Data structures

parser_file_input	here path of the parser output text file is stored
prune_file_input	here path of the prune output text file is stored
wx_file_input	here path of the wx text file is stored
ner_file_input	here path of the ner output file is stored
parser_output_list	Store each element of each row of parser output as a list
parser_output_lines	Store all the lines of parser output
prune_output_list	Store each element of each row of prune output as a list
wx_output_list	Stores out of wx file as a list
wx_list	Contains every line of wx_file_input
root_word_dict_reverse	Its a dictionary where key is wx_word and value is root_word from prune output.
root_word_dict	Key is root_word and value is wx_word
suffix_dictionary	where key is word and value is suffix a.k.a 8th vector
wx_words_dictionary	here wx_words are value and keys are indexes.
temp_wx_words_dict	Same content as wx_words_dictionary .made temporary for loop usage
parser_output_dict	contains parser output list as value and index as key
NER_dict	Key = wx word and key = ner output
concept_list	

`info_list_final`	list containing important values from parser output file
`TAM_dictionary_list`	List containing every element of TAM dictionary
`group_list`	Getting dependency number of each word in the sentence

Row1

```
Import re
```

– This module is basically used for regular expressions; it contains functions for regular expression matching. Regular expression matching is a way of finding patterns in text.

```
parser_file_input="txt_files/parser-output.txt"
```

– This is a string variable in which the path of the parser output text file is stored. This code is likely used to store the file path or filename of a text file named "parser-output.txt" in the "txt_files" directory.

```
prune_file_input="txt_files/prune-output.txt"
```

– This is a string variable in which the path of the prune - output text file is stored.It is storing the file path or filename of a text file named "prune-output.txt" in the "txt_files" directory.

```
wx_file_input="txt_files/wx.txt"
```

– This is a string variable in which the path of the wx text file is stored. The string "txt_files/wx.txt" represents the path to a file called “wx.txt” located in a directory named “txt_files”.

```
ner_file_input="txt_files/ner_output"
```

– This is a string variable that contains the path of ner(Named Entity Recognition) output files, in the directory “txt_files”.

```
parser_output_list=[]
```

```
with open(parser_file_input,"r",encoding="UTF-8") as pf:
        parser_output_lines=pf.readlines
parser_output_lines.pop()
```

–Firstly, initialize an empty list named “parser_output_list”. The line “with open(parser_file_input, "r", encoding="UTF-8") as pf:” opens the file specified by the “parser_file_input” variable in read mode ("r"), and assigns the file object to the pf variable. The encoding parameter is set to UTF-8 to ensure that the file is read correctly.
– Here, “readlines()” is a method that is used to read all the lines of text from the parser output file, from the file object ‘pf’ and then assigns them to the variable “parser_output_lines”. Then “pop()” method will remove and return the last element of the given list.

```
for line in parser_output_lines:
        parser_output_list.append(line.split())
```

–The for loop iterates over the lines in the “parser output file”. The “split()” method splits the line into a list of strings, using the space character as the delimiter. The list of strings is then appended to the “parser_output_list” list. The output of split() is a list, therefore the list form is a 2D list.

```
prune_output_list=[]
f=open(prune_file_input,"r",encoding="UTF-8")
for line in f:
   prune_output_list.append(line.strip())
f.close()
```

–The “prune_output_list” list is initialized. The “open()” function opens the prune output file in read mode and assigns the file object to the f variable. The encoding parameter is set to “UTF-8” to ensure that the file is read correctly. The for loop iterates over the lines in the prune output file. The “strip()” method removes any whitespace characters from the beginning and end of the line. The line is then appended to the “prune_output_list” list. The “close()” function closes the file object.

```
wx_output_list=[]
with open(wx_file_input,"r",encoding="UTF-8") as pf:
    wx_list=pf.readlines()
    wx_output_list=wx_list[0].split()
```

–The "wx_output_list" list is initialized. The with statement opens the "wx file input" in read mode and assigns the file object to the pf variable. The encoding parameter is set to UTF-8 to ensure that the file is read correctly.
The "wx_list" variable is a list that contains the lines of text from the "wx output file"(only one line is there in wx output file). The first line of the wx output file is used to initialize the wx_output_list list.

```
root_word_dict_reverse={}
root_word_dict={}
for sent in prune_output_list:
    split_sent=sent.split(",")
    wx_word=split_sent[1]
    root_word=split_sent[2]
    root_word_dict_reverse[wx_word]=root_word
    root_word_dict[root_word]=wx_word
```

–This code is used to create a dictionary that maps wx words to their root words. The dictionary is created by iterating over the prune_output_list, which is a list of strings that contain the wx words and their root words.

For each string in the list, the code splits the string into two parts, using the comma as the delimiter. The first part of the string is the wx word, and the second part is the root word. The code then adds the wx word and its root word to the dictionary.

