UNIVERSITY OF CALIFORNIA

SANTA CRUZ

Larry Birnbaum's Conceptual Analyzer in Python

A project submitted in partial satisfaction

of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Shayan Salsabilian, Maithili Luktuke, Arka Pal

June 2023

The Master's Project of
Shayan Salsabilian, Maithili Luktuke,
and Arka Pal is approved:

_____

Professor Leilani Gilpin, Chair

_____

Professor Cormac Flanagan

_____

Peter Biehl
Vice Provost and Dean of Graduate Studies

Abstract

The Larry Birnbaum conceptual analyzer was one of the first real natural language interpreters to deal with conceptual dependencies. At the time, this research was groundbreaking and led to many of the current breakthroughs with explainable AI. In 2022, Jamie Macbeth converted the original conceptual analyzer from MLISP to Common Lisp. This document describes our work to convert Jamie Macbeth's Common Lisp version of the Larry Birnbaum's conceptual analyzer to Python. Through this process, we learned a great deal about how conceptual dependencies work within conceptual analyzers to break down the understanding of sentences.

I. Introduction

In the late 1970s, the AI group at Yale University made some advancements in the field of natural language processing. They developed a set of natural language interpreters that took surface language or sentences and turned them into conceptual dependency (CD) semantic notation. A conceptual analyzer (CA) was one of the systems that they used in order to semantically parse the language into conceptual dependencies. This conceptual analyzer was built by Larry Birnbaum in MLISP and it was based on ELI or English Language Interpreter which was developed by Chris Riesbeck.

Our goal in this project was to convert Larry Birnbaum's conceptual analyzer designed by Mark Burnstein and Jamie Macbeth to Python. The codebase that we used was written entirely in Common Lisp. However, the original implementation was in UCI Lisp code. This conversion was a difficult task, as both languages have vastly different syntax and expressiveness from one another. The following sections in this report talk about the literature that we reviewed in order to gain a better understanding of the topic, an overview of Lisp, our process and the struggles that we encountered throughout this project, a description of the Lisp code along with an explanation of the Python code, and lastly the results of our project showing the progress we made in converting the codebase of the conceptual analyzer to Python.

II. Literature Review

We studied various papers that focused on conceptual dependencies and conceptual analyzers to obtain a better understanding of the Lisp implementation. . Our research began with a paper that talks about conceptual dependency theory and the primitive acts of conceptual dependency. The basics of conceptual dependency is that an action is the basis of any proposition and a proposition is made up of conceptualizations. These consist of an actor, and action, and a set of events dependent on that action. The main insight of this theory is that there has to be a canonical form for meaning representations. This means that words are broken down into their conceptual parts and placed in meaning representations based on their conceptual roles. The concept of primitive acts was introduced to handle any similarities or overlap in meaning. Conceptual dependency representations use these primitive acts and the following list of acts shows a few examples and their descriptions:

- PTRANS: The transfer of physical location for an object. It requires actor, object, and direction.
- PROPEL: The application of a physical force to an object. If movement occurs because of PROPEL, then a PTRANS has taken place too if PROPEL was intended by the actor. It requires direction, object, and actor.
- INGEST: The taking in of an object by an animal. It requires actor, object, and direction.
- MOVE: A thing moving a part of its body or part of itself. It requires actor, object, and direction.

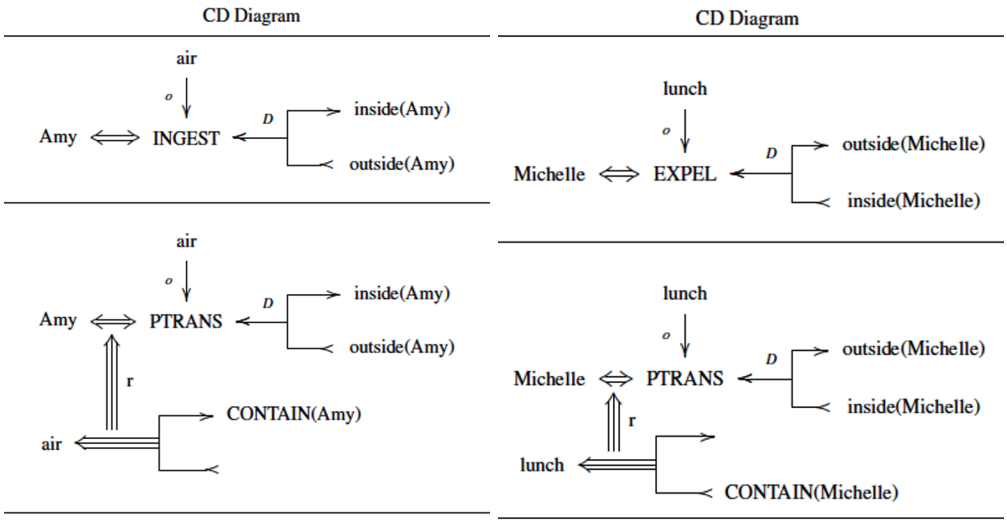
Some other primitive acts include GRASP, SPEAK, ATRANS, and EXPEL. The list of primitive acts that are given is not an exhaustive list and more can be created as needed. These acts in memory help organize the inference process. There are many kinds of possible primitives and they underlie the words of a language [1].

