CS 5600/6600: F21: Intelligent Systems
Assignment 08
Connecting Conceptual Analysis and Script-Based Understanding
Part 01: Knowledge Engineering for CA

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 23, 2021

Learning Objectives

1. Conceptual Dependency Theory
2. Conceptual Analysis
3. Natural Language Processing
4. Script-Based Understanding
5. Knowledge Engineering

Introduction

In the next two assignments, we'll do some natural language (NL) knowledge engineering to connect the CA and SAM systems. We'll split this task into 2 parts. In the first part (this assignment), we'll focus on knowledge engineering for the CA system. In the second part (the next assignment), we'll connect CA to the Script Applier Mechanism (SAM) system that models script-based understanding.

Recall that in the previous assignment, we read "A Natural Language Interface to a Robot Assembly System" by Selfridge and Vannoy. Mallory Selfridge is one of the founders of conceptual analysis (CA), along with Roger Schank and Larry Burnbaum. This article was one of the first articles in the NLP literature to pose the problem of knowledge engineering NL interfaces to robots. The interface proposed by Selfridge and Vannoy is based on CA and interacts with the robot's planning, vision, and learning modules (recall the concept of a teachable student system from Dr. Turing's article you read a week ago!). AI Planning is the next big theme we'll tackle in this class.

We also read "Mobile Robot Programming using Natural Language" by Lauria et al. This paper describes the design of a system that uses NL to teach a vision-based robot how to navigate in a miniature town and presents an instruction-based learning architecture for converting NL to robot-executable procedures.

Both papers are on the fundamental problem of interacting with robots in NL, which still remains the Holy Grail of symbolic AI. Why should we care about NL interfaces to robots? Because such interfaces 1) give access to robots to non-experts (i.e., non-programmers) and 2) enable all of us (programmers and non-programmers alike) to program robots by NL, which is how we "program" each other daily.

Problem 1 (5 points)

The zip archive of this assignment contains the Lisp source of the CA system.

Loading and Running CA

Open the file `ca-loader.lisp` in your editor and edit the value of the variable `*ca-dir*` accordingly. This directory should point to the directory where you unzipped the contents of the folder `ca_sam`. Here's how I load CA in CLISP on Linux (I skip intermediate loading messages to save space).

```
> (load "ca-loader.lisp")
T
> (ca-loader *files*)
T
```

The file `ca-defs.lisp` contains several words defined for CA. I defined in `ca-defs.lisp` a few concepts we discussed in class like `apple`, `a`, `an`, `apple`, `john`, `mary`, `ate`, `bought`, etc. Here're some examples.

```
(define-ca-word
    jack
    (concept nil (human :name (jack) :sex (male))))

(define-ca-word
    ate
  (concept ?act (ingest :time (past)))
  (request (test (before ?act ?actor (animate)))
           (actions (modify ?act :actor ?actor)))
  (request (test (after ?act ?food (food)))
   (actions (modify ?act :object ?food))))

(define-ca-word
    apple
    (concept nil (apple)))
```

Each concept is defined with the CA macro `define-ca-word`. This macro is not Lisp; it's part of the CA system that's built on top of Lisp.

As we run the CA examples below, pay attention to the tracer messages which tell you what the system is doing and when. Remember that after the system prints `----- Done -----`, it prints the contents of the C-LIST (concept list) and the R-LIST (request list) separated by a semicolon. For example, the output

```
; ----- Done -----
((APPLE)) ;
NIL
```

indicates that the C-LIST consists of one concept `(APPLE)` and the R-LIST is empty (i.e., NIL).

Several students ask me in class what is the `(<CA>)` on the C-LIST. It's a place holder on the C-LIST that allows one to run the before and after tests in concept definitions. It's placed on the C-LIST when the `(mark)` instruction is used in the definition. Here's a quick example. Suppose we define the `m` (we can be use any other symbol name) concept to place a marker on the C-LIST and bind its place to the variable `?z`. Let's assume that we added this definition to `ca-defs.lisp`

```
(define-ca-word
  m
  (mark ?z))
```

Let's load `ca-defs.lisp` into Lisp and run the CA on it. The `ca` is the top-level function that takes a backquoted list of symbols and applies CA to it.

```
> (ca '(m))
; ----- Reading M -----
; Action (M 0): (MARK ?Z)
; ----- Done -----
((<CA>)) ;
NIL
```

