Homework 4
CS161, Spring 2014

Due: Wednesday, June 4th. Submit via CourseWeb.

This will be your final homework. It involves 2 parts:

1. **Inference:** inferring unstated story facts and/or causal connections between story episodes by using common sense knowledge (CSK) in semantic memory.
2. **Question and Answer:** understanding questions, finding their answers, and mapping each conceptual answer into English.

Part-1 will be achieved by applying Horn-clause style rules in a forward-chaining manner. Part-2 will be achieved by applying backward chaining and defining a simple, concept-to-English recursive descent concept-based English generator. We will restrict both parts to keep them very simple. Thus, you will only have to define 7 new Lisp functions.

Part I: The Inference Engine

Let's first consider Part I: When demons (in HW3) fired they formed new complete concepts in working memory. They should also have created structures in episodic memory (EP-MEM). For example, these demons should have created an episodic character CHARLOTTE-1 with information about who she is and what actions she did. We will skip this step and assume that these episodic structures already exist. They will be our story facts. We will also have a bit of common sense knowledge (CSK), in the form of Horn-clause rules. To apply these rules and infer new, unstated story facts will require a unifier.

Let's look at some of the function definitions we'll need in order to accomplish this task.

[!] Important:
Many algorithm steps referenced in the following function descriptions are found on page 328 of your textbook in the UNIFY algorithm listed there-on.

Problem 1.1: (SF-UNIFY farg1 farg2 beta)

This Lisp function will unify a list of slot-filler frames **farg2** with another list of frames **farg2** and return a list of bindings **beta**. An farg can be:

- A list of frames
- A single frame
- A variable. (Recall from HW1 that variables are of the form (V atom). We have ignored variables until now.)

Let us first consider the central case, namely, when **farg1** and **farg2** are each a single frame; we'll call them **frm1** and **frm2**.

SF-UNIFY will recursively go through each slot in **frm1** and look for a corresponding slot in **frm2**. If a corresponding slot is found and the fillers are recursively unified in a successful manner, then **SF-UNIFY** will return **beta** (i.e., whatever bindings **beta** already had, along with those added during the unification process). If unsuccessful, then SF-UNIFY returns NIL.

Note-1:
Since **SF-UNIFY** looks only at **frm1** for slots to be unified with corresponding slots in **frm2**, it is possible that **frm2** could have additional slots (that do not appear in **frm1**) and the unification will still be successful. So SF-UNIFY is seeking to unify only that subset of the slots in **frm2** that are found in **frm1**, so **SF-UNIFY** is slightly different from the UNIFY in your textbook.

Note-2:
SF-UNIFY will not implement the OCCUR-CHECK? function mentioned in the text.

The UNIFY in your textbook calls on a helper function UNIFY-VAR. **SF-UNIFY** should also call on a similar helper function. This helper function will be called **SF-UNIVAR**, and is described in Problem 1.2.

In your textbook, theta (θ) is a ***set*** of bindings, but since you are using Lisp, **beta** will be a ***list*** of bindings. **beta** will take the form:

(T (*atom binding*) ... (*atom binding*)), (T), or NIL.

(T) indicates successful unification in the case in which there were no variables and so there are no bindings to consider. NIL indicates that **SF-UNIFY** failed to unify. Each *atom* in **beta** will come from a *variable* appearing in frames (in **farg1** and **farg2**).

Note-3:
(as specified in HW1) frame *variables* appear in a frame only as the *filler* of a slot.

Another case in **SF-UNIFY** is when **farg1** and **farg2** are lists of frames. In this case, **SF-UNIFY** will be successful if, for *each* frame **frm1** in **farg1**, there is a frame **frm2** in **farg2** that unifies using bindings **beta**. So SF-UNIFY is looking for that subset of frames in **farg2** that unifies with the frames in **farg1**. So **farg2** can have more frames than **farg1**.

Example 1:

Here are some cases with which to test your **SF-UNIFY** function. First, cases with no variables:

```
FRM-0 = (SEE AGENT (A)
             OBJECT (THROW AGENT (B)
                           OBJECT (D)))
FRM-1 = (SEE AGENT (A)
             LOC (HOME-1)
             OBJECT (THROW OBJECT (D)
                           AGENT (B)))
```

(SF-UNIFY   FRM-0   FRM-1   '(T))
returns: (T)   ; Since a unification exists without any substitutions

(SF-UNIFY   FRM-1   FRM-0   '(T))
returns: NIL   ; Since FRM-1 has more slots than FRM-0

(SF-UNIFY   FRM-0   FRM-1   '(T (X01 (JOE))))
returns: (T (X01 (JOE)))   ; Since input beta was '(T (X01 (JOE))) and needed no new unification

(SF-UNIFY FRM-1 FRM-0 '(T (X01 (JOE))))
returns: NIL   ; Since unification still fails