In another such paper [2], the authors are concerned with developing cognitive representations that enhance the abilities of self-supervised learning systems to learn language as a part of their exploration. One of the main challenges with cognitive representations is making the representations language-free. This is so that commonsense reasoning can be accomplished through the symbolic structures of mental imagery instead of relying on being expressed linguistically. They look into systems, called conceptual analyzers, that decompose language into language-free structures made up of primitives such as object permanence, spatial relationships, and movement. They are focused on improving the conceptual dependency representation system, its primitive decompositions, and its conceptual analyzer. They used the ProPara (Process Paragraphs) dataset that is made up of paragraphs describing biological, chemical, and physical processes from grade-school textbooks so complex concepts of society and thought don't pose challenges. The authors build a mental motion pictures representation system in order to meet the challenges posed by this dataset. The system has some important innovations such as using image schemas instead of CD primitives and decoupling containment relationships into separate primitives [2].

The meaning representation system that the authors devised is a mental imagery system consisting of world objects, spatial maps of the spatial relationships between objects, and event primitives that represent changes in objects' states or spatial relationship changes. The mental motion picture is a sequence of frame objects. Each frame consists of three mental maps: containment, position, and contact maps. Containment maps tell us when one object goes inside another for example when a pen is put into a pencil case. Position maps tell us the position of the objects with respect to the center of the earth. Contact maps record whether two objects in a story are touching each other. The frames are nodes of a linked list and are capable of inheriting objects, edges, and primitive acts from the frame that precedes it. Unless an object transforms, gets destroyed, or disappears, it will be copied from one frame to the next. Their system outputs a much more detailed analysis than other conceptual analyzers [2].

Their conceptual analyzer is an expectation-based analyzer based on an earlier model called English Language Interpreter (ELI). Expectation-based analyzers sequentially process input words, creating partial or incomplete non-linguistic conceptual structures that correspond to certain words. These structures serve as instructions for completing the overall structure using information from other words. Like ELI, expectations are represented as "requests" and the analyzer's state includes a stack for organizing and processing requests, along with global variables and registers that can be modified by triggered requests [2].

Their conceptual analyzer has a conceptual lexicon. This contains the imagery and expectation structure to instantiate the sequential word processing. It is essentially a packet that consists of a list of requests. While processing words, the analyzer finds the entry for it in the conceptual lexicon, then places the corresponding packet at the top of the stack of previously loaded packets. Then the packet is examined to see if any requests can be triggered and if so then that packet is popped off of the stack and executed. This cycle continues until no requests are triggered or the stack is empty thus making the analyzer move to the next word. When a word is seen, expectations as to what the next word will be are formed and that attribute given to each word. This is meant to be similar to the human thought process where if an expectation is satisfied, more expectations are formed in hopes that the subsequent words will satisfy those expectations [2]. The following two images show a graphic of the CD decomposition that show the conceptual primitive, the direction, object, etc [3].



From their work, the authors found a need to add another primitive called PSTOP, which represents the end of a PTRANS act and the resulting static situation. Their analyzer uses structures in the conceptual lexicon, which specifies as much information about the word it represents to create an understanding of a text. The authors also mention a limitation in how their design of the mental motion picture systems represented objects. Their current system can decompose situations that involve movement and containment, and these containment relationships may be used to infer shapes of objects. Since this inference can be wrong, the

authors speak about future work where new primitives would be designed to allow for the decompositions of shapes, relative sizes, distances, and orientations of objects [2].

People have also been working on explainable AI that deals with conceptual primitives decompositions. We looked into Gilpin et al. 's work [4], where they talk about how even though there have been many advances made in connecting text and perception, the computer-generated image captions still lack common sense. This led them to developing a reasonableness monitor. They created a system that uses dependency tracking, commonsense knowledge and conceptual primitives in order to explain whether a certain scene description is reasonable or not. Their work was meant to improve the automated systems that perform tasks such as image labeling and captioning. As these autonomous systems become a more prevalent part of our world, their common sense mistakes could prove dangerous. The authors' reasonableness monitoring system takes a natural language sentence as an input and outputs whether or not that sentence is reasonable or, along with an explanation in natural language, as to why it is reasonable. Their system first uses a commonsense knowledge base and constructs a frame around the abstracted conceptual primitives. To build the conceptualization, the system determines which primitives best represent the act described in the sentence. After this, the system can detect any violations that have occurred. Finally, based on what the violations were, the system gives its decision and provides an explanation as to why it gave that decision [4].

In order to build their reasonableness monitor and connect it to a preexisting perceptual system, the authors had to consider the following to make certain specific choices:

- The output of the perceptual system will give the input for the reasonableness monitor system.
- Perception needs to follow the rules of commonsense which can come from a commonsense knowledge base.
- Natural language can be used as an interface between common sense knowledge bases and perceptual systems.
- A major challenge with using natural language as an interface is language variation, several different ways to express the same concept.
- To model perception in the inner language, a reasoning system is required that utilizes knowledge of perceived objects and their typical interactions.
- In order for engineers and system designers to be able to interact with monitors successfully, the system has to be able to explain and verify what has been done with the core reasons and support for a reasonableness judgment [4].

The authors demonstrate how to evaluate the output of a machine perception algorithm, a natural language description, for reasonableness. This description would be put into a reasonableness monitor, transformed into a set of conceptual primitive frames, search for relevant knowledge in ConceptNet which is a semantic network and commonsense knowledge base, and provide an explanation for the judgment. After an input is given, a part-of-speech tagging is performed using the Python NLTK part-of-speech tagger and a regular expression parser to find noun, verb, object, and prepositional phrases. ConceptNet represents its knowledge

as a graph with the concept being the nodes and the relations between the concepts being edges. There are certain concept nodes that are selected to be anchors which in this case are person, plant, animal, object, vehicle, and weather since they fit their use case of autonomous vehicles. These are the best representatives of the non-linguistic primitives of conceptual dependency. The system attempts to anchor words with a concept in ConceptNet and flags any violations if they occur. Based on these violations, if no other information is given to solve the violation, then the statement is deemed unreasonable, and an explanation as to why is given as output. For this paper, the authors used six conceptual dependency primitives: PTRANS, PROPEL, MOVE, EXPEL, INGEST, and GRASP. They did, however, combine MOVE and PTRANS into one primitive called MOVE-PTRANS [4].