The C-LIST contains the marker `(<CA>)` and the R-LIST is empty. Here's another example.

```
> (load "ca-defs.lisp")
T
> (ca '(apple))
; ----- Reading APPLE -----
; Action (APPLE 0): (CONCEPT NIL (APPLE))
; ----- Done -----
((APPLE)) ;
NIL
```

The system read APPLE, found the concept associate with that word, placed it on the C-LIST, and stopped. The C-LIST contains one concept `(APPLE)` and the R-LIST is empty (i.e., NIL). The `ca-defs.lisp` contains the following definition of `an`.

```
(define-ca-word
    an
    (mark ?x)
    (request (test (after ?x ?con (concept)))
             (actions (modify ?con :ref (indef))))
    (request (test (after ?x ?con (concept)))
             (actions (modify ?con :number (singular)))))
```

This definition states that we'll place a marker to the C-LIST and then add two requests to the R-LIST. The first request will look for a concept after the marker and if such a concept is found, its `:ref` (i.e., reference) slot will be modified with the concept `(indef)` (i.e., indefinite) and its `:number` slot will be modified with the concept `(singular)`. Let's run the CA on it. The strings "fails" and "succeeds" indicate whether a specific request fails or succeeds, respectively. The character `#` in the output below is how CLISP abbreviates symbolic expressions that may be too long for output. Other versions of Common Lisp may print differently.

```
> (ca '(an))
; ----- Reading AN -----
; Action (AN 0): (MARK ?X)
; Action (AN 1): (REQUEST (TEST #) (ACTIONS #))
; Action (AN 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
```

```
; ----- Done -----
((<CA>)) ;
((REQUEST 2 AN) (REQUEST 1 AN))
```

What does the above output show us? That the C-LIST consists of the marker `(<CA>)` and the R-LIST consists of two requests associated with `an`, because they have failed. Let's add the word `apple` after `an` and see what happens.

```
> (ca '(an apple))
; ----- Reading AN -----
; Action (AN 0): (MARK ?X)
; Action (AN 1): (REQUEST (TEST #) (ACTIONS #))
; Action (AN 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; ----- Reading APPLE -----
; Action (APPLE 0): (CONCEPT NIL (APPLE))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 2 REQUEST 0): (MODIFY ?CON :NUMBER (SINGULAR))
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 1 REQUEST 0): (MODIFY ?CON :REF (INDEF))
; ----- Done -----
((APPLE :REF (INDEF) :NUMBER (SINGULAR)) (<CA>)) ;
NIL
```

As you can see, this time, the C-LIST contains the concept `APPLE` with the slots `:REF` and `:NUMBER` filled in and no requests, because the requests associated with `an` have succeeded.

Let's run an incomplete sentence `(mary ate)`. Bear in mind that `ate` is defined as follows in `ca-defs.lisp`.

```
(define-ca-word
  ate
  (concept ?act (ingest :time (past)))
  (request (test (before ?act ?actor (animate)))
           (actions (modify ?act :actor ?actor)))
  (request (test (after ?act ?food (food)))
           (actions (modify ?act :object ?food))))
```

```
> (ca '(mary ate))
; ----- Reading MARY -----
; Action (MARY 0): (CONCEPT NIL (HUMAN :NAME # :SEX #))
; ----- Reading ATE -----
; Action (ATE 0): (CONCEPT ?ACT (INGEST :TIME #))
; Action (ATE 1): (REQUEST (TEST #) (ACTIONS #))
; Action (ATE 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; Test   (ATE 1 REQUEST TEST): (BEFORE ?ACT ?ACTOR (ANIMATE)) succeeds
; Action (ATE 1 REQUEST 0): (MODIFY ?ACT :ACTOR ?ACTOR)
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; ----- Done -----
((INGEST :ACTOR (HUMAN :NAME (MARY) :SEX (FEMALE)) :TIME (PAST))
 (HUMAN :NAME (MARY) :SEX (FEMALE))) ;
((REQUEST 2 ATE))
```

The C-LIST contains two concepts:

1. (INGEST :ACTOR (HUMAN :NAME (MARY) :SEX (FEMALE)) :TIME (PAST));
2. (HUMAN :NAME (MARY) :SEX (FEMALE)).

The concept (HUMAN :NAME (MARY) :SEX (FEMALE)) is the filler of the :ACTOR slot of the INGEST concept. But, INGEST does not have its :OBJECT slot filled. Hence, the second request associated with ate is still on the R-LIST waiting for a concept that can fill in :OBJECT of INGEST.

Let's do another incomplete sentence.

```
> (ca '(ate an apple))
; ----- Reading ATE -----
; Action (ATE 0): (CONCEPT ?ACT (INGEST :TIME #))
; Action (ATE 1): (REQUEST (TEST #) (ACTIONS #))
; Action (ATE 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; Test   (ATE 1 REQUEST TEST): (BEFORE ?ACT ?ACTOR (ANIMATE)) fails
; ----- Reading AN -----
; Action (AN 0): (MARK ?X)
; Action (AN 1): (REQUEST (TEST #) (ACTIONS #))
; Action (AN 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; Test   (ATE 1 REQUEST TEST): (BEFORE ?ACT ?ACTOR (ANIMATE)) fails
; ----- Reading APPLE -----
; Action (APPLE 0): (CONCEPT NIL (APPLE))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 2 REQUEST 0): (MODIFY ?CON :NUMBER (SINGULAR))
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 1 REQUEST 0): (MODIFY ?CON :REF (INDEF))
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) succeeds
; Action (ATE 2 REQUEST 0): (MODIFY ?ACT :OBJECT ?FOOD)
; Test   (ATE 1 REQUEST TEST): (BEFORE ?ACT ?ACTOR (ANIMATE)) fails
; ----- Done -----
((APPLE :REF (INDEF) :NUMBER (SINGULAR)) (<CA>)
 (INGEST :OBJECT (APPLE :REF (INDEF) :NUMBER (SINGULAR)) :TIME (PAST))) ;
((REQUEST 1 ATE))
```

The C-LIST contains

1. (APPLE :REF (INDEF) :NUMBER (SINGULAR));
2. (<CA>);
3. (INGEST :OBJECT (APPLE :REF (INDEF) :NUMBER (SINGULAR)) :TIME (PAST)).

The concept INGEST has its :OBJECT slot filled in, but the :ACTOR slot is empty. Hence, the first request associated with ate is still on the R-LIST, because, as the output above shows, it failed.

Let's run a complete sentence.

```
> (ca '(jack ate an apple))
; ----- Reading JACK -----
; Action (JACK 0): (CONCEPT NIL (HUMAN :NAME # :SEX #))
; ----- Reading ATE -----
; Action (ATE 0): (CONCEPT ?ACT (INGEST :TIME #))
; Action (ATE 1): (REQUEST (TEST #) (ACTIONS #))
; Action (ATE 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; Test   (ATE 1 REQUEST TEST): (BEFORE ?ACT ?ACTOR (ANIMATE)) succeeds
; Action (ATE 1 REQUEST 0): (MODIFY ?ACT :ACTOR ?ACTOR)
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; ----- Reading AN -----
; Action (AN 0): (MARK ?X)
; Action (AN 1): (REQUEST (TEST #) (ACTIONS #))
; Action (AN 2): (REQUEST (TEST #) (ACTIONS #))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) fails
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) fails
; ----- Reading APPLE -----
; Action (APPLE 0): (CONCEPT NIL (APPLE))
; Test   (AN 2 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 2 REQUEST 0): (MODIFY ?CON :NUMBER (SINGULAR))
; Test   (AN 1 REQUEST TEST): (AFTER ?X ?CON (CONCEPT)) succeeds
; Action (AN 1 REQUEST 0): (MODIFY ?CON :REF (INDEF))
; Test   (ATE 2 REQUEST TEST): (AFTER ?ACT ?FOOD (FOOD)) succeeds
; Action (ATE 2 REQUEST 0): (MODIFY ?ACT :OBJECT ?FOOD)
; ----- Done -----
((APPLE :REF (INDEF) :NUMBER (SINGULAR)) (<CA>)
 (INGEST :OBJECT (APPLE :REF (INDEF) :NUMBER (SINGULAR)) :ACTOR
  (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))
 (HUMAN :NAME (JACK) :SEX (MALE))) ;
NIL
```

The R-LIST is empty and the `INGEST` concept has all of its slots filled in. Nirvana!

Doing the CA of a Story

Let's do the CA of a version of the restaurant story we analyzed in class. Here's the story.

```
Jack went to a restaurant. He ate a lobster. He went home.
```

We'll simplify it by replacing the pronoun "he" with its reference "Jack" in each sentence so that we don't have to deal with pronoun reference resolution. I note in passing that this is a fascinating NLP problem, but marginal to what we're doing in this assignment. After the pronouns are removed, the story reads as follows.

```
Jack went to a restaurant. Jack ate a lobster. Jack went home.
```

We need to convert the story into CD's (aka conceptualizations) that formalize its meaning so that we can later (i.e., in the next assignment) feed them to the SAM system to apply scripts. Recall

that SAM models script-based understanding. In other words, SAM will make sense of this story by matching it with a script and filling in the missing segments of causal chains.

Let's put three sentences of our story into a variable. This variable `*restaurant-story*` is a list of 3 lists.

```
> (setf *restaurant-story*
     '((jack went to a restaurant)
       (jack ate a lobster)
       (jack went home)))
> *restaurant-story*
((JACK WENT TO A RESTAURANT) (JACK ATE A LOBSTER) (JACK WENT HOME))
```

We can use the Lisp `elt` function to get individual sentences. This function retrieves the i-th element of a sequence (e.g., a list, array, string) given to it as the first argument.

```
> (elt *restaurant-story* 0)
(JACK WENT TO A RESTAURANT)
> (elt *restaurant-story* 1)
(JACK ATE A LOBSTER)
> (elt *restaurant-story* 2)
(JACK WENT HOME)
```

We can also use the Lisp macro `dolist` to iterate through the sentences of the story and print each of them out.

```
> (dolist (x *restaurant-story*)
        (print x))

(JACK WENT TO A RESTAURANT)
(JACK ATE A LOBSTER)
(JACK WENT HOME)
NIL
```

To process this story, we need to knowledge engineer a few more definitions for CA in `ca-defs.lisp`. Toward that end, define the concepts of `went`, `restaurant`, `home`, `lobster`, and `to`.

```
(define-ca-word
    went
    ;;; your definition here
    )

(define-ca-word
    restaurant
    ;;; your definition here
    )

(define-ca-word
    home
    ;;; your definition here
    )

(define-ca-word
    lobster
    ;;; your definition here
```

```
    )

(define-ca-word
    to
    ;;; your definition here
    )
```

Once we have defined these concepts, we can load these definitions into Lisp by loading `ca-defs.lisp` into Lisp (don't re-start Lisp, just load `ca-defs.lisp` into it after saving this file with your definitions) and run CA on each sentence of the restaurant story to make sure that CA can handle it.

Let's do the first sentence.

```
> (load "ca-defs.lisp")
> (ca (elt *restaurant-story* 0))
; ----- Done -----
((RESTAURANT :REF (INDEF) :NUMBER (SINGULAR)) (<CA>) (TO)
 (PTRANS :TO (RESTAURANT :REF (INDEF) :NUMBER (SINGULAR)) :ACTOR
  (HUMAN :NAME (JACK) :SEX (MALE)) :OBJECT (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))
 (HUMAN :NAME (JACK) :SEX (MALE))) ;
((REQUEST 4 A) (REQUEST 3 A) (REQUEST 3 WENT))
```

The above output shows that the C-LIST contains the concept

```
(PTRANS
  :TO (RESTAURANT :REF (INDEF) :NUMBER (SINGULAR))
  :ACTOR (HUMAN :NAME (JACK) :SEX (MALE))
  :OBJECT (HUMAN :NAME (JACK) :SEX (MALE))
  :TIME (PAST))
```

which is what we want. On to the second sentence.

```
> (ca (elt *restaurant-story* 1))
; ----- Done -----
((LOBSTER :REF (INDEF) :NUMBER (SINGULAR)) (<CA>)
 (INGEST :OBJECT (LOBSTER :REF (INDEF) :NUMBER (SINGULAR)) :ACTOR
  (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))
 (HUMAN :NAME (JACK) :SEX (MALE))) ;
((REQUEST 2 A) (REQUEST 1 A))
```

The above output shows that the C-LIST contains the concept

```
(INGEST
  :OBJECT (LOBSTER :REF (INDEF) :NUMBER (SINGULAR))
  :ACTOR (HUMAN :NAME (JACK) :SEX (MALE))
  :TIME (PAST))
```

which is what we want. Let's do the third sentence.

```
> (ca (elt *restaurant-story* 2))
((HOME)
 (PTRANS :TO (HOME) :ACTOR (HUMAN :NAME (JACK) :SEX (MALE)) :OBJECT
  (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))
 (HUMAN :NAME (JACK) :SEX (MALE))) ;
((REQUEST 3 WENT))
```

The above output shows that the C-LIST contains the concept

```
(PTRANS
  :TO (HOME)
  :ACTOR (HUMAN :NAME (JACK) :SEX (MALE))
  :OBJECT (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))
```

as expected.

What to Submit

1. Save your word definitions in `ca-defs.lisp` submit it in Canvas.

Happy Knowledge Engineering!