> Our algorithm has been tested for correctness on the Windows operating system with C++11.
> It has also been tested on Ubuntu 16.04 LTS using the flag -std=gnu++11

KEY-POINTS OF OUR MODEL/ALGORITHM

1. **Assigning of priority-based weights**: Unit weights are assigned to the possible grammars on the basis of presence of concepts, placeholders and keywords. In case of keywords, we have created a list of reject words (articles, prepositions) and added them to our custom resource folder. The keywords which are not present in the reject word list are assigned value of 1 whereas the keywords which occur in the name of a grammar are given weights of 2 (Reject words present in the grammar names are removed before checking for the occurrence of keywords). This step ensures the generality and robustness of the algorithm to the addition of new grammars.

2. **Generalization of open-phrase keywords**: We've not hard-coded any open-phrase keywords. The types of the open-phrase keywords have been loaded from PlaceHolderDetails.txt and if any further types of open-phrase placeholders are added in the future to PlaceHolderDetails.txt, they'll get incorporated into the program automatically (without the loss of generality).

3. **Handling of spelling-mistakes in keywords using cosine similarity**: We handle spelling mistakes in the keywords by cosine similarity of two keywords that have a difference in spelling. Two keywords with different spelling are treated as the same if they have cosine similarity greater than a minimum threshold which can be varied (currently set to 75%). We have implemented the function in python and attached as cosine similarity but were unable to incorporate into our program within the stipulated time of the hackathon.

4. **Deciding between grammars that have same weights**: Since the grammars having same weights are similar we propose to differentiate between them by searching for the presence of synonyms of the keywords that are exclusive to each of the conflicting grammar names.

5. **Inclusion of custom resources for improving accuracy**: We have included a dataset of more than 15,000 common and celebrity names. Also we've included around 500 place names (city names).

6. **Use of high performance pattern-matching algorithms**: In order to boost the running time of our algorithm, we have used the Knuth-Morris-Pratt (KMP) algorithm wherever possible as it is one of the fastest string pattern matching algorithms.

7. **Use of optimal data-structures**: The datastructures used are stl implementations of red-black trees and hashmaps which further boost the running time of our algorithm.

ERRATA

During the conception and integration of our algorithm into a wholesome model, we encountered the following inconsistencies in the files provided to us:

- The inclusion of proper nouns in the provided files was not in line with the sample test cases. To handle this, we have added a dataset of approximately 15000 proper nouns as a custom resource file.
- Multiple test cases included the mention of “London” as a place, whereas the same was not included in “places.txt” as a valid place input. To handle this, we have added a dataset of approximately 500 place names as a custom resource file.
- “Update”, naturally exists as a “tell” concept, but was not included in the text file “tell_concept.txt”. To overcome this problem, we could include “Update” into our concepts data structure, which we have not done so as to minimize hard-coding. Since we were not allowed to modify the files provided to us and were supposed to keep the code as generalized as possible, we have left Case #7 with an incomplete output. To generate the correct output, the file can be modified to include the valid string or “Update” can be hard-coded into the data structure.
- The `CreateAlarmByName` grammar included a spelling mistake and listed ‘alarm’ as ‘alram’. The `RenameEvent` grammar also included a spelling mistake listing ‘event’ as ‘even’.

ABSTRACT

Our algorithm uses an approach in which we assign *scalar scores* (representative of the probability that the corresponding grammar is the grammar of the input string) to each of the grammars based on the concepts/placeholders/keywords present in the input string and finally the grammar with the highest scalar score is written to the final output file. Based on the inner workings of our algorithm, detection of the values of the placeholders such as contact names and places, was trivially implemented by storing the location of the substring that was being used to detect their presence.

Handling open-phrases involved the detection of the grammar line (from the predicted grammar file) which most closely corresponded to the input string. After this step, a rejection-based scheme was implemented which removed all the concept phrases, placeholders such as contact names and places and reject-words such as articles and prepositions and extracted the open-phrases from the remaining string.

DATA STRUCTURES USED

(These datastructures are generated from the provided text files)

1. `concepts` : It is a dictionary (map) whose keys are the concept names and value is a vector which constitutes of all the possible strings that make up the concept. (This is provided. The concepts present in this datastructure are extracted from all the text files present in the 'Concepts' directory)

2. `placeholder` : It is a dictionary (map) whose keys are the placeholder names and value is a vector which constitutes of all the possible strings that make up the placeholder. (This is provided. The placeholders present in this datastructure are extracted from all the text files present in the 'Placeholders' directory)

3. `grammar` : It is a dictionary of grammar name and its values contain the corresponding concepts, placeholders and keywords.

4. `counters` : It is a dictionary (map) whose keys are the provided grammar names and its values are scalar scores representative of the probability that the corresponding grammar is the grammar of the input string. Thus the grammar with the highest scalar score is the required grammar.