Their system demonstrated 82% accuracy on a set of 100 examples and was able to give human-readable explanations for the judgments that it made. The key goal of even having a monitoring system in the first place is to explain errors for better diagnostics; hence, problematic evidence can be used to make better decisions in the future. The system in this paper is designed forautonomous perception systems to determine the reasonability of the perception [4].

III. Lisp

Lisp is a functional programming language commonly used in artificial intelligence/natural language processing applications. The language uses fully parenthesized prefix notation. It is a language that consists solely of expressions. Lisp is a language that uses prefix notation. In general, there is no need for while or for loops in this language. A variation of lisp, Common Lisp, works well with language constructs, and is used to develop conceptual analyzers to break down phrases into conceptual dependencies using primitives. As lisp is a relatively old programming language it has a much more different syntax from the common programming languages used today.

In order to gain a better understanding of Lisp and how it converts to Python, we studied Peter Norvig's Lisp Interpreter which he coded in Python. Since Lisp is a language that is made up solely of expressions, there is no need to distinguish between statements or expressions. Numbers and symbols can be broken into pieces and are called atomic expressions. Characters that are usually considered to be operators in other languages such as "+" or "<" are considered as symbols in Lisp and are treated the same way that other symbols are treated. The rest of the code in Lisp is a list expression. This consists of an opening parenthesis, "(", followed by a zero or other expressions, and an ending parenthesis, ")". The first element of the list defines it. For example if the first element starts with a keyword such as: (if…..) then this is called a special form. There are several different keywords so the meaning of the special form depends on the keyword. If the first element starts with a non-keyword such as: (fn…..) then this is a function call. There are several different dialects of Lisp such as Common Lisp, used for real-world programming and Scheme, used for computer programming [5].

Lisp has several keywords and non-keywords, some of the most common ones that we encountered were the following:

- defun: defines a new function in the global environment
- let: creates new variable bindings in parallel. We can take this to mean variable declarations in Python
- cond: is a construct that permits branching. These can be taken as if, elif, else statements in Python
- setq: is a variable assignment statement
- if: allows the execution of a form to be dependent on a single test form

The long-standing lineage of Lisp programming languages includes Common Lisp. It stands out as a modern, compiled, high-performance, ANSI-standardized, and multi-paradigm language, alongside Scheme. It is a successor to MacLisp, strongly influenced by ZetaLisp and even Scheme to some extent. Common Lisp is known for its exceptional flexibility, robust support for object-oriented programming, and rapid prototyping capabilities. Sporting an extremely powerful macro system, Common Lisp allows for a flexible runtime environment that lets developers modify or debug running applications and tailor the language to their specific application. Since it is a multi-paradigm programming language, it allows developers to choose the approach along with the paradigm appropriate for their application domain [6].

Common Lisp is meant to fulfill the following goals: commonality, portability, consistency, expressiveness, compatibility, efficiency, power, and stability. It was designed to simplify writing programs without having to deal with machine-specific characteristics such as word length. It is has a wider variety of features which encompasses a vast array of data types and more complex control structures. There are many different implementations of Common Lisp, some by research groups in universities and some by companies selling those implementations commercially [7]. These are some of the reasons why the original project was coded in Common Lisp. Since Python is a very human-readable language, we were tasked with translating this code to Python.

IV. Process and Struggles

When we began this project, we read a few papers on conceptual analyzers and conceptual dependencies to gain an understanding of what they are and how they work. In Section II, the literature review, we talk in depth about the papers that we researched. After that, we were given the codebase of the conceptual analyzer that we would have to convert to Python. Since this codebase was entirely written in Lisp, we had to first get an understanding of how Lisp works before moving on to development. In order to do this, we studied Peter Norvig's Lisp to Python interpreter. While reading about this interpreter, we learned a lot about how Lisp works and the different types of keywords and non-keywords that can exist. He also provided his Python code for the interpreter on his website, which we downloaded and used to get more familiar with Lisp.To make sure we were understanding how Lisp works, we played around with his lis.py program along with the examples he had on his website [8].

Organizing the project was difficult with three different people. There was a lot of overlap at first, and attempts to work through Python collaborative notebooks proved messy

since a lot of files relied on each other. Eventually, we decided the best decision was to migrate our work over to github and create a living google document to organize our tasks and struggles. This helped streamline the process and allowed us to collectively get further in our work.

One of the first hurdles we ran into was getting the Lisp code to work. We did not just want to translate the code to Python without making sure that what we were doing was actually correct or not. In order to do this, we had to download and run the Lisp code on SBCL (Steel Bank Common Lisp) which is a high performance Common Lisp compiler. All three of us tried installing SBCL on our laptops multiple times unsuccessfully. There wasn't really any good documentation for troubleshooting either. After much research about the installation and errors that we were getting, we were successfully able to install SBCL.

One of the main struggles we ran into was our word not having any requests in its class instances when calling the make-request function. The confusion existed over where exactly crash-dic.lisp and dictionary-fns.lisp existed. Finally, through extensive debugging of the lisp code, we were able to determine that the database (crash-dic and dictionary-fns) is created during compile time. We solved this by creating our database at the start of the program and saving the requests in a string format.

Another struggle was determining the best way to represent property tags and cons inside our program. Maintaining the list structure would be easier from a design standpoint as it will more closely resemble the original lisp code, but dictionaries made more sense in terms of property tags as the property tags could be keys. In the end, we decided to go with the list implementation because it worked best with the build-c and build-m functions and current design for recursive con building.

A final struggle that we worked through was that the globals() keyword was mainly used in Python to create or find variables within the file using the string passed to it. We learned that if you use the globals() keyword to modify arrays in class instances the values will stay in new class instances. We solved this problem by using the none keyword instead and having a conditional for if it's the first value in the array versus appending to the array.

V. Code

- Key Terms
    - Request: a request is the main way that we determine how a word works within our specific system; it typically follows a conditional based format of clause and action.
    - Pool: a pool will hold all the requests for a word. There may be multiple requests based on the uniqueness of the word.
    - Con: This is the main way each word is defined. Cons will be inserted into each other and the final con should have a decently detailed explanation of how all the words work together in the sentence.
- Compile time in Lisp

- At compile time, the lisp code implements crash-dic.lisp and dictionary-fns.lisp.
- Crash-dic.lisp works as essentially the database for our conceptual analyzer.
  - Modern conceptual analyzers have large graph based databases and anchor words that define their functionality.
  - However, Larry Birnbaum's Conceptual analyzer used a rudimentary conditional based request system to determine how each word worked within the specific sentence.
  - This is part of the reason why it is only currently designed to work on 6 sentences.
  - Here is an example image of how a word is defined in crash-dic:

```
(defterm TWIN-ENGINE (ADJ)
  (REQ (t
        (:= str1 (build-con (*PP* :class (GROUP) :number (*num* number (*2*))
                                  :member (*PP* :class (structure) :type (*engine*)))))
        (ACTIVATE (REQ ((:= STR2 (if-find (and (feature c 'pp) (follows c str1))))
                        (fill-gap :has-part str2 str1)))))))
```

  - There is a conditional in this case just true, and actions that are evaluated based on that conditional.
  - In this case there will be two actions
    - Build-con will create a basic con defining what a twin engine is.
    - Activate is a function defined in request-fns.lisp, but essentially it will create a new request in the same format.
  - There is also an extra parameter at the top in this case adj (short for adjective) this attribute is mainly used to determine if we are in a noun phrase or not.
- Dictionary-fns.lisp takes that input and assigns the adjective parameter (in this example) to atts (short for attributes). It then assigns the request parameter to requests. Finally, it modifies the format of our requests a little to have a clause parentheses around the conditional and an action parentheses around the action so that it looks something like this:

```
(def A
  (:atts art)
  (:requests
   (request
    (clause (test t)
            (actions
             (:= str1 (build-con '(*indef*) nil nil))
             (activate
              (request
               (clause (test (:= str2 (if-find #'(lambda (foo)
                                                   (or (feature foo 'loc)
                                                       (feature foo 'pp))))))
                       (actions (fill-gap (add-gap '(ref) str2)
                                          str2 str1)))))))))))
```

- In our version, we simply implement this code as strings and run them at the start of the program. We chose strings to represent the precompile time code because they can be easily executed using the eval function whenever we wish and are in an easy to read format. We do not use dictionary-fns.py and instead specify the format and define the attributes in crash-dic.py like so:
  - 
```
def dic_a(art = ["art"]): #this is my best version so far may need to tweak later but follows his Test Action format and easyish
    req = ["request", "clause(test True)", "(actions(crash_dic.actions_a())"]
    atts = art
    return req, atts

def actions_a():
    wd = Global.find_class("a")
    wd.str1 = concept_fns.build_con(["*indef*"], [], [])
    new_bind = ["str1", wd.str1]
    idx = 0
    while(idx < len(Global.bindings)):
        if Global.bindings[idx] == "str1":
            Global.bindings[idx] = new_bind
            break
        idx=idx+1
    request_fns.activate([["request", "clause(test crash_dic.cond_a())", "actions(crash_dic.actions_a_1())"]]) # we want it as on

def cond_a():
    # getting the latest con should be the first c_list
    if(request_fns.if_find(request_fns.feature(Global.c_list[0], ['loc', '*PP*'])) and not Global.flagon("noun_group_flag")):
```
  - We have the conditionals, actions, and bindings split into different functions that will be evaluated when eval is called on each particular string (this will be done in control.py).
- On a high level, the program is broken down into these 6 functions:

```
;; remove request pools which no longer contain active requests
(clean-up-request-pools)
(consider-lexical-requests) ;; considers all if no lexical requests
(check-end-np) ;; moved COND below
;; activate requests associated with :word;
(activate-item-requests WORD) ;; was LEX
(consider-requests)
(check-begin-np) ;; moved COND below
```

- Clean-up-request-pools: This function will run through every pool and check to see if there are any active requests. If there are not, it removes the pool from the system.
- Consider-lexical-requests: This function has not been implemented in our version but is used for the specific scenario where one word is reliant on the next word in the system.
- Check-end-np: This checks the attribute of the current word and the next word against a number of conditionals to determine whether we are at the end of the noun phrase. If we are, it will remove the noun phrase group flag.
- Activate-item-requests: This will generate a request, build a pool, and place the generated request on that pool.
- Consider-requests: This will parse the request, evaluate the conditional, and do the action based on if the conditional is true.
- These functions are all run in a loop on each word in the system

