CS 5600/6600: F20: Intelligent Systems
Assignment 10
Natural Language Knowledge Engineering for CA and SAM

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 7, 2020

Learning Objectives

1. Conceptual Dependency
2. Script-Based Understanding
3. Script Applier Mechanism
4. Knowledge Engineering

Introduction

In this assignment, we'll do some natural language (NL) knowledge engineering to connect the CA and SAM systems.

Problem 1 (2 points)

Write a review of "A Natural Language Interface to a Robot Assembly System" by Selfridge and Vannoy. Dr. Selfridge is one of the founders of conceptual analysis. This article was one of the first articles in the NLP literature to pose the problem of knowledge engineering NL interfaces to robots, which is a fundamental problem that is still with us.

The interface proposed by Selfridge and Vannoy is based on the use of conceptual analysis to perform NLP. The interface also interacts with the robot's planning, vision, and learning (recall the concept of a teachable student system from Dr. Turing's article you read a week ago!) systems. AI Planning is the next big theme we'll tackle in this class.

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 natural language, which is how we "program" each other daily.

Problem 2 (3 points)

The zip archive of this assignment contains the Lisp source (in the `ca_sam` folder) of two systems: CA and SAM. Let's load and run both.

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 skipped the 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. For example,

```
(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)))
```

Let's load these definitions and run CA on a few inputs. As you run CA, 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.

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

Loading and Running SAM

The SAM system is comprised of three files `sam.lisp`, `for.lisp`, and `cd-functions.lisp`. Here's what I do in CLISP on Linux.

```
> (load "sam.lisp")
T
```

Let's consider the restaurant story we analyzed in class.

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

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

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

The variable `*restaurant-story-cds*` contains the CD representations (aka conceptualizations) that correspond to this story. Of course, this should be the output of the CA system, but we'll work on it later after we get SAM to process these CDs.

```
(defparameter *restaurant-story-cds*
  '((ptrans
     (:actor (human (:name (jack)) (:sex (male))))
     (:object (human (:name (jack)) (:sex (male))))
     (:to (restaurant))
     (:time (past)))
    (ingest
     (:actor (human (:name (jack)) (:sex (male))))
     (:object (lobster))
     (:time (past)))
    (ptrans
     (:actor (human (:name (jack)) (:sex (male))))
     (:object (human (:name (jack)) (:sex (male))))
     (:from (restaurant))
     (:time (past)))))
```

Recall that SAM, as a system, models script-based understanding. In other words, the system must make sense of this story by matching it with a script and filling in the missing links of causal chains. The script we'll engineer for SAM to deal with this story is defined in `sam.lisp` and shown below. Recall that all script names, by convention, start with the character `$` (e.g., `$bus`, `$restaurant`, etc.)

The `$restaurant` script has nine primitive act CDs while there are only three CDs in the restaurant story above. SAM's objective is to fill in all the missing details. That's what scripts are used for – making sense of the context by filling in the blanks that are not explicitly mentioned.

```
(setf (events-script '$restaurant)
      '((ptrans (:actor ?client)
                (:object ?client)
                (:to ?restaurant)
                (:time ?time))
        (ptrans (:actor ?client)
                (:object ?client)
                (:to (table))
                (:time ?time))
        (mtrans (:actor ?client)
                (:object (menu))
                (:to ?client)
                (:time ?time))
        (mbuild (:actor ?client)
                (:object (ingest (:actor ?client)
                                 (:object ?meal)))
                (:time ?time))
        (mtrans (:actor ?client)
                (:object (ingest (:actor ?client)
                                 (:object ?meal))
                (:to (server))
                (:time ?time))
        (ptrans (:actor (server))
                (:object ?meal)
                (:to ?client)
                (:time ?time))
        (ingest (:actor ?client)
                (:object ?meal)
                (:time ?time))
        (atrans (:actor ?client)
                (:object (money))
                (:from ?client)
                (:to ?restaurant)
                (:time ?time))
        (ptrans (:actor ?client)
                (:object ?client)
                (:from ?restaurant)
                (:to ?elsewhere)
                (:time ?time))))

(setf (associated-script 'restaurant) '$restaurant)
```

Let's see what SAM will do with the restaurant story CDs now that it has access to the restaurant script. As the output below shows, SAM fills in the details of the restaurant script by filling in the role fillers with the appropriate bindings.

```
> (sam *restaurant-story-cds*)
((PTRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:TO (RESTAURANT)) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:TO (TABLE)) (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:OBJECT (MENU))
  (:TO (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:TIME (PAST)))
 (MBUILD (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (INGEST (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:OBJECT (LOBSTER))))
  (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (INGEST (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:OBJECT (LOBSTER))))
  (:TO (SERVER)) (:TIME (PAST)))
 (PTRANS (:ACTOR (SERVER)) (:OBJECT (LOBSTER)) (:TO (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:TIME (PAST)))
 (INGEST (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:OBJECT (LOBSTER))
  (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:OBJECT (MONEY))
  (:FROM (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:TO (RESTAURANT)) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (:FROM (RESTAURANT)) (:TO NIL)
  (:TIME (PAST)))
 ($RESTAURANT (CLIENT (HUMAN (:NAME (JACK)) (:SEX (MALE)))) (RESTAURANT (RESTAURANT))
  (TIME (PAST)) (MEAL (LOBSTER))))
```

Connecting CA with SAM

The next step is to connect CA with SAM, because the input to a complete NL system should be natural language sentences, not CDs. Below we represent the jack restaurant story as a list of three lists and save these sentences in the variable `*restaurant-story*`. Here's how.

```
> (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))
```

To process this story, we need to knowledge engineer a few more definitions for CA in `ca-defs.lisp`. Let's do it.

```
(define-ca-word
    went
    (concept ?act (ptrans :time (past)))
    (request (test (before ?act ?actor (animate)))
             (actions (modify ?act :actor ?actor)))
    (request (test (after ?act ?dir (direction)))
             (actions (modify ?act :to ?dir)))
    (request (test (after ?act ?loc (location)))
     (actions (modify ?act :to ?loc))))

(define-ca-word
    restaurant
    (concept nil (restaurant)))

(define-ca-word
    home
    (concept nil (home)))

(define-ca-word
    to
    (concept ?to (to))
    (request (test (and (after ?to ?loc (location))
(before ?dir ?ptrans (ptrans))))
             (actions (modify ?ptrans :to ?loc))))

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

Now we can load these definitions into Lisp and run CA on each sentence of the restaurant story to make sure that CA can handle it. The Lisp `elt` function below retrieves the i-th element of a sequence (e.g., a list, array, string) given to it as the first argument.

```
> (ca (elt *restaurant-story* 0))

((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))

> (ca (elt *restaurant-story* 1))

((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))

> (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))
```

So far so good! But you may have noticed that the CD notation of SAM's input and CA's output are slightly different. For example, when processing `(jack ate a lobster)` CA's output is as follows.

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

However, SAM expects something like this.

```
(INGEST
  (:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))
  (:OBJECT (LOBSTER))
  (:TIME (PAST)))
```

The order of the roles does not matter. What matters is slightly different representation of role-filler pairs. In particular, in SAM, unlike in CA, the role-filler pairs are represented as lists. For example, in SAM the actor roler-filler pair is `(:ACTOR (HUMAN (:NAME (JACK)) (:SEX (MALE))))` whereas in CA the same role-filler pair looks like `:ACTOR (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))`.

What gives? The theory of conceptual analysis is presented in two seminal books: 1) "Inside Computer Understanding" (ICU) by R. Schank and C. Riesbeck and 2) "Scripts, Plans, Goals, and Understanding" (SPGU) by R. Schank and R. Abelson. The version of CA for this assignment is written in line with the ICU specs while the version of SAM is written along the lines of SPGU. This is not a major issue but something to be aware of.

The function `ca-cd-to-sam-cd` in `ca.lisp` is used to convert CDs from one format to another. The function `sents-to-cds` in `ca.lisp` is used to do the conceptual analysis of a list of sentences and extract from them only primitive act CDs. Here's a sample call of this function on `*restaurant-story*`.

```
> (sents-to-cds *restaurant-story*)
((PTRANS (:TIME (PAST)) (:OBJECT (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))))
 (INGEST (:TIME (PAST))
  (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK)) (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF)))))
 (PTRANS (:TIME (PAST)) (:OBJECT (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:TO (HOME (:NUMBER (SINGULAR)) (:REF (INDEF))))))
```

Now we're in the position to unleash SAM on the restaurant story. Let's do it!

```
> (sam (sents-to-cds *restaurant-story*))
((PTRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT (HUMAN (:SEX (MALE)) (:NAME (JACK)))) (:TO (TABLE)) (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK)))) (:OBJECT (MENU))
  (:TO (HUMAN (:SEX (MALE)) (:NAME (JACK)))) (:TIME (PAST)))
 (MBUILD (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT
   (INGEST (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
    (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))))
  (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT
   (INGEST (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
    (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))))
  (:TO (SERVER)) (:TIME (PAST)))
 (PTRANS (:ACTOR (SERVER)) (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:TO (HUMAN (:SEX (MALE)) (:NAME (JACK)))) (:TIME (PAST)))
 (INGEST (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK)))) (:OBJECT (MONEY))
  (:FROM (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:OBJECT (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (:FROM (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:TO (HOME (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 ($RESTAURANT (CLIENT (HUMAN (:SEX (MALE)) (:NAME (JACK))))
  (RESTAURANT (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))) (TIME (PAST))
  (MEAL (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (ELSEWHERE (HOME (:NUMBER (SINGULAR)) (:REF (INDEF))))))
```

Note that SAM managed to fill all the blanks in the restaurant script.

Undestanding a Shopping Story

Consider the following shopping story.

```
Ann went to a store. Ann bought a kite. Ann went home.
```

Let's save the story's sentences in a variable `*shopping-story*`.

```
(setf *shopping-story*
  '((ann went to a store)
    (ann bought a kite)
    (ann went home)))
```

Add new definitions to `ca-defs.lisp` for CA to convert these sentences into CDs and knowledge engineer a shopping script in `sam.lisp` for SAM to use in interpreting this story. Below is SAM's output for my shopping script. Yours should be similar but may be different depending on the primitive acts you'll use in it.

```
> (sam (sents-to-cds *shopping-story*))
((PTRANS (:ACTOR (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (:OBJECT (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (:TO (STORE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (ATRANS (:TIME (PAST))
  (:ACTOR (HUMAN (:SEX (FEMALE)) (:NAME (ANN)) (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF)))))
 (PTRANS (:ACTOR (HUMAN (:SEX (FEMALE)) (:NAME (ANN)))) (:OBJECT NIL)
  (:TO (HUMAN (:SEX (FEMALE)) (:NAME (ANN)))) (:TIME (PAST)))
 (ATRANS (:ACTOR (STORE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:OBJECT NIL)
  (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:TO (HUMAN (:SEX (FEMALE)) (:NAME (ANN)))) (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:SEX (FEMALE)) (:NAME (ANN)))) (:OBJECT (MONEY))
  (:FROM (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (:TO (STORE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (:OBJECT (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (:TO (HOME (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 ($SHOPPING (SHOPPER (HUMAN (:SEX (FEMALE)) (:NAME (ANN))))
  (STORE (STORE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (TIME (PAST))
  (ELSEWHERE (HOME (:NUMBER (SINGULAR)) (:REF (INDEF))))))
```

What to Submit

1. Save your paper analysis in `cs5600_6600_f20_hw10_paper.pdf`. Remember to PDF your texts!
2. Save your word definitions in `ca-defs.lisp` and your script in `sam.lisp` and submit these two files through Canvas.

Happy Writing, Thinking, and Knowledge Engineering!