CS 5600/6600: F24: Intelligent Systems
Assignment 09
onceptual Analysis and Script-Based Understanding

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 27, 2024

Learning Objectives

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

Introduction

In this assignment, we will work with two systems - the Conceptual Analyzer (CA) and the Script Applier Mechanism (SAM). CA implements some elements of Conceptual Dependency Theory (CDT) and SAM implements some elements of the theory of script-based understanding that Drs. Schank and Abelson presented in their famous book "Scripts, Plans, Goals, and Understanding: An Inquire into Human Knowledge Structures." We will connect them into one system to the script-based understanding of simple NL stories.

Conceptual Analysis Coding Lab

The zip contains, in the `ca_sam` folder with the Lisp source of CA and SAM. Let's load and run both.

Loading and Running CA

I won't spend much time on this part, because we covered it in the lectures and worked with CDT representations in the previous assignment.

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 into CLISP on my Linux. I skipped the intermediate loading messages below to save space.

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

The file `ca-defs.lisp` contains several word definitions for CA. Let's load these definitions and run CA on a few inputs.

```
> (load "ca-defs.lisp")
;; Loading file ca-defs.lisp ...
;; Loaded file ca-defs.lisp
#P"/home/vladimir/teaching/AI/F24/hw/hw09/sols/ca_sam/ca-defs.lisp
T
```

The file `ca-defs.lisp` contains several CA definitions of the concepts like `jack`, `ate`, `an`, and `apple`. Let's use them to process the sentence `jack ate an apple`. The `ca` is the top-level function that takes a quoted list of symbols and applies conceptual analysis to it. The `#` is what the CLISP writer uses to abbreviate the display of embedded lists. The strings "fails" and "succeeds" in the output below 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. E.g., instead of printing `CONCEPT NIL (HUMAN :NAME (JACK))`, it prints `CONCEPT NIL (HUMAN :NAME #))`.

```
> (ca '(jack ate an apple))
; ----- Reading JACK -----
; Action (JACK 0): (CONCEPT NIL (HUMAN :NAME #))
; ----- 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)) :TIME (PAST))
 (HUMAN :NAME (JACK))) ;
NIL
```

The first list after `----- Done -----` is the list of concepts, i.e., C-LIST. NIL after the semicolon, is the request list, i.e., R-LIST, which means that the list is empty. CA does not have any remaining

requests to run. We have the correct INGEST CD on the C-LIST which, if properly idented, is as follows. Remember it does not matter in which order slots, e.g., `:OBJECT`, `:ACTOR`, and `:TIME`, are given inside a CD. The C-LIST contains all the concepts discovered during the analysis of this sentence. The CD that represents the meaning of the read sentence is this.

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

But, what's `(<CA>)` on the C-LIST? In brief, it's just a place holder that allows us 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) 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.

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

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

```
> (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))
```

Loading and Running SAM

Let's load and run SAM. SAM is comprised of three files `sam.lisp`, `for.lisp`, and `cd-functions.lisp`. Here's how I do in CLISP. In the code below `#P/...` is how Lisp specifies paths on Linux.

```
> (load "sam.lisp")
;; Loaded file sam.lisp
```

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

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

Recall that we simplified this story by replacing the pronoun "he" with its reference "Jack" in each sentence so that we don't have to deal with pronoun resolution. After the pronouns are resolved to "Jack," the story reads as follows.

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

The variable `*restaurant-story-cds*` in `sam.lisp` contains the CD representations (aka conceptualizations) that correspond to this story. Of course, this should be the output of the CA system. We'll work on it later, as soon as we get SAM to process these CDs. If you open `sam.lisp`, you'll the following parameter definition.

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

After loading `sam.lisp`, we can see the value of the variable global variable `*restaurant-story-cds*` at the Lisp prompt as follows. I've indented the Lisp output for better legibility.

```
> *restaurant-story-cds*
((PTRANS (:ACTOR  (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:TO     (RESTAURANT))
         (:TIME   (PAST)))
 (INGEST (:ACTOR  (HUMAN (:NAME (JACK))))
         (:OBJECT (LOBSTER))
         (:TIME   (PAST)))
 (PTRANS (:ACTOR  (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:FROM   (RESTAURANT))
         (:TO     (HOME))
         (: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 (a cognitive structure modeled as a sequence of symbolic patterns in Lisp) and filling in the missing links of its causal chain. The script we'll engineer for SAM to deal with this story is defined in `sam.lisp` and shown below. All script names, by convention, start with the character `$` (e.g., `$bus`, `$restaurant`, `$train`, `$museum`, 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 events. That's what scripts are used for – making sense of the context by filling in the blanks that are not explicitly mentioned. This is known as *script-based inference*.

```
(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. I've indented the output for your convenience and omitted some output. The header of the script contains the variables found in the input CDs and

their matches. The database contains the CDs filled in by SAM from the story. I've indented the output for better legibility.

```
> (sam *restaurant-story-cds*)
Story done - final script header

($RESTAURANT (CLIENT (HUMAN (:NAME (JACK))))
             (RESTAURANT (RESTAURANT))
             (TIME (PAST))
             (MEAL (LOBSTER))
             (ELSEWHERE (HOME)))

Database contains:

((PTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:TO (RESTAURANT))
         (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:TO (TABLE))
         (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (MENU))
         (:TO (HUMAN (:NAME (JACK))))
         (:TIME (PAST)))
 (MBUILD (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (INGEST (:ACTOR (HUMAN (:NAME (JACK))))
                          (:OBJECT (LOBSTER))))
                          (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (INGEST (:ACTOR (HUMAN (:NAME (JACK))))
                          (:OBJECT (LOBSTER))))
                          (:TO (SERVER))
                          (:TIME (PAST)))
 (PTRANS (:ACTOR (SERVER))
         (:OBJECT (LOBSTER))
         (:TO (HUMAN (:NAME (JACK))))
         (:TIME (PAST)))
 (INGEST (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (LOBSTER))
         (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (MONEY))
         (:FROM (HUMAN (:NAME (JACK))))
         (:TO (RESTAURANT))
         (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:FROM (RESTAURANT))
         (:TO (HOME))
         (:TIME (PAST)))
 ($RESTAURANT (CLIENT (HUMAN (:NAME (JACK))))
              (RESTAURANT (RESTAURANT))
              (TIME (PAST))
              (MEAL (LOBSTER))
              (ELSEWHERE (HOME))))
```

Connecting CA with SAM

The next step is to connect CA with SAM, because the input to a complete NL system should be NL sentences, not CDs. Below we represent the restaurant story as a list of three lists, each of which is a sentence, and save these sentences in the variable `*restaurant-story*`. Here's how we can do it at the Lisp prompt.

```
> (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)
    (modify ?act :object ?actor)))
  (request (test (and (after ?act ?to (to))
      (after ?to  ?loc (location))))
   (actions (modify ?act :to ?loc))))

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

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

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

(define-ca-word
  to
  (concept ?to (to)))
```

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))
         :OBJECT (HUMAN :NAME (JACK))
         :ACTOR  (HUMAN :NAME (JACK))
         :TIME (PAST))
 (HUMAN :NAME (JACK))) ;
((REQUEST 4 A) (REQUEST 3 A))

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

((LOBSTER :REF (INDEF) :NUMBER (SINGULAR)) (<CA>)
 (INGEST :OBJECT (LOBSTER :REF (INDEF) :NUMBER (SINGULAR))
```

```
         :ACTOR (HUMAN :NAME (JACK))
         :TIME (PAST))
 (HUMAN :NAME (JACK))) ;
((REQUEST 2 A) (REQUEST 1 A))

> (ca (elt *restaurant-story* 2))

((HOME)
 (PTRANS :TO     (HOME)
         :OBJECT (HUMAN :NAME (JACK))
         :ACTOR  (HUMAN :NAME (JACK))
         :TIME   (PAST))
 (HUMAN :NAME (JACK))) ;
((REQUEST 3 WENT))
```

OK. 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))
  :TIME (PAST))
```

However, SAM expects the inputs where the roles and their fillers are parenthesized.

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

The order of the roles does not matter. What does matter is the 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))))` whereas in CA the same role-filler pair looks like a sequence of keyword and lists. `:ACTOR (HUMAN :NAME (JACK)) :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 though. Just something to be cognizant 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*`. This function runs CA on each sentence to obtain the CDs in CA' format and then converts them into SAM's format.

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

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

```
> (sam (sents-to-cds *restaurant-story*))
Story done - final script header
```

```
($RESTAURANT (CLIENT (HUMAN (:NAME (JACK))))
 (RESTAURANT (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
 (TIME (PAST)) (MEAL (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
 (ELSEWHERE (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))))

Database contains:

((PTRANS (:ACTOR (HUMAN (:NAME (JACK)))) (:OBJECT (HUMAN (:NAME (JACK))))
  (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))) (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK)))) (:OBJECT (HUMAN (:NAME (JACK))))
  (:TO (TABLE)) (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK)))) (:OBJECT (MENU))
  (:TO (HUMAN (:NAME (JACK)))) (:TIME (PAST)))
 (MBUILD (:ACTOR (HUMAN (:NAME (JACK))))
  (:OBJECT
   (INGEST (:ACTOR (HUMAN (:NAME (JACK))))
    (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))))
  (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (INGEST
                     (:ACTOR (HUMAN (:NAME (JACK))))
                     (:OBJECT (LOBSTER
                                 (:NUMBER (SINGULAR))
                                 (:REF (INDEF))))))
          (:TO (SERVER)) (:TIME (PAST)))
 (PTRANS (:ACTOR (SERVER))
         (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TO (HUMAN (:NAME (JACK))))
         (:TIME (PAST)))
 (INGEST (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (MONEY))
         (:FROM (HUMAN (:NAME (JACK))))
         (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (JACK))))
         (:OBJECT (HUMAN (:NAME (JACK))))
         (:FROM (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TO (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TIME (PAST)))
 ($RESTAURANT (CLIENT (HUMAN (:NAME (JACK))))
  (RESTAURANT (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (TIME (PAST)) (MEAL (LOBSTER (:NUMBER (SINGULAR)) (:REF (INDEF))))
  (ELSEWHERE (RESTAURANT (:NUMBER (SINGULAR)) (:REF (INDEF)))))
```

We did it! You may want to read through the list of filled events to understand the mechanics behind SAM's scrip filling. Note that SAM managed to fill in all the blanks in the restaurant script.

Problem 1 (5 points): 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*`. Notice that we've resolved the pronouns to "Ann."

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

Add a few definitions in `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. My script has 10 CDs. Your script may be different, depending on what knowledge engineering you'll do.

```
> (sam (sents-to-cds *shopping-story*))

Story done - final script header

($SHOPPING (SHOPPER (HUMAN (:NAME (ANN))))
           (STORE (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
           (TIME (PAST))
           (ITEM (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
           (ELSEWHERE (HOME)))

Database contains:

((PTRANS (:ACTOR  (HUMAN (:NAME (ANN))))
         (:OBJECT (HUMAN (:NAME (ANN))))
         (:TO (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TIME (PAST)))
 (MTRANS (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TIME (PAST)))
 (MBUILD (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT
            (ATRANS (:ACTOR (HUMAN (:NAME (ANN))))
                    (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
                    (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
                    (:TO (HUMAN (:NAME (ANN))))
                    (:TIME (FUTURE)))))
 (PTRANS (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT (HUMAN (:NAME (ANN))))
         (:TO (KITE (:NUMBER (SINGULAR))
         (:REF (INDEF)))) (:TIME (PAST)))
 (GRASP (:ACTOR (HUMAN (:NAME (ANN))))
        (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
        (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT (HUMAN (:NAME (ANN))))
         (:TO (CASHIER)) (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TO (HUMAN (:NAME (ANN))))
         (:TIME (PAST)))
 (ATRANS (:ACTOR (HUMAN (:NAME (ANN))))
         (:OBJECT (MONEY))
         (:FROM (HUMAN (:NAME (ANN))))
         (:TO (CASHIER))
         (:TIME (PAST)))
 (ATRANS (:ACTOR (CASHIER))
         (:OBJECT (KITE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
         (:TO (HUMAN (:NAME (ANN))))
```

```
        (:TIME (PAST)))
 (PTRANS (:ACTOR (HUMAN (:NAME (ANN))))
        (:OBJECT (HUMAN (:NAME (ANN))))
        (:FROM (STORE (:NUMBER (SINGULAR)) (:REF (INDEF))))
        (:TO (HOME))
        (:TIME (PAST)))
 ($SHOPPING (SHOPPER (HUMAN (:NAME (ANN))))
  (STORE (STORE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (TIME (PAST))
  (ITEM (KITE (:NUMBER (SINGULAR)) (:REF (INDEF)))) (ELSEWHERE (HOME))))
```

What to Submit

1. Save your definitions in `ca-defs.lisp` and your script in `sam.lisp` and submit these two files in Canvas. You shouldn't have to modify anything else in either system.

Happy Knowledge Engineering!