In depth explanation of Python code:

- The code begins in main.py with the target sentence being fed to the CA function in control.py. We have only tested the program on the sentence: “a small twin-engine plane stuffed with marijuana crashed south of here yesterday”.
    - We specifically chose this sentence because it was the hardest of the test sentences and used the largest combination of words. Therefore, if this sentence worked there is a good chance that the rest of the test sentences would work with minimal adjustments.
- Inside the CA function
    - We split the input into a list at the spaces
    - We call init_ca()
        - init_ca() initializes a number of global variables
        - It also sets the requests and attributes of each word (defined as a class) as properties.
            - Ex: For the letter “A”

```
req, atts = crash_dic.dic_a()
a.atts = atts
a.requests = [req]
```

        - As you can see we run the crash_dic version of "a" and get the attributes and requests as a return. We then set those as property values of the class "a".
    - We get the next word each time using the get_next_item() function
    - Then in a loop, we call clean_up_request_pools()
        - This calls clean_up_special_pools, which then calls save_live_reqs with the string "Lexical_pools".
            - Save_live_reqs uses the function pool_reqs in Global.py. The goal of this function is to find the variable that matches the string given and return its value.
                - In this case, we get the global variable Lexical_pools value.
            - It then runs through all Lexical_requests for each pool and determines if any of them have an active flag. If it doesn't it is removed from the variable.
        - After calling clean_up_special_pools, we go through all the normal requests and call live_reqs on them.
            - Live_reqs works exactly the same as save_live_reqs except it's passed in the pool and does not get the value using pool_reqs.
    - After clean_up_request_pools, we call check_end_np (please note that consider_lexical_request has not been implemented and will be a decent share of the work for the next group)
        - This checks if the flag noun_group_flag is set (which it would have to be if we were ending a noun phrase) and if the function end_noun_phrase returns true.
            - End_noun_phrase calls begin_noun_phrase() with the next word.
                - Begin_noun_phrase checks if the next word has an attribute of adj, arg, name, noun, num, poss, or title1. If it does, it returns that attribute.
            - It then checks the attribute and n_p_record against a number of conditionals if any evaluate to true; it means we're still in a noun group and we return false.
                - We also modify n_p_record to be the unique list of atts and n_p_record.

        - If not, we remove the noun_group_flag, clear n_p_record, and return true.
    - If it returns true, we remove the noun_group_flag otherwise we return.
- We then call activate_item_requests.
    - This will call make_request and save the results in a variable called reqs (The results will be the names of the newly created requests).
        - The parameters passed into make_request will be the word and the request property of the class instance of the word.
            - This is the requests information specified in crash-dic.py specifically the req of functions like this:

```
def dic_a(art = ["art"]): #this is my best version so far may need to tweak later but follows his Test A
    req = ["request", "clause(test True)", "(actions(crash_dic.actions_a())"]
    atts = art
    return req, atts
```

        - Make_requests will run through all requests and call the gen_request function and append their values to our result variable, which we return.
            - Gen_request will generate a unique name for a request like (“REQ-A-1”) using the new_req function in macros.py
            - It will then create a new request class instance with that unique name using create_req in Global.py.
            - We will then remove the request and clause words from the request.
                - The body property of the request class instance will be set to the result of those changes.
            - We will also set the word property of the request to the current word, the bindings property of the request to the current bindings, and the active property of the request to true.
    - We then return all the way back to activate_item_requests and append our new active form requests to extra requests.
    - We then call build_pool.
        - Build_pool will call new_pool in macros.py, which will generate a unique name for the pool like “Pool-small-6”.
        - It will then create a new pool class instance with that name and set its value to the array of reqs.
    - We then call activate_pool.

            - This adds the new pool to the request_pools variable in Global.py if it's not already there.
    - Afterwards, we call consider_requests().
        - Consider_requests will call consider_latest_request if the noun_group_flag, otherwise it will call consider_all_request.
            - Consider_latest_requests will grab the first two pools in Global.request_pools, whereas consider_all_request will grab all requests.
            - Both will call consider_pool until all of the calls return false.
                - Consider_pool gets all the requests for the pool given then runs the consider function on them.
                    - Consider checks to see if the active flag is set.
                    - If it is we use the eval_test function to evaluate the conditional.
                    - If the conditional returns true we use eval_actions to do the action. This is usually building a con which will be explained in the concept-fns section and setting the active flag to none.
                    - If the active flag is not set it returns true, otherwise it returns false.
    - Finally, we call check_begin_np():
        - Which checks if the noun_group_flag is already set (if it is it's already begun).
        - If the noun_group_flag is set
            - We call begin_noun_phrase again on the next word and save its results in Global.n_p_records.
            - If Global.n_p_records has a value we add the flag.

Building a con

- Building a con is mainly done in concept-fns.py. It is usually one of the actions declared inside crash-dic.
- We start inside build_con, which calls add_con.
    - Add_con calls build_cd.
        - Build_cd calls make_cd.
        - Make_cd calls build_c.
            - Build_c and build_m both recursively call each other to create cons for each property tag. This would turn this

sentence:

```
(*PP* CLASS (GROUP) NUMBER (*NUM* NUMBER (*2*)) MEMBER
 (*PP* CLASS (STRUCTURE) TYPE (*ENGINE*))) and seen: NIL in Build-C
```

- Into this:

```
Adding CON11 = (*PP* :CLASS CON12 :NUMBER CON13 :MEMBER CON15)
```

- Each of the lists will have its own separate cons with values.
    - We then insert the new con in changed_cons.
- Afterwards, we go through all equivalences and substitute each of the cons values. We do the same with filler values.