5. `grammar_words` : This generates all the keywords present in all the grammars and stores in a map with keyword as the grammmar and value as the vector of keywords.

6. `placeholder_values` : It is a dictionary (map) whose keys are the placeholders found in the input string and the value is a vector that stores the value of the corresponding placeholder.

7. `grammar_lines` : It is a dictionary (map) whose keys are the name of the grammar and the values are the lines that appear in that particular grammar. This is used for identifying the grammar line which is the most similar to the input string.

8. `months` : It is a vector of strings that stores months. It is populated using the text file *month.txt* in the *DateTimeFormats* folder.

9. `days` : It is a vector of strings that stores days. It is populated using the text file *days.txt* in the *DateTimeFormats* folder.

10. `number_calendar` : It is a vector of strings that stores the possible values of days in a month (1-31).

11. `datetimeformats` : It is a vector of strings that stores all the possible formats for a placeholder of type `<dateTime>` . It is populated from the text file *datetimeformats.txt* in the *DateTimeFormats* folder.

12. `dateformats` : It is a vector of strings that stores all the possible formats for a placeholder of type `<date>` . It is populated from the text file *dateformats.txt* if the *DateTimeFormats* folder. This placeholder is then used to check for complex `<dateTime>` placeholders.

13. `year` : It is a vector of integers that is used to find four digit numbers from the input string.

These numbers are considered to be year values and are treated as a `<year>` placeholder. This placeholder is then used to check for complex `<dateTime>` placeholders.

CUSTOM RESOURCE FILES

1. `reject_words.txt` : It contains a collection of all common words that are not relevant enough to be used as keywords that might increase the probability of a certain grammar being related to a given input string. For example: *a*, *an*, *the*, *for*, *etc*.

2. `conjunctions.txt` : It contains a collection of all common conjunctions that might have been used in the input string to separate two different commands present in an input string. For example: and, meanwhile, also, etc.

3. `month.txt` : It contains a list of all months in both lowercase and camelcase.

4. `day.txt` : It contains a list of all possible days that can be found in an input string.

5. `dateformats.txt` : It contains a list of all possible date formats as given in the problem statement.

6. `datetimeformats.txt` : It contains a list of all possible day, date and time combinations that might constitute a `<dateTime>` placeholder in an input string. This was generated from the list given in the problem statement.

DESCRIPTION OF FUNCTIONS

1. `make_placeholder` , `make_grammar` , `make_concept` , `make_custom_resource` , `make_conjunctions` : These functions populate all the described datastructures by reading the provided text files.
2. `preKMP` , `KMP` : These functions implement the Knuth-Morris-Pratt string matching algorithm (this is one of the fastest string matching algorithms).
3. `is_concept_present` , `concepts_that_are_present` : These functions generate the vector of concepts which are present in the input string.
4. `is_placeholder_present` , `placeholders_that_are_present` : These functions generate the vector of placeholders which are present in the input string.
5. `find_grammar_scores_concept` , `find_grammar_scores_placeholder` , `find_grammar_scores_keywords` : These functions provide each of the grammars their corresponding scalar scores based on the presence of the concepts, placeholders and keywords in the input string.
6. `split` : In case of the presence of two commands in the same input string, this function splits the input string into two independent parts containing each of the two different commands, and returns them as a pair.
7. `command` , `give_command` : These functions work in unison to return the grammar that the input string corresponds to.

8. `return_placeholders` : This function returns a dictionary whose keys are the placeholders found in the input string and the values are the corresponding values of the placeholder extracted from the input string.
9. `give_grammar_line_score` : This function selects the most closely matched grammar-line from the predicted grammar file using the weights assigned to the concepts and placeholders present. This step helps in finding the values for the placeholders.
10. `make_day_date_time_vectors` : This function populates the `months` , `days` , `dateformats` , `datetimeformats` and `year` vectors. These vectors are required to find the location and values of the `<dateTime>` placeholders.
11. `replace` , `replace_year` , `replace_colon_time` , `replace_o_clock` , `handle_number_orders` : These functions take strings as input and return a string where the placeholder value is replaced by a generic placeholder tag within angular brackets which later helps to find the correct grammar. These functions also insert the actual value of the placeholder into the corresponding dictionary (map) for future reference.
12. `reconstruct_datetime` , `date_time_value` : After a `<dateTime>` placeholder is detected, these functions help to recreate the value of the `<dateTime>` placeholder using the values of the smaller placeholders which are already present in the dictionary we created earlier.
13. `date_time_string` : This function returns a string that replaces the dateTime placeholders (if any) with the generic `<dateTime>` tag within angular brackets to simplify the procedure of finding the correct grammar. It also stores the value of the `<dateTime>` placeholder in placeholder_values.