CS 5600/6600/7890: Intelligent Systems
Assignment 9
Knowledge Engineering for CA and SAM

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 27, 2018

Learning Objectives

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

Introduction

In this assignment, we'll continue to work with knowledge engineering with Conceptual Dependency (CD) but make our focus programmatic rather than purely conceptual as in the previous assignment. We'll also play with the Script Applier Mechanism (SAM).

Problem 1

The zip archive of Assignment 9 contains the Lisp source of two systems: CA and SAM. Let's load both systems into the Lisp Virtual Machine.

Loading CA

Open the file `ca-loader.lisp` in your editor and edit the value of the variable `*ca-dir*`. This directory should point to the directory where you unzipped the contents of the folder `ca_sam`. If you're on Windows, your path should use backslashes. In other words, your definition should look something like this:

```
(defparameter *ca-dir* "C:\\Users\\vladimir\\programming\\lisp\\ca_sam\\")
```

If you're on Linux, cd into to the directory where you unzipped `ca_sam`, then start your Lisp VM and load CA.

```
$ clisp
> (load "ca-loader.lisp")
;; Loading file ca-loader.lisp ...
;; Loaded file ca-loader.lisp
```

```
T
> (ca-loader *files*)
;; Loading file /home/vladimir/programming/lisp/ca_sam/ca-utilities.lisp ...
;; Loaded file /home/vladimir/programming/lisp/ca_sam/ca-utilities.lisp
;; Loading file /home/vladimir/programming/lisp/ca_sam/cd.lisp ...
;; Loaded file /home/vladimir/programming/lisp/ca_sam/cd.lisp
;; Loading file /home/vladimir/programming/lisp/ca_sam/ca.lisp ...
;; Loaded file /home/vladimir/programming/lisp/ca_sam/ca.lisp
;; Loading file /home/vladimir/programming/lisp/ca_sam/ca-functions.lisp ...
;; Loaded file /home/vladimir/programming/lisp/ca_sam/ca-functions.lisp
;; Loading file /home/vladimir/programming/lisp/ca_sam/ca-lexicon.lisp ...
;; Loaded file /home/vladimir/programming/lisp/ca_sam/ca-lexicon.lisp
T
```

If you're on Windows and using Allegro Common Lisp (ACL) IDE, start it and go to the listener window. The listener window should say "Listener 1" in its top left corner. First, you need to change the package from `CG-USER` to `CL-USER` by typing into the prompt the following command.

```
CG-USER(1): :package :user
CL-USER(2):
```

Then load the loader file as follows.

```
CG-USER(1): :package :user
CL-USER(2): (load "C:\\Users\\vladimir\\programming\\lisp\\ca_sam\\ca-loader.lisp")
; Loading C:\Users\vladimir\programming\lisp\ca_sam\ca-utilities.lisp
; Loading C:\Users\vladimir\programming\lisp\ca_sam\cd.lisp
; Loading C:\Users\vladimir\programming\lisp\ca_sam\ca.lisp
; Loading C:\Users\vladimir\programming\lisp\ca_sam\ca-functions.lisp
; Loading C:\Users\vladimir\programming\lisp\ca_sam\ca-lexicon.lisp
T
CL-USER(3):
```

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 debugging tracer messages which tell you what the system is doing and when.

```
> (load "ca-defs.lisp")
;; Loading file ca-defs.lisp ...
;; Loaded file 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 SAM

Let's load and run SAM. The system is comprised of three files `sam.lisp`, `for.lisp`, and `cd-functions.lisp`. To load SAM, just do

```
> (load "sam.lisp")
;; Loading file sam.lisp ...
;;  Loading file /home/vladimir/programming/lisp/ca_sam/cd-functions.lisp ...
;;  Loaded file /home/vladimir/programming/lisp/ca_sam/cd-functions.lisp
;; Loaded file sam.lisp
T
```

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

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

We'll simplify this story by replaceing the pronoun "he" with its reference "Jack" in each sentence so that we don't have to deal with pronoun resolution, a fascinating problem but tangential to the objectives of this assignment.

```
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)))))
```

SAM must make sense of this story by matching it with a script and filling in the missing links of causal chains. The script that we'll engineer for SAM to deal with this story is defined in `sam.lisp` as follows. Note that this script has nine primitive act CDs while there are only three CDs in the restaurant story above. SAM's job is to fill in the missing details.

```
(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. Note how 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

We have to connect CA with SAM, because the input to the system should be natural language sentences, not CDs. We'll represent the above story as a list of three lists. Let's save these sentences in the variable `restaurant-story`.

```
> (setf restaurant-story
     '((jack went to a restaurant)
       (jack ate a lobster)
       (jack went home)))
```

To process this story, we need to engineer a few more definitions for CA in `ca-defs.lisp`.

```
(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)))
```

Let's load these definitions into the Lisp VM and run CA on each sentence of the restaurant story to make sure that CA can handle it.

```
> (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

```
(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 it is `:ACTOR (HUMAN :NAME (JACK) :SEX (MALE)) :TIME (PAST))`. What gives? The theory of conceptual analysis is presented in two 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 more along the lines of SPGU. This is not a major issue but something that you should 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 is 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 can call SAM on natural language stories. Here is how.

```
> (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))))))
```

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` so that CA can convert these sentences into CDs and engineer a shopping script in `sam.lisp` for SAM to process it. 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

Save your word definitions in `ca-defs.lisp` and your script in `sam.lisp` and submit these two files through Canvas.

Happy Hacking and Knowledge Engineering!