These dictionaries enable easy and efficient bidirectional lookups between wx_word and root_word, facilitating further analysis or processing based on these relationships.

```
suffix_dictionary={}
word=str()
for sent in prune_output_list:
  split_sent=sent.split(",")
  wx_word=split_sent[1]
  #print(split_sent)
  suffix=split_sent[5]
  suffix_dictionary[wx_word]=suffix
```

– This code is used to create a dictionary that maps wx words to their suffixes. The dictionary is created by iterating over the prune_output_list, which is a list of strings that contain the wx words and their suffixes. For each string in the list, the code splits the string into parts, using the comma as the delimiter. The second part(index 1) of the string is the wx word, and the sixth part(index 5) is the suffix. The code then adds the wx word(key) and its suffix(value) to the dictionary.

```
wx_words_dictionary_new={}
index=1
for words in wx_output_list:
    wx_words_dictionary_new[words]=index
    index+=1
temp_wx_words_dict=wx_words_dictionary_new
```

– The code you provided creates a dictionary called `wx_words_dictionary` that maps integers to words. The first line of code creates an empty dictionary, and the second line uses a for loop to iterate over the list of words in `wx_output_list`. For each word in the list, the code adds the word to the dictionary with the key being the index of the word in the list. The third line assigns the value of `wx_words_dictionary` to the variable `temp_wx_words_dict`, we made this variable for loop usage.

```
parser_output_dict={}
index=1
for par_value in parser_output_list:
```

```
    parser_output_dict[index]=par_value
    index+=1
```

– here we are creating a dictionary for parser_output_list where parser_output_list is the value and index is our key.

```
NER_dict={}
inx=0
y=open(ner_file_input,"r")
for line in y:
    inx+=1
    token=line.strip()
    if token.split("\t")[1]!="O":
        wx_word=wx_words_dictionary[inx]
        #print(wx_word)
        NER_dict[wx_word]=token.split("\t")[1]
```

– The code you provided creates a dictionary called `NER_dict` that maps words to their NER tags. the second line of the code uses a for loop to iterate over the lines in the file `ner_file_input`. For each line in the file, the code splits the line into tokens using the `\t` character as a delimiter. If the first token in the line is not equal to "O", then the code extracts the word from the dictionary `wx_words_dictionary`(because otherwise we get input in utf format) and adds it to the dictionary `NER_dict` with the NER tag as the value.

```
concept_list=[]
used_root_word=set()
updated_root_word={}
info_list_final=[] #list containing important values from parser output file
for line in parser_output_list:
  info_list_temp=[]
  word_index=int(line[0])
  pos_tag=line[3]
  class_index=line[6]
```

```
    word_info=line[7]
    info_list_temp.append(word_index)
    info_list_temp.append(pos_tag)
    info_list_temp.append(class_index)
    info_list_temp.append(word_info)
    info_list_final.append(info_list_temp)
```

– The code you provided is a Python program that reads the output of a parser and extracts the important information from it.

The program first creates an empty list called concept_list. This list will be used to store the concepts that are extracted from the parser output.The program then creates two empty sets called used_root_word and updated_root_word. The used_root_word set will be used to keep track of the root words that have already been used, and the updated_root_word dictionary will be used to store the updated root words.

The program then creates an empty list called info_list_final. This list will be used to store the important values from the parser output. The program then loops through the lines in the parser output list.

For each line, the program creates an empty list called info_list_temp. The program then appends the word index, the part-of-speech tag, the class index, and the word information to the info_list_temp list. The program then appends the info_list_temp list to the info_list_final list.

```
f=open("TAM-num-per-details.tsv.wx","r")
TAM_dictionary_list=[]
data=f.readlines()
for line in data:
    line=line.split("\t")
    line=[s.strip() for s in line]
    TAM_dictionary_list.append(line)
```

```
for line in TAM_dictionary_list:
    line.pop(0)
```

– This code reads data from a file named "TAM-num-per-details.tsv.wx" and creates a list of lists called TAM_dictionary_list to store the processed data. It splits each line of the file using tabs as the delimiter, removes leading and trailing whitespace from the elements(using strip), and appends the modified lines as sublists to TAM_dictionary_list. The first element (column) is then removed from each sublist because 1st and 2nd element is repeated.
The purpose of this code is to organize and clean the data from the file, preparing it for further analysis or manipulation in the subsequent parts of the program.

```
def search_concept(search_word):
    key=0
    for word in concept_dictionary_list:
        if word==search_word:
            key=1
            break
    if key==0:
            print("Warning:{} not found in dictionary.".format(search_word))
```

– The search_concept(search_word) function is designed to find a specific word (search_word) in the concept_dictionary_list. It iterates through each word in the list, and if a match is found, it sets the variable key to 1 and stops the loop using the break statement. If no match is found, the key remains 0, indicating that the word is not present in the dictionary.

#To print 1st row of USR which is the Original Sentence

```
def get_row1():
    row1="#"
    file_temp=open("bh-2","r",encoding="UTF-8")
    sent_temp=file_temp.readline()
    file_temp.close()
```

```
    row1=row1+""+sent_temp
    row1=row1.strip()
    return row1
```

– The code opens a file named "bh-2" in read mode ("r") with the specified encoding ("UTF-8"). The “open()” function returns a file object that allows us to interact with the file.The “readline()” method is used to read a single line from the file. It reads the first line of the file and assigns it to the variable “sent_temp”.

The “close()” method is called to close the file and free up system resources. The line “row1 = row1 + "" + sent_temp” appends the contents of “sent_temp” to the existing value of “row1” (assuming `row1` is a string variable).

The “strip()” method is then called on “row1” to remove any leading or trailing whitespace characters and returns it.

```
group_list=[]
index=len(parser_output_list)
for val in range(index):
     if val==index-1 or parser_output_list[val][7]=="pof" or
parser_output_list[val][7]=="pof__cn":
        group_list.append("0")
        #print(parser_output_list[val][1])
    elif parser_output_list[val][7]=="lwg__psp":
        group_list.append("-1")
        #print(parser_output_list[val][1])
    else:
        group_list.append("1")
        #print(parser_output_list[val][1])
```

– It initializes an empty list named “group_list” to store the results.The variable “index” is assigned the length of “parser_output_list” which represents the number of elements in that list.

