Format View Help

Introduction
"
The
deepest
parts
of
the
ocean
are
totally
unknown
to
us
,
admits
Professor
Aronnax
early
in
this
novel
.
What
goes
on
in
those
distant
depths
?
What
creatures

Image 2.9.2. Input file.



Twenty Thousand Leagues Under the Sea_out.txt - Notepad

File Edit Format View Help

1- Introduction = Noun.
2- " = Punctuation.
3- The = Determiner.
4- deepest = ?.
5- parts = Noun|:s, Verb:s.
6- of = Preposition.
7- the = Determiner.
8- ocean = Noun.
9- are = Noun, Aux_verb.
10- totally = Adverb.
11- unknown = Noun, Adjective.
12- to = Preposition.
13- us = Noun, Pronoun.
14- , = Punctuation.
15- " = Punctuation.
16- admits = Verb:s.
17- Professor = Noun.
18- Aronnax = ?.
19- early = Adverb, Adjective.
20- in = Noun, Adverb, Adjective, Preposition.
21- this = Determiner.
22- novel = Noun, Adjective.
23- . = Punctuation.
24- " = Punctuation.
25- What = Pronoun, Determiner.
26- goes = Noun:s, Verb:s.
27- on = Adverb, Adjective, Preposition.
28- in = Noun, Adverb, Adjective, Preposition.
29- those = Determiner.
30- distant = Adjective.
31- depths = Noun|:s.
32- ? = Punctuation.
33- What = Pronoun, Determiner.
34- creatures = Noun:s

Image 2.9.3. Output file.

The sentence parser function can be called using “sr_parse” command. This function firstly tokenizes sentences into tokens then runs morphological analysis. (Image 3.1)

```
ECLI $ help sr_parse
[INFO]: sr_parse : This command parse a sentence and doesn't take any parameter.
Example usage;
ECLI $ sr_parse "This is a simple sentence."
```

Image 3.1. Help sr_parse.

Example input is given as "The aim of this paper is twofold. On the one hand, it attempts to explore several machine learning models for pronoun resolution in Turkish". The output is shown below. (Image 3.2)

```
ECLI $ sr_parse "The aim of this paper is twofold. On the one hand, it attempts to explore several machine learning models for pronoun resolution in Turkish"

[SENTENCE 1]:
1- [The]: Determiner.
2- [aim]: Noun, Verb.
3- [of]: Preposition.
4- [this]: Determiner.
5- [paper]: Noun, Verb.
6- [is]: Aux_verb.
7- [twofold]: Adverb, Adjective.
8- [.] : Punctuation.

[SENTENCE 2]:
1- [On]: Adverb, Adjective, Preposition.
2- [the]: Determiner.
3- [one]: Noun, Pronoun, Adjective.
4- [hand]: Noun, Verb.
5- [,]: Punctuation.
6- [it]: Noun, Pronoun.
7- [attempts]: Noun:s, Verb:s.
8- [to]: Preposition.
9- [explore]: Verb.
10- [several]: Pronoun, Adjective.
11- [machine]: Noun, Verb.
12- [learning]: Noun, Verb:ing.
13- [models]: Noun:s, Verb:s.
14- [for]: Preposition, Conjunction.
15- [pronoun]: Noun.
16- [resolution]: Noun.
17- [in]: Noun, Adverb, Adjective, Preposition.
18- [Turkish]: Noun, Adjective.
```

Image 3.2. Results of sr_parse.