Globals.py

- This file contains all global variables.
- All class instances for each word, request, and con.
- All flag searching, insertion, and deletion.
- Functions to create class instances, set class instance values, and remove class instance values.

Macros.py

- Functions for printing.
- Functions for determining unique names for req, pool, lex, and con.

Request_fns.py

- Add_con
    - Begins the con building process described in concept-fns.py
- Activate function
    - Calls activate1
        - Which calls make_requests in control.py to generate a new request based on the request parameter.
        - Add this request to the current pool.
        - Then it activates the pool.
- Feature function
    - This function checks to see if specific preds are in an obj.
    - It is usually used to determine if a con has certain members in conditionals of crash-dic.py
        - For example, in this call in crash-dic.py:

```
def cond_small():
    if request_fns.if_find(request_fns.feature(Global.c_list[0], ['*PP*'])):
```

        - We check if the latest con has the string “*PP*”.
- If-find function
    - This is used in conjunction with the feature, it checks that the con we grabbed is not embedded and can satisfy the conditional.

- If the embedded property is there then the con has already been used.
- Follows function
  - Checks to make sure that both cons are in the con list and are not the same value.
- Fill_gap function
  - Works in conjunction with set_gap to add a new value and property tag to the back of a con.
  - This allows our final con to fully analyze the sentence as it will insert previous cons into new cons based on conditionals.
- Set_gap function
  - Calls set_role_filler
    - Set_role_filler checks if the property tag is already set and if it is, it replaces the value.
    - Otherwise, it appends the new property tag and value.

VI. Results

This is the complete lisp output:

Current input: ((A SMALL TWIN ENGINE PLANE STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY))
New sentence is (A SMALL TWIN ENGINE PLANE STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY)

---------- Current word: A ---------- Phrase: (A) rest: (SMALL TWIN ENGINE PLANE STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) ACTIVATE ITEM REQUESTS for word A: (REQ A 1) REQ A 1 has fired Adding CON3 = (INDEF) REQ A 1 activating new requests: (REQ A 4) Begin noun group: Begin noun group:

---------- Current word: SMALL ---------- Phrase: (A SMALL) rest: (TWIN ENGINE PLANE STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) Begin noun group: ACTIVATE ITEM REQUESTS for word SMALL: (REQ SMALL 5) REQ SMALL 5 has fired Adding CON7 = (LTNORM) REQ SMALL 5 activating new requests: (REQ SMALL 8)

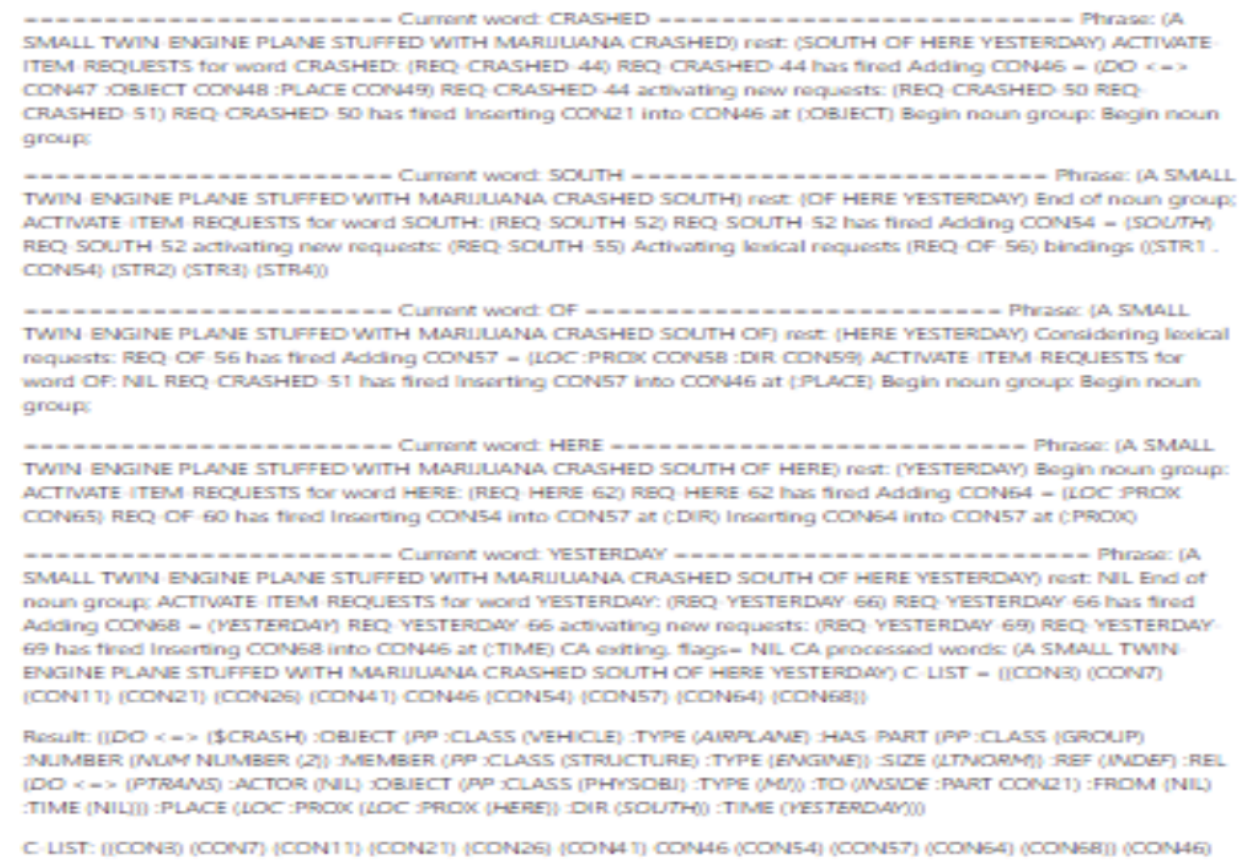
---------- Current word: TWIN ENGINE ---------- Phrase: (A SMALL TWIN ENGINE) rest: (PLANE STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) Begin noun group: ACTIVATE ITEM REQUESTS for word TWIN ENGINE: (REQ TWIN ENGINE 9) REQ TWIN ENGINE 9 has fired Adding CON11 = (PP :CLASS CON12 :NUMBER CON13 :MEMBER CON15) REQ TWIN ENGINE 9 activating new requests: (REQ TWIN ENGINE 18) REQ SMALL 8 has fired Inserting CON7 into CON11 at (:SIZE)

---------- Current word: PLANE ---------- Phrase: (A SMALL TWIN ENGINE PLANE) rest: (STUFFED WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) End of noun group; ACTIVATE ITEM REQUESTS for word PLANE: (REQ PLANE 19) REQ PLANE 19 has fired Adding CON21 = (PP :CLASS CON22 :TYPE CON23) REQ TWIN ENGINE 18 has fired Inserting CON11 into CON21 at (:HAS-PART) REQ A 4 has fired A = CON3 found pp CON21 = CON3 Inserting CON3 into CON21 at (:REF)

---------- Current word: STUFFED ---------- Phrase: (A SMALL TWIN ENGINE PLANE STUFFED) rest: (WITH MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) ACTIVATE ITEM REQUESTS for word STUFFED: (REQ STUFFED 24) REQ STUFFED 24 has fired Adding CON26 = (DO <=> CON27 :ACTOR CON28 :OBJECT CON29 :TO CON30 :FROM CON32 :TIME CON33) Activating lexical requests (REQ WITH 34 REQ WITH 35) bindings ((STR1 . CON26) (STR2) (STR3))

---------- Current word: WITH ---------- Phrase: (A SMALL TWIN ENGINE PLANE STUFFED WITH) rest: (MARIJUANA CRASHED SOUTH OF HERE YESTERDAY) Considering lexical requests: REQ WITH 34 has fired REQ WITH 35 has fired ACTIVATE ITEM REQUESTS for word WITH: NIL REQ WITH 36 has fired Inserting CON26 into CON21 at (:REL) Begin noun group: Begin noun group:

---------- Current word: MARIJUANA ---------- Phrase: (A SMALL TWIN ENGINE PLANE STUFFED WITH MARIJUANA) rest: (CRASHED SOUTH OF HERE YESTERDAY) End of noun group; ACTIVATE ITEM REQUESTS for word MARIJUANA: (REQ MARIJUANA 39) REQ MARIJUANA 39 has fired Adding CON41 = (PP :CLASS CON42 :TYPE CON43) REQ WITH 37 has fired Inserting CON41 into CON26 at (:OBJECT)



The lisp output breaks down each additional word and contains debugging statements to let us know that the analyzer is properly breaking down the information. Each word will run through one or more requests, create a con of that request, and potentially insert previous cons into the new con. The final output will be the latest con and will be the full deconstruction of the sentence.

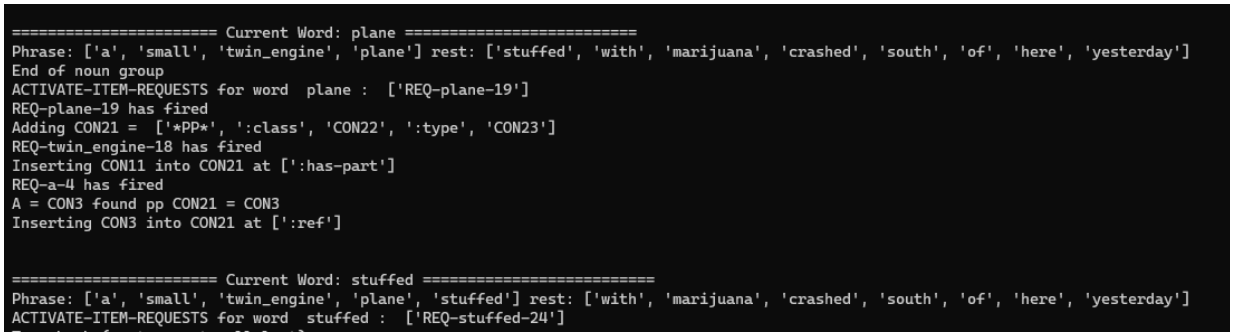
And this is the current Python output:

```
C:\Users\salsa\Documents\GitHub\Masters-Project>python main.py
Current Input: ['(a small twin-engine plane stuffed with marijuana crashed south of here yesterday)']
New Sentence is ['a', 'small', 'twin-engine', 'plane', 'stuffed', 'with', 'marijuana', 'crashed', 'south', 'of', 'here', 'yesterday']


====================== Current Word: a ========================
Phrase: ['a'] rest: ['small', 'twin-engine', 'plane', 'stuffed', 'with', 'marijuana', 'crashed', 'south', 'of', 'here', 'yesterday']
ACTIVATE-ITEM-REQUESTS for word  a :  ['REQ-a-1']
REQ-a-1 has fired
Adding CON3 =  ['*indef*']
REQ-a-1 activating new requests:  ['REQ-a-4']
Begin noun group:
Begin noun group;


====================== Current Word: small ========================
Phrase: ['a', 'small'] rest: ['twin-engine', 'plane', 'stuffed', 'with', 'marijuana', 'crashed', 'south', 'of', 'here', 'yesterday']
Begin noun group:
ACTIVATE-ITEM-REQUESTS for word  small :  ['REQ-small-5']
REQ-small-5 has fired
Adding CON7 =  ['*ltnorm*']
REQ-small-5 activating new requests:  ['REQ-small-8']


====================== Current Word: twin_engine ========================
Phrase: ['a', 'small', 'twin_engine'] rest: ['plane', 'stuffed', 'with', 'marijuana', 'crashed', 'south', 'of', 'here', 'yesterday']
Begin noun group:
ACTIVATE-ITEM-REQUESTS for word  twin_engine :  ['REQ-twin_engine-9']
REQ-twin_engine-9 has fired
Adding CON11 =  ['*PP*', ':class', 'CON12', ':number', 'CON13', ':member', 'CON15']
REQ-twin_engine-9 activating new requests:  ['REQ-twin_engine-18']
REQ-small-8 has fired
Inserting CON7 into CON11 at [':size']
```



We have been able to properly process four words with the same output as the lisp code. The next team that takes over will need to figure out how to complete lexical requests in order to make progress on the word stuffed.

VII. Discussion

The Lisp programming language is one of the first high level programming languages. Understanding its intricacies was quite difficult and involved a large amount of debugging using print statements on the original code. A parentheses based language like Lisp is far more complicated because you need to consistently count to ensure you're on the proper parentheses for the command. Furthermore, conceptual analyzers in general are complicated systems that require a deep understanding of conceptual dependencies.

Building the Python version of Larry Birnbaum's conceptual analyzer is important because it was one of the first conceptual analyzers and can be used as a valuable teaching tool to explain how conceptual analyzers work. In the original version of lisp, it is hard to understand the intricacies of the language, but Python is one of the simplest programming languages today. This will make the understanding of the analyzer a lot simpler. Furthermore, many conceptual analyzers today are designed in Python due to its wide adoption among the programming community so continuing the migration of older conceptual analyzers to Python can only aid in their progress.

VIII. Conclusion

We provide a detailed overview of Larry Birnbaum's conceptual analyzer, give a high level walkthrough of our translated codebase, explain key functions/concepts that motivated our design choices, mention the numerous challenges encountered during the translation process, and show how our Python implementations compares to the original lisp implementation. We also provide a step-by-step walkthrough of how a phrase is parsed by the conceptual analyzer and the resulting output.

We successfully parsed the first 4 words from the phrase "A Small Twin-Engine Plane stuffed with marijuana crashed south of here yesterday". Our output matches with that of the Lisp implementation. Our work demonstrates that a fully functional conceptual analyzer in Python is feasible. Many modern explainable AI applications that deal with conceptual

primitives are implemented in Python, and will significantly benefit from a conceptual analyzer implemented in Python.

IX. Future Work

- The next team will have to figure out how to implement lexical requests in order to complete the word stuffed.
- They will also need to figure out how to process the other 8 words in the sentence.
- There are multiple functions that are not properly fleshed out within concept_fns.py with regards to substituting values on property tags verus just appending values.
- The predicates.py function has implemented functions, but they have not really been tested extensively.
- The feature function also needs a lot of improvement and currently only grabs values at the head of the con.
    - There need to be implementations for recursive calls of get_role_filler to replace cons that are within cons with their values.

X. References

[1] Schank, Roger C. "The Primitive ACTs of Conceptual Dependency." Theoretical Issues in Natural Language Processing, 1975, https://aclanthology.org/T75-2008.

[2] Zhou, Mackie, et al. "Novel Primitive Decompositions for Real-World Physical Reasoning." Proceedings of the Third International Workshop on Self-Supervised Learning, edited by Kristinn R. Thórisson, vol. 192, PMLR, 2022, pp. 22-34.

[3] Macbeth, Jamie C. and Gromann, Dagmar, "Towards Modeling Conceptual Dependency Primitives with Image Schema Logic" (2019). Computer Science: Faculty Publications, Smith College, Northampton, MA. https://scholarworks.smith.edu/csc_facpubs/160

[4] Gilpin, Leilani H., Jamie C. Macbeth, and Evelyn Florentine. "Monitoring Scene Understanders with Conceptual Primitive Decomposition and Commonsense Knowledge." In Proceedings of the conference, 2018, pp. 45-63.

[5] Norvig, Peter. "Lispy - Lisp in Python." Peter Norvig, 2023, https://norvig.com/lispy.html.

[6] "Common-Lisp.net." Common-Lisp.net, n.d., https://common-lisp.net/.

[7] Steele, Guy L. Common Lisp: The Language. 2nd ed., Digital Press, 1990.

[8] https://github.com/norvig/pytudes/blob/main/py/lis.py

[9] "Creating an instance of a class with a variable in Python." Stack Overflow, stackoverflow.com/questions/2136760/creating-an-instance-of-a-class-with-a-variable-in-Python. Accessed 7 June 2023.

[10] "Removing duplicates from a list of lists." Stack Overflow, stackoverflow.com/questions/2213923/removing-duplicates-from-a-list-of-lists. Accessed 7 June 2023.
[11] "How to find all occurrences of a substring." Stack Overflow, stackoverflow.com/questions/4664850/how-to-find-all-occurrences-of-a-substring. Accessed 7 June 2023.