The code then enters a loop that iterates over “range(index)”. This loop will iterate from 0 to `index-1`.Inside the loop, the code checks various conditions to determine the value to append to `group_list`.

If “val” is equal to `index - 1` (the last index) or if “parser_output_list[val][7]” (the 8th element in the sublist) is equal to either "pof" or "pof__cn", the value "0" is appended to `group_list`. If “parser_output_list[val][7]” is equal to "lwg__psp", the value "-1" is appended to `group_list`. If none of the above conditions are met, the value "1" is appended to `group_list`.

Row 2: Row 2 of USR is Concept. for every element in wx_output_list,check its POS TAG. The third column in parser_output_list[][3] is POS TAG,index from 0

```
already_visited={}
VM_already_visited={}
def for_VM_no__1(row_2):
    for word in range(len(row_2)):
```

```
        if "+" in row_2[word]:
            ini_part=row_2[word].split("+",1)[0].strip("_1")

            final_part=row_2[word].split("+",1)[1]
            #print("finalpart:",final_part)
            row_2[word]=ini_part+"+"+final_part
    return row_2
```

– The code you have provided is a function that takes a list of strings as input and returns a new list of strings with the same elements, but with the "_1" suffix removed from the first element of each string.

The function works by iterating over the list of strings, and for each string, splitting it into two parts at the first occurrence of the "+" character. The first part(ini_part) is then stripped of the "_1" suffix, and the two parts(final_part) are then concatenated together with a "+" character between them. The new string is then assigned to the corresponding element in the output list.

The function returns the output list.

```
def get_8th_vector(word):
    vector_8th=suffix_dictionary[word]
    return vector_8th
```

– The given function get_8th_vector(word) takes a word as input and retrieves the corresponding vector from a dictionary called suffix_dictionary. It then returns the obtained vector.

```
def get_root_word(word):#updated
    root_word=root_word_dict_reverse[word]
    return root_word
```

– The function get_root_word(word) takes a word as input and retrieves the corresponding root word from a dictionary called root_word_dict_reverse. It then returns the obtained root word as the output.

```
def for_handling_nnc_tag_or_pof(word,class_index):
    root_word=get_root_word(word)
    class_word=wx_words_dictionary[class_index]
    already_visited[class_word]=1
    if class_word in VM_already_visited:
        root_word_class=VM_already_visited[class_word]
        concept_list.remove(root_word_class)
    else:
        root_word_class=get_root_word(class_word)
    if root_word_class in used_root_word:
        root_word_class=updated_root_word[root_word_class]

concept_list.remove(updated_root_word[root_word_class])
    else:
        used_root_word.add(root_word_class)

updated_root_word[root_word_class]=root_word+"+"+root_word_cl
ass
    final_word=root_word+"_1"+"+"+root_word_class+"_1"
    VM_already_visited[class_word]=final_word
    return final_word
```

– The given function, for_handling_nnc_tag_or_pof(word, class_index), handles the processing of a given word and its associated class word based on specific conditions. It first retrieves the root word for the input word and the class word for the input class_index. Then, it checks if the class word has already been visited; if so, it removes its corresponding root word from the concept_list. Next, it checks if the root word has been used before; if it has, it generates an updated version of the root word and removes the original version from the concept_list. If the root word is new, it adds it to the set of used root words and creates an updated version.

The final word is formed by combining the root word and the class word with appropriate suffixes, and it is stored in a dictionary called VM_already_visited. The function returns the final word as the output.

```
def identify_vb(concept_list):
    tam_list_row2=[]
    for concept in concept_list:
        if "-" in concept:
            tam_list_row2.append(concept)
    return tam_list_row2
```

– The function identify_vb(concept_list) takes a list of concepts as input. It looks through each concept in the list, and if a concept contains the "-" character, it means it is a verb (or a concept related to verbs). The function then collects all such verb-related concepts and returns them in a new list called tam_list_row2.

```
def word_search_from_end(search_word):

    if "_1" in search_word:
        search_word=search_word.strip("_1")
        search_word=root_word_dict[search_word]

    match_key_list=[]
    matched_tams={}
    real_tam_search=search_word
    for line in TAM_dictionary_list:

        for value in line:
            if real_tam_search.endswith(value) and  value!=""
:
                key=value
                line0_length=len(line[0])
                if key not in matched_tams.keys():
                    matched_tams[value]=line[0]
```

```
                else:
                                                if  line0_length  >
len(matched_tams[value]):
                        matched_tams[value]=line[0]
    longest_key=int(0)
    longest_key_value=int(0)
    key_tam=None
    for key in matched_tams:
        key_temp_length=len(key)
        if key_temp_length>longest_key:
            longest_key_value=key
            longest_key=key_temp_length
    try:
        if longest_key_value !=0:
            key_tam=matched_tams[longest_key_value]

        return key_tam
    except Exception as e:
        print(e)
```

– The word_search_from_end(search_word) function aims to find the most suitable matching word from a dictionary based on the input search_word. The function first checks if search_word ends with the suffix "_1" and converts it to its corresponding root word using a dictionary called root_word_dict. It then iterates through a list of dictionary entries (TAM_dictionary_list) and searches for words that match the search_word from their ends. When a match is found, and the matched word is not empty, the corresponding dictionary entry's key is added to a new dictionary called matched_tams. Subsequently, the function identifies the longest matching key from matched_tams, which represents the most appropriate word. The function attempts to return the value associated with this key (best matching word) as the output. In case of any exceptions, it prints the error message. Overall, the function helps

determine the best matching word from the dictionary, based on the input search_word by searching for matches from the end of the words in the dictionary.

```
def search_tam_row2(concept_list):
    tam_list_row2=identify_vb(concept_list)

    for concept_with_hyphen in tam_list_row2:
              if concept_with_hyphen.split("-")[1]=="":
            real_tam_temp="0"
        else:
            real_tam_temp=concept_with_hyphen.split("-")[1]
        #print(real_tam_temp)
        if real_tam_temp=="0":
            root_concept=concept_with_hyphen.split("-")[0]
            #print(root_concept)
            search_word=root_concept.strip("_1")
            if "+" in root_concept:
                search_word=root_concept.split("+")[1]
            if "_1" in search_word:
                search_word=search_word.strip("_1")
            search_word=root_word_dict[search_word]
            #print("sw in row2:",search_word)

            key_tam=word_search_from_end(search_word)
                      if key_tam !=None:
                final_concept=root_concept+"-"+key_tam
            else:
                final_concept=root_concept
            #print(final_concept)
        else:
            #Word after hyphen in verb group
            root_concept=concept_with_hyphen.split("-")[0]
```

```
                    root_concept=root_concept.strip("_1")
            if "+" in root_concept:
                root_concept=root_concept.split("+")[1]
            if "_1" in root_concept:
                root_concept=root_concept.strip("_1")

wx_word_for_root_concept=root_word_dict[root_concept]

            for char in real_tam_temp:
                if char=="_":

real_tam_temp=real_tam_temp.replace(char," ")

real_tam_search=wx_word_for_root_concept+""+real_tam_temp

key_tam=word_search_from_end(real_tam_search)

            init_part=concept_with_hyphen.split("-")[0]
            if key_tam is not None:
                final_concept=init_part+"-"+key_tam
            else:
                final_concept=init_part
                                concept_list=list(map(lambda
x:x.replace(concept_with_hyphen,final_concept),concept_list))
        #print(concept_list)

    return concept_list
```

– The function search_tam_row2(concept_list) is designed to search for specific types of words in the input list concept_list and make modifications to the list based on the search results. It first identifies verbs or verb-related concepts in the list using the identify_vb function, and stores these verb-related concepts in a new list called tam_list_row2. For each word in tam_list_row2, the function checks if it contains a hyphen. If it does, it separates the word

into two parts: the root concept (before the hyphen) and the TAM information (after the hyphen). Depending on the presence or absence of TAM information, the function performs different steps. If there is no TAM information (TAM is "0"), it finds the corresponding root word from a dictionary called root_word_dict and searches for a matching TAM word using the word_search_from_end function. If a match is found, it combines the root concept and the matched TAM to form a new final concept. On the other hand, if there is TAM information, the function forms different versions of the combined concept by replacing underscores with spaces in the TAM information and searches for matching TAM words for each variation using word_search_from_end. The best matching word is then used to form the final concept. The function repeats this process for each word with a hyphen in tam_list_row2, updating the concept_list as it goes. Finally, the modified concept_list is returned as the output, containing the updated concepts based on the search results and TAM information.

```
def pronouns_to_replace(concept_list):
    concept_list_temp=concept_list

category_1=["wuma","wumhArA","wumako","wuJe","wU","wuJako","A
pa"]

category_2=["mEM","hama","hamArA","hamako","hameM","muJe","mu
Jako"]
    category_3=["Ji"]
    for index in range(len(concept_list_temp)):
        word=concept_list_temp[index]
        if "+" in word:
            continue
        else:
            if word in category_1:
                concept_list_temp[index]="addressee"
            elif word in category_2:
                concept_list_temp[index]="speaker"
            elif word in category_3:
                concept_list_temp[index]="respect"
```

```
            else:
                continue
    return concept_list_temp
```

– The given code defines a function called pronouns_to_replace, which takes a list of words called concept_list as input. It creates a temporary copy of the input list called concept_list_temp. Then, the function iterates through each word in the concept_list_temp and checks if it contains the "+" symbol. If it does, the function skips the word and continues to the next one. Otherwise, it compares each word with specific categories of pronouns, and if it matches any category, it replaces the word with a corresponding label. The three categories are "category_1," "category_2," and "category_3," which correspond to different types of pronouns. If a word matches any of these categories, it is replaced with the appropriate label ("addressee," "speaker," or "respect"). The function then returns the modified concept_list_temp with the pronouns replaced by their corresponding labels.

```
def get_row2():
  #pos_tag_dictionary={}#where key is wx_word and value is pos_tag
  #already_visited={}
counter=0
  for word in wx_output_list:

    word_index=wx_words_dictionary_new[word]
#word index of the word in wx_list
```

– Inside the function named “get_row2()”, a variable “counter” is initialized to 0. The code uses a for loop to iterate over each element (`word`) in the list “wx_output_list”. “word_index = wx_words_dictionary_new[word]”. This line retrieves the index of each word in “wx_output_list” from the dictionary “wx_words_dictionary_new”. The function “get_row2()” appears to be trying to find the indices of words in “wx_output_list” by looking them up in “wx_words_dictionary_new’ and storing those indices in the “word_index” variable.

```
    for line in info_list_final:
      if word_index in line:
        pos_tag=line[1]
```

```
            class_index=int(line[2])
            word_info=line[3]
```

– It checks whether the value of the variable “word_index” exists in the current “line”. If “word_index” is present in the “line”, the code performs the following assignments:
“pos_tag = line[1]”:- It assigns the value of the second element (index 1) in the “line” list to the variable “pos_tag”. “class_index = int(line[2])”: It assigns the value of the third element (index 2) in the “line” list to the variable “class_index”. “word_info = line[3]”: It assigns the value of the fourth element (index 3) in the `line` list to the variable “word_info”.

```
#main condition check begins here:

      if pos_tag=="PSP"  or pos_tag=="SYM" or pos_tag=="CC":
         continue
      elif pos_tag=="VM":
 #Do not add suffix for these words because we have to do TAM search on them and it creates a problem later.
         #print("this is the word:",word)
         root_word=get_root_word(word)
         vector_8th=get_8th_vector(word)
         if word in VM_already_visited:

            root_word=VM_already_visited[word]
            concept_list.remove(root_word)

            final_word=root_word+"-"
         else:
            final_word=root_word+"_1"+"-"
 for line in info_list_final[word_index:-1]: #updated simply vaux to vaux root
            pos_tag=line[1]
            if pos_tag=="VAUX":
               word_index=line[0]
               temp=wx_words_dictionary[word_index]
```

```
            #temp_root=root_word_dict_reverse[temp]
            final_word=final_word+"_"+temp
            #print(final_word)
            already_visited[temp]=1
        else:
            break

    VM_already_visited[word]=final_word #adding to the dictionary
    #sprint("this is final word:",final_word)
    concept_list.append(final_word)
elif pos_tag=="VAUX" :
    if word in already_visited:
        if already_visited[word]==1:
            continue
    else:
        root_word=get_root_word(word)
        concept_list.append(root_word+"_1")
elif pos_tag=="NNC" or word_info=="pof":

    final_word=for_handling_nnc_tag_or_pof(word,class_index)
    '''if pos_tag=="NN":
        f_right=final_word.split("+")[1]
        final_word_temp=final_word.split("+")[0].strip("_1")
        final_word=final_word_temp+"+"+f_right'''

    concept_list.append(final_word)
else:
    if word in already_visited:
        continue
    root_word=get_root_word(word)
    if pos_tag=="PRP" or pos_tag=="DEM" or pos_tag=="NNP" :
        concept_list.append(root_word)
```

```
        else:
            concept_list.append(root_word+"_1")
    counter+=1

#print(concept_list)
concept_list_final=search_tam_row2(concept_list)
#print("concept list final:",concept_list_final)
return concept_list_final
```

- The code uses an `if` statement to check the value of the variable `pos_tag`.If “pos_tag” is equal to “PSP", "SYM", or "CC", the `continue` statement is executed. . If “pos_tag’ is equal to “VM” , the code executes the subsequent block of code. It first calls a function “get_root_word(word)”, which appears to retrieve the root word for the given `word`. It then calls a function “get_8th_vector(word)”, which seems to retrieve the 8th vector for the given `word`. The code checks if the current `word` is present in the “VM_already_visited” dictionary. If the word is found in the dictionary, it sets the “root_word” to the value associated with the word in the dictionary. It then removes “root_word” from the “concept_list”. Finally, it assigns the value of “root_word + "-"” to the `final_word` variable.

If the `word` is not found in the “VM_already_visited” dictionary, it assigns the value of “root_word + "_1" + "-"” to the `final_word` variable.

The code uses a `for` loop to iterate through elements (`line`) in the “info_list_final” list, starting from the “word_index” position and ending at the penultimate index (index `-1`).

For each `line`, it extracts the “pos_tag” from the second element (index 1) of the `line` list. If `pos_tag` is equal to “VAUX", it performs the following operations: It updates “word_index” to the value from the first element (index 0) of the `line` list.

It retrieves the value associated with the “word_index” in the “wx_words_dictionary” and assigns it to the variable `temp`. It appends “"_" + temp” to the existing value of “final_word”. It sets the value `1` in the “already_visited” dictionary with the key `temp`.

After the above loop, it adds the current “final_word” to the “VM_already_visited” dictionary with `word` as the key.It also appends the “final_word” to the “concept_list”.

If “pos_tag” is `"VAUX"`, it checks if `word` is present in the `already_visited` dictionary. If it is present and its value is `1`, it continues to the next iteration (skipping the subsequent code for the current `word`). If it is not present or its value is not `1`, it performs the following operations: It retrieves the `root_word` for the current `word` using the `get_root_word()` function. It appends the `root_word + "_1"` to the

`concept_list`. If ‘pos_tag` is `"NNC"` or `word_info` is `"pof"`, it calls the function `for_handling_nnc_tag_or_pof(word, class_index)` to get the `final_word`.
It appends the `final_word` to the `concept_list`. If none of the above conditions are satisfied (i.e., `pos_tag` is not `"VAUX"`, `"NNC"`, or `"pof"`), it checks if the `word` is already visited in the `already_visited` dictionary.If it is already visited, it continues to the next iteration. If the `word` is not already visited, it performs the following operations: It retrieves the `root_word` for the current `word` using the `get_root_word()` function. If `pos_tag` is `"PRP"`, `"DEM"`, or `"NNP"`, it appends the `root_word` to the `concept_list`. Otherwise, it appends the `root_word + "_1"` to the `concept_list`.
After all the iterations, it increments the `counter` variable by 1.
The code calls the function `search_tam_row2(concept_list)` with `concept_list` as an argument and assigns the result to `concept_list_final`. Finally, it returns the `concept_list_final` from the function.

Row 3 of USR:Index for concepts

```
def get_row3(concept_list):
    index_for_concepts=[]
    for ind in range(len(concept_list)):
        index_for_concepts.append(ind+1)
    return index_for_concepts
```

–The function “get_row3” is defined, which takes “concept_list” as a parameter. An empty list named “index_for_concepts” is initialized to store the indices of elements. The code enters a loop that iterates over the range of the length of “concept_list’. For each iteration, the current index (`ind`) is incremented by 1 and appended to the “index_for_concepts” list. After all elements in “concept_list” have been processed, the function returns the “index_for_concepts” list.
This function generates a list of indices starting from 1 and ending at the length of the “concept_list”.

Row 4 of USR:Semantic category of Nouns

```
def get_row4(row_2):
    sem_category_list=[]
    if len(NER_dict)==0:
        for concept in row_2:

            sem_category_list.append("")
    else:
        for concept in row_2:
            flag=0
            for word in NER_dict:
                if word in concept:
                    flag=1
                    if NER_dict[word] =="B-PER":
                        ner_val="per"
                    elif NER_dict[word]=="B-LOC":
                        ner_val="loc"
                             elif NER_dict[word]=="B-ORG" or
NER_dict[word]=="I-ORG":
                        ner_val="org"
                    sem_category_list.append(ner_val)
            if flag==0:
                sem_category_list.append("")
```

This function "get_row4" that takes an input parameter called "row_2." This function aims to determine the semantic category of each noun in row 2 based on a dictionary called NER_dict, which presumably contains named entity recognition (NER) labels for certain words.

The function first initializes an empty list called "sem_category_list," which will store the semantic categories of the nouns. It then checks if the NER_dict is empty. If it is, the function iterates through each concept in row_2 and appends an empty string to the sem_category_list.

If the NER_dict is not empty, the function proceeds to iterate through each concept in row_2. For each concept, it sets a flag variable to 0. Then, it iterates through the words in the NER_dict. If a word from the NER_dict is found within the concept, the flag is set to 1, and the function determines the semantic category based on the value associated with the word in the NER_dict.

The semantic category is determined as follows: If the NER_dict value for the word is "B-PER," the semantic category is set to "per" (indicating a person). If the value is "B-LOC," the category is set to "loc" (indicating a location). If the value is "B-ORG" or "I-ORG," the category is set to "org" (indicating an organization). The determined semantic category is then appended to the sem_category_list.

Finally, if the flag is still 0 after iterating through all the words in the NER_dict, indicating that no match was found in the concept, an empty string is appended to the sem_category_list.

Overall, this code assigns semantic categories to nouns in row 2 based on a dictionary of NER labels. It uses the NER_dict to check for matches between words in row 2 and the dictionary, and then assigns the appropriate category based on the NER label. The resulting semantic categories are stored in the sem_category_list.

row_5_temporary is a list containing words and commas which we need to check

```
        for word in row_5_temp:
        if word==",":
            gnp_list_temp.append("")
        else:
            for line in prune_output_list:
                if word == line.split(",")[1]:
                    #print("line",line)
                    gender=line.split(",")[3]
                    number=line.split(",")[4]
                    person=line.split(",")[5]
                    if gender=="unk":
```

```
                        gender="-"
                    if number=="unk":
                        number="-"
                    if person=="unk":
                        person="-"
                gnp_list_temp.append("["+""+gender+" "+number+"
"+person+"]")

    return gnp_list_temp
```

It defines a function called `process_row_5` that takes two parameters: `row_5_temp` and `prune_output_list`. Inside the function, it initializes an empty list called `gnp_list_temp`, which will store the processed results. The code then enters a loop that iterates over each word in `row_5_temp`. If the current word is a comma (","), it appends an empty string to `gnp_list_temp`. If the current word is not a comma, it enters another loop that iterates over each line in `prune_output_list`.It checks if the word matches the second element in the comma-separated line (`line.split(",")[1]`). If there's a match, it extracts the gender, number, and person information from the line based on specific indices (3, 4, 5). If any of these extracted values are "unk" (unknown), it replaces them with a hyphen ("-"). After processing the data in `prune_output_list` for the current word, it appends a formatted string to `gnp_list_temp`.The string contains gender, number, and person values enclosed in square brackets. Finally, the function returns the `gnp_list_temp`, which will contain the processed data based on the conditions described. The function can be used to process specific data and populate `gnp_list_temp` with gender, number, and person information for each word in `row_5_temp`.

Row 6: dependency:

```
def get_row2_index(word):
    corr_index=0
    matched_concepts={}
    counter=1
    for concept_final in row_2:
        temp_list=concept_final.split("+")
```

```
        matched_concepts[counter]=temp_list
        counter+=1
```

– The given code defines a function called get_row2_index(word), which appears to process a list called row_2. The function takes a single argument word, presumably a string. It initializes three variables: corr_index set to 0, matched_concepts initialized as an empty dictionary, and counter set to 1.

The code then iterates through each element in the list row_2, which contains strings with a specific format (separated by "+"). Each element is split using the "+" character, and the resulting substrings are stored in a temporary list called temp_list.

The matched_concepts dictionary is updated with the counter as the key and temp_list as the value. This effectively maps each element of row_2 to its corresponding index in the dictionary.

```
for key,value in matched_concepts.items():
    for root_words in value:
        if "-" in root_words:
            root_words=root_words.split("-")[0]
        if "_1" in root_words:
            root_words=root_words.strip("_1")
        if word==root_words:
            corr_index=key
```

– The code snippet is part of a function that tries to find a specific word in a list called row_2. It first creates a dictionary called matched_concepts, where it maps each element in row_2 to its index in the list and the words obtained by splitting the element using the "+" symbol.

The snippet then checks each word in the matched_concepts dictionary. It removes any part after the "-" symbol and strips the "_1" suffix from each word to simplify them. Then, it compares each simplified word with the given word.

If there is a match between the given word and any of the simplified words in matched_concepts, it means the word is found in the row_2 list. In that case, the corresponding index of the word in row_2 is stored in the variable corr_index.

```
def get_row6(row_2):

    row_6=[]
    row2_iter=[]
    row2_wx_index_iter=[]
    class_word_index_list=[]
    class_word_index_dict={}
    class_word_list=[]
    root_word_from_wx=[]
    dependency_col7_list=[]
    correct_index_list=[]
    pos_tag_list=[]
```

– This code snippet defines a function called get_row6 that takes a single argument row_2. It initializes several empty lists and dictionaries: row_6, row2_iter, row2_wx_index_iter, class_word_index_list, class_word_index_dict, class_word_list, root_word_from_wx, dependency_col7_list, correct_index_list, and pos_tag_list.

```
for concepts in row_2:
        if "+" in concepts:
            concepts=concepts.split("+")[1]
        if "-" in concepts:
            concepts=concepts.split("-")[0]
        concepts=concepts.split("_")[0]
        row2_iter.append(concepts)
```

– This code snippet processes each element in the `row_2` list, removing any part after "+" and any part after "-", and only keeping the first part before "_". The simplified words are then stored in the `row2_iter` list.

```
 for root_word in row2_iter:
     wx_word=root_word_dict.get(root_word)

     wx_word_inx=wx_words_dictionary_new[wx_word]

     row2_wx_index_iter.append(wx_word_inx)
```

– This code snippet processes a list of simplified words, row2_iter, derived from another list called row_2. For each word in row2_iter, it looks up a related word in a dictionary called root_word_dict. The related word, referred to as wx_word, is then used to find its corresponding index in a different dictionary named wx_words_dictionary_new. The obtained indices are collected and stored in the row2_wx_index_iter list.

```
 for index_value in row2_wx_index_iter:
     par_value=parser_output_dict[index_value]
     class_word_index=int(par_value[6])
     pos_tag_value=par_value[3]
     dependency=par_value[7]
     class_word_index_list.append(class_word_index)
     dependency_col7_list.append(dependency)
     pos_tag_list.append(pos_tag_value)
```

– This code snippet takes a list of indices, row2_wx_index_iter, and uses them to access corresponding values from a dictionary called parser_output_dict. These values contain various pieces of information, such as part-of-speech tags, dependencies, and class word indices. The code extracts specific elements from these values and stores them in separate lists: class_word_index_list, pos_tag_list, and dependency_col7_list.

```
for index_6 in class_word_index_list:
     #do something about index_6==0 here
     if index_6==0:
```

```
        class_word_list.append(0)
    else:
        class_word_list.append(wx_words_dictionary[index_6])
```

– The code checks if the index value is equal to 0 using the condition `if index_6 == 0`. If the index value is 0, it means there is no corresponding class word, so the value 0 is appended to the `class_word_list` to represent this absence of a class word. If the index value is not 0, it means there is a valid class word associated with the index. In this case, the code retrieves the class word from a dictionary called `wx_words_dictionary`, which maps indices to class words. The obtained class word is then appended to the `class_word_list`.

```
for word in class_word_list:
    if word==0:
        root_word_from_wx.append(0)
    else :
        key=root_word_dict_reverse[word]
        root_word_from_wx.append(key)
```

– The `class_word_list` is iterated through, and for each `word` in the list, the code checks if it is equal to 0, indicating the absence of a class word. If it is 0, the value 0 is appended to the `root_word_from_wx` list. Otherwise, the code finds the corresponding root word for the given class word by using a reverse dictionary lookup (`root_word_dict_reverse`) and appends the root word to the `root_word_from_wx` list. This process helps to create a new list `root_word_from_wx` with either the actual root words or 0 for class words that have no root word associations.

```
for word in root_word_from_wx:
    if word==0:
        correct_index_list.append(0)
        continue
    correct_index=get_row2_index(word)
    correct_index_list.append(correct_index)
```

– The root_word_from_wx list is iterated through, and for each word in the list, the code checks if it is equal to 0, which means it has no corresponding root word. If it is 0, the value 0 is appended to the correct_index_list, and the loop continues to the next word. Otherwise, the code calls a function called get_row2_index(word), passing the word as an argument, to find the correct index of the word in the row_2 list. The obtained correct index is then appended to the correct_index_list. This process helps to create a list correct_index_list with the correct indices for each root word or 0 if there is no root word association.

```
for val in range(len(dependency_col7_list)):
    if dependency_col7_list[val]=="main":
        row_6.insert(val,"0:main")
    else:
        #print(pos_tag_list[val])
        index_6a=str(correct_index_list[val])
        dependency_col7=dependency_col7_list[val]
        if  pos_tag_list[val]=="JJ":
            dependency_col7="mod"
        elif pos_tag_list[val]=="QC":
            dependency_col7="card"
        elif pos_tag_list[val]=="QO":
            dependency_col7="ord"
        elif dependency_col7=="lwg__neg":
            dependency_col7="neg"
        elif dependency_col7=="r6-k2":
            dependency_col7="k2"
        row_6.append(index_6a+":"+dependency_col7)
```

– In this code snippet, a loop iterates through the elements of dependency_col7_list and pos_tag_list. It modifies and constructs a new list called row_6 based on specific conditions. If the dependency value at a particular position is "main," it inserts "0:main" into row_6. Otherwise, it checks the part-of-speech tag and the dependency value and makes

appropriate modifications. For certain cases, like "JJ," "QC," "QO," "lwg__neg," or "r6-k2," it updates the dependency value accordingly. The final result in the form of "index_6a:dependency_col7" is appended to row_6. The process ensures that row_6 contains the correct dependency information for each element, considering the conditions and modifications applied during the loop.

Row 7:

```
def get_row_unk(row_2):
    comma_list=[]
    for x in range(len(row_2)):
        comma_list.append("")
    return comma_list
```

The code you provided is a function called "get_row_unk" that takes an input parameter called "row_2." This function creates a list called "comma_list" with the same length as row_2 and fills it with empty strings.It uses a for loop to iterate over the indices of row_2. For each index, it appends an empty string to the comma_list. Finally, the function returns the comma_list, which is a list of empty strings with the same length as row_2.

Row 10:Sentence type

```
def get_row10():
    sentence_type=[]
    if "nahI" in wx_output_list or "nahIM" in wx_output_list:
        sentence_type.append('negative')
    else:
        if "?" in wx_output_list:
            sentence_type.append("interrogative")
        elif "|" in wx_output_list:
            sentence_type.append("affirmative")
        elif "!" in wx_output_list:
```

```
            sentence_type.append("exclamatory")
    return sentence_type
```

The function "get_row10" determines the sentence type based on the contents of a list called "wx_output_list." It first initializes an empty list called "sentence_type" which will store the determined sentence type.

The function checks if either the string "nahI" or "nahIM" is present in the wx_output_list. If either of these strings is found, it appends the string 'negative' to the sentence_type list.

If the strings "nahI" or "nahIM" are not found in wx_output_list, the function proceeds to the else block. Here, it checks if the question mark "?" is present in wx_output_list. If it is, the function appends the string "interrogative" to the sentence_type list.

If "?" is not found, the function checks if the pipe symbol "|" is present in wx_output_list. If it is, the function appends the string "affirmative" to the sentence_type list.

If both "?" and "|" are not found, the function checks if the exclamation mark "!" is present in wx_output_list. If it is, the function appends the string "exclamatory" to the sentence_type list.

Finally, the function returns the sentence_type list, which contains the determined sentence type based on the contents of wx_output_list.

this code determines the sentence type (negative, interrogative, affirmative, or exclamatory) based on the contents of wx_output_list, and returns a list with the determined sentence type.

```
def get_warning(row_2):
    for concept in row_2:
        if "_1" in concept:
            if "+" in concept and "-" in concept:
```

```
search_word=concept.split("+")[1].split("-")[0]
                search_concept(search_word)
            elif "-" in concept:
                search_word=concept.split("-")[0]
                search_concept(search_word)
            else:
                search_concept(concept)
```

– The given Python function, get_warning(row_2), processes elements in the row_2 list to find relevant warning concepts. It iterates through each element and looks for occurrences of "_1" within them. If found, it performs specific checks based on the presence of "+" and "-". If both symbols are present, it extracts the substring between them and calls the search_concept function with that substring as an argument. If only "-" is present, it extracts the substring before it and uses it as the argument for search_concept. If neither "+" nor "-" is found, it directly calls search_concept with the original element. However, for the code to work correctly, the search_concept function should be defined elsewhere in the code.

Main Function:

```
if __name__=="__main__":
      #row2 copy is a newlist,a copy of older one just to
replace the pronouns.

        row_1=get_row1()
        row_2=get_row2()

        row_2_temp=row_2.copy()
        row_2_chg=pronouns_to_replace(row_2_temp)
        row_3=get_row3(row_2)
        row_4=get_row4(row_2)
```

```
        row_5=get_row5(row_2)
        row_6=get_row6(row_2)
        row_7=get_row_unk(row_2)
        row_8=get_row_unk(row_2)
        row_9=get_row_unk(row_2)
        row_10=get_row10()
        #get_warning(row_2)
        print(row_1)
        print(",".join(row_2_temp))
        print(",".join(map(str,row_3)))
        print(",".join(row_4))
        print(",".join(map(str,row_5)))
        print(",".join(row_6))
        print(",".join(row_7))
        print(",".join(row_8))
        print(",".join(row_9))
        print(",".join(row_10))
```

– The provided Python script contains a series of function calls and print statements. The script checks if it is being executed as the main module and then proceeds to retrieve data for `row_1` and `row_2`. It creates a temporary copy of `row_2`, replaces pronouns in the copy, and stores the modified version. The script then calls several functions to process data from `row_2` and generate `row_3`, `row_4`, `row_5`, `row_6`, `row_7`, `row_8`, and `row_9`. Additionally, it retrieves data for `row_10`. The actual functionality of these functions is not provided, as they seem to be defined elsewhere in the program. The script concludes by printing the results of these data processing steps using the `print` function.