Example 2.1:

Here are cases with variables:

```
FRM-2 =
(SEE   AGENT (CHARLOTTE-1)
       OBJECT (THROW AGENT   (CHARLES-1)
                     OBJECT (V Y1)
                     INTO (TRASH-1))
       LOC (V LOC1))

FRM-3 =
(SEE AGENT (V X1)
      LOC (HOME-1)
      OBJECT (THROW AGENT (CHARLES-1)
                    OBJECT (PRINTED-1)
                    INTO (TRASH-1)
                    MANNER (SECRETLY)))
```

(SF-UNIFY   FRM-2   FRM-3   '(T))
returns:
(T (X1 (CHARLOTTE-1)) (Y1 (PRINTED-1)) (LOC1 (HOME-1)))

```
(SF-UNIFY   FRM-3   FRM-2   '(T))
returns:   NIL   ; Since FRM-3 has more slots (in the OBJECT filler) than does FRM-2
```

Example 2.2:

```
FRM-4 = (V X)

(SF-UNIFY   FRM-4   FRM-2   '(T))
returns:
(T (X (SEE AGENT (CHARLOTTE-1)
           OBJECT (THROW AGENT (CHARLES-1)
                         OBJECT (V Y1)
                         INTO (TRASH-1))
           LOC (V LOC1))))
; Since the entire frame of FRM-2 is substituted for X!
```

Example 2.3:

```
FRM-5 = (KILLS   AGENT (V Z1)   OBJECT (V Z1))
FRM-6 = (KILLS   AGENT (EMMA-1)   OBJECT (V Z2))

(SF-UNIFY   FRM-5   FRM-6   '(T))
returns:   (T (Z1 (EMMA-1))   (Z2 (EMMA-1)))
```

Example 3:

Here's a case where farg1 and farg2 are lists frames:

```
L-FRMS-1 =
(   (SEE AGENT (V AGG1)
         OBJECT (THROW AGENT (V AGG2)
                       OBJECT (V OBJJ1)
                       INTO (V INTOO1)
                       MANNER (SECRETLY))
         LOC (V LOCC1))

    (MYSTERIOUS AGENT (V AGG2))

    (STORY OBJECT (V OBJJ1)
           ABOUT (CRIME))
)
```

L-FRMS-2 =
((SEE **AGENT** (CHARLOTTE-1)
            OBJECT (THROW **AGENT** (CHARLES-1)
                                      OBJECT (PRINTED-1)
                                      INTO (TRASH-1)
                                      MANNER (SECRETLY))
            LOC (HOME-1))

    (STORY **OBJECT** (PRINTED-1)
                ABOUT (CRIME))

    (MYSTERIOUS **AGENT** (CHARLES-1))

    (UNCLE-OF **AGENT** (CHARLES-1)
                      OBJECT (CHARLOTTE-1))
)

(SF-UNIFY   L-FRMS-1   L-FRMS-2   '(T))
returns:
(T (AGG1 (CHARLOTTE-1))   (AGG2 (CHARLES-1))   (OBJJ1 (PRINTED-1))
    (INTOO1 (TRASH-1))   (LOCC1 (HOME-1)))

(SF-UNIFY   L-FRMS-2   L-FRMS-1   '(T))
returns: NIL   ; Since (UNCLE-OF ...) in L-FRMS-2 does not unify with anything in L-FRMS-1

[!] Hints:

- The book uses predicates VARIABLE?, COMPOUND?, and LIST?, respectively, to determine if inputs are variables, frames, or lists of frames, respectively. You would be wise to use similar helper functions!

- The book uses the following names in its algorithm to talk about familiar concepts in frames:
    - **x.OP**: Refers to the predicate of frame x
    - **x.ARGS**: Refers to the slot-filler pairs of frame x (i.e., with the predicate removed; in other words, (rest x)).

    - **x.REST**: When x is a *list* of frames, x.REST refers to all but the first frame in the list
    - **x.FIRST**: When x is a *list* of frames, x.FIRST refers to the first frame in the list

- For frames, we know that variables will appear only as fillers, so we need to unify them with the corresponding filler in the other frame (i.e., corresponding to the same slot), but remembering to check for matching predicates as well!

- For lists of frames, each frame in **farg1** should unify with *any* frame in **farg2**, but we don't care about the order of unification so long as a binding is found (the book implies they must be in the same order too but our frames don't really care about that). NB: Multiple frames in **farg1** MAY NOT unify to the same frame in **farg2**

- As a corollary to the above hints, because of the different formats of possible unification, you might also make helper functions that deal with input frames and input lists of frames separately. Just a suggestion though!

- Assume **beta** is passed in as (T   (*atom   binding*)*)

- You may assume that all fillers will be non-null.

- If a variable in farg1 (let's say, (V X)) matches a variable in farg2 (let's say, (V Y)), then the variable in farg1 should be bound to the variable name in farg2.

  Example:

  (SF-UNIFY   '(HUMAN **F-NAME** (V X))   '(HUMAN **F-NAME** (V Y)))
  returns:
  (T (X (V Y)))

Problem 1.2: (SF-UNIVAR var x beta)

Helper function for SF-UNIFY, that takes in a variable **var** (i.e., a list formatted as (V atom)), a frame, variable, or a frame list **x** (just like as described in Problem 1.1), and a binding list **beta** (just like as described in Problem 1.1). This function will follow the pseudocode for the UNIFY-VAR function on page 328 of the textbook, *except* for the OCCUR-CHECK? step, which we omit entirely for this project.

SF-UNIVAR returns a substitution list, **beta**, which consists of the valid variable bindings that we make along the way during our unification process. In this capacity, beta is "built up" as we're trying to find bindings for variables, and so SF-UNIFY will call SF-UNIVAR, which will then look for a binding for **var** to **x** by recursively calling SF-UNIFY again, etc. and so forth.

Here is the pseudocode with explanation:

```
(defun SF-UNIVAR (var   x   beta)
    (cond
        ; ...where var is a variable and val is the binding of var in beta:
        ((var is in the binding list beta)   (SF-UNIFY   val   x   beta))
        ; ...where x is a variable and val is the binding of x in beta:
         ((x is in the binding list beta)   (SF-UNIFY   var   val   beta))
        ; ...otherwise, neither var nor x have bindings in beta, so add the binding to beta
         (t (add-binding   var/x   beta))
    )
)
```

Examples:

```
(SF-UNIVAR   '(V   VAR1)   '(CHARLOTTE-1)   '(T))
returns:
(T (VAR1   (CHARLOTTE-1)))   ; Since neither VAR1 isn't in beta, and CHARLOTTE-1 isn't a
variable, then we simply tack on the binding to beta!

(SF-UNIVAR   '(V   VAR1)   '(V   VAR2)   '(T   (VAR1   (CHARLOTTE-1))))
calls SF-UNIFY with:
(SF-UNIFY   '( CHARLOTTE-1)   '(V   VAR2)   '(T   (VAR1   (CHARLOTTE-1))))

(SF-UNIVAR   '(V   VAR1)   '(V   VAR2)   '(T   (VAR2   (CHARLES-1))))
calls SF-UNIFY with:
(SF-UNIFY   '(V   VAR1)   '( CHARLES-1)   '(T   (VAR2   (CHARLES-1))))
```

Note:
Visibly, SF-UNIVAR and SF-UNIFY are intertwined and part of the same unification task, which is why they are described as separate problems but will be graded together.

Problem 2: (SF-SUBST   frame   beta)

This function looks for each variable in **frame** and if that variable has a binding in **beta**, then SF-SUBST returns **frame** with those bindings replacing the corresponding variables.

Example:

```
FRM-2 =
(SEE   AGENT (CHARLOTTE-1)
       OBJECT (THROW AGENT   (CHARLES-1)
                     OBJECT (V Y1)
                     INTO (TRASH-1))
       LOC (V LOC1))

(SF-SUBST   FRM-2   '(T (LOC1 (HOME-1))   (Z1 (GOD-1))   (Y1 (PRINTED-1)))
returns:
(SEE AGENT (CHARLOTTE-1)
      OBJECT (THROW AGENT (CHARLES-1)
                    OBJECT (PRINTED-1)
                    INTO (TRASH-1))
      LOC (HOME-1))
```

Note:
The extra binding of (Z1 (GOD-1)) never gets used in the (SEE …) frame but that is fine.

Problem 3: (SF-INFER   rule   facts)

This function takes a single **rule** and a list of **facts**, attempts to unify the premises in the IF-part of **rule** with a subset of the **facts**, and if successful, returns the SF-SUBST of the THEN-part, using the bindings created by SF-UNIFY.

A **rule** is represented as a list of the form:
(IF frame-1 frame2 … frame-n THEN frame)

Facts are lists of zero or more frames:
(frame*)

Note:
SF-INFER stops once it has found a subset of **facts** that unifies with **rule's** premises. It does not look for other subsets of **facts** that might also unify with **rule**'s IF-part.

Example 1:

```
RULE-1 =
(IF (SEE AGENT (V AGG1)
         OBJECT (THROW AGENT (V AGG2)
                       OBJECT (V OBJJ1)
                       INTO (V INTOO1)
                       MANNER (SECRETLY))
         LOC (V LOCC1))

    (MYSTERIOUS AGENT (V AGG2))

    (STORY OBJECT (V OBJJ1)
           ABOUT (CRIME-1))

    (IDENTITY AGENT (UNKNOWN)
              OF (CRIME-1))

 THEN
    (BELIEVES AGENT (V AGG1)
              OBJECT (COMMITTED AGENT (V AGG2)
                                OBJECT (CRIME-1)))
)
```

```
FACTS-1 =
(   (UNCLE-OF **AGENT** (CHARLES-1)   **OBJECT** (CHARLOTTE-1))
    (MYSTERIOUS **AGENT** (CHARLES-1))
    (STORY **OBJECT** (PRINTED-1) **ABOUT** (CRIME-1))
    (SEE **AGENT** (CHARLOTTE-1)
            **OBJECT** (THROW **AGENT** (CHARLES-1)
                               **OBJECT** (PRINTED-1)
                               **INTO** (TRASH-1)
                               **MANNER** (SECRETLY))
            **LOC** (HOME-1))

    (IDENTITY **AGENT** (UNKNOWN)
                   **OF** (CRIME-1))
)

(SF-INFER   RULE-1   FACTS-1)
returns:
(BELIEVES **AGENT** (CHARLOTTE-1)
                **OBJECT** (COMMITTED **AGENT** (CHARLES-1)
                                              **OBJECT** (CRIME-1)))
```

Problem 4: (FORWARD1 rules)

This function is a restricted version of forward chaining. FORWARD1 accesses a global variable EP-MEM that contains a list of story facts. For each rule in **rules**, FORWARD1 calls on SF-INFER once to infer a conclusion for that rule. If a conclusion is inferred (or if SF-INFER returns NIL) then FORWARD1 will go on to try the next rule, until all rules have been tried. FORWARD1 will store all the conclusions in a local variable. Call this variable NEW-FACTS. If NEW-FACTS is not NIL, then FORWARD1 will append NEW-FACTS to the front of EP-MEM and recursively call itself until no NEW-FACTS are returned. Upon termination, FORWARD1 returns a list of all the new facts discovered, which have also been added to EP-MEM.

FORWARD1 is restricted because it does not try to generate all possible conclusions from a given rule and a set of facts. It just generates the first conclusion it can infer and then goes on to the next rule. FORWARD1 is also restricted in that it does not bother to standardize-apart the variable names--nor does it bother to test if new conclusions are re-namings of existing conclusions.

We are only going to consider 3 simplified CSK rules (of the many that would be needed to handle the MWM story). You have already seen RULE-1, which can be described in English as:

Rule-1:
"If x sees y with a mysterious past secretly throw z away and z turns out to be a story about a crime and the person who committed the crime is unknown, then x will believe that y is the person who committed that crime."

Here are the other 2 CSK rules:

Rule-2:
"If x believes that y committed some crime and x is a relative of y then x will feel conflicted."

```
RULE-2 =
(IF (BELIEVES AGENT (V AGG4)
              OBJECT (COMMITTED AGENT (V AGG5)
                                OBJECT (CRIME-1)))

    (RELATIVE-OF AGENT (V AGG5)
                 OBJECT (V AGG4))

  THEN
    (FEELS AGENT (V AGG4)
           STATE (CONFLICT))
)
```

Rule-3:
"If x is the Uncle of y then y is a relative of x"

```
RULE-3 =
(IF (UNCLE-OF AGENT (V AGG6)
              OBJECT (V AGG7))

  THEN
    (RELATIVE-OF AGENT (V AGG6)
                 OBJECT (V AGG7)))
```

Example:

```
CSK-RULES = (LIST RULE-1 RULE-2 RULE-3)
EP-MEM = FACTS-1   ; From above

(FORWARD1   CSK-RULES)
returns:
(   (FEELS AGENT (CHARLOTTE-1)
           STATE (CONFLICT))

    (RELATIVE-OF AGENT (CHARLES-1)
                 OBJECT (CHARLOTTE-1))

    (BELIEVES AGENT (CHARLOTTE-1)
              OBJECT (COMMITTED AGENT (CHARLES-1)
                                OBJECT (CRIME-1)))
)
```

Part II: Question & Answer

Part-2 is restricted to just answering "Why" questions.   It involves matching the THEN-part of a rule to return the IF-part as the conceptual answer. It is used largely to provide human-readable answers to questions posed to our inference engine.

Problem 5: (BACKWARD1   query   rules)

This function takes frame as a **query**. BACKWARD1 looks for the first rule in **rules** whose THEN-part SF-UNIFYs with **query**. If successful, it returns a list of frames, from the IF-part of that rule, with any variables in the IF-part replaced by their bindings (via SF-SUBST).

In other words, BACKWARD1 does only the *first step* of the entire backward chaining algorithm, by finding a rule consequent that successfully unifies to the input query, and then returning the conditions for that consequent with the proper binding substitutions (required to unify the consequent with the query) in place.

Example:

```
BFRAME =
(BELIEVES AGENT (CHARLOTTE-1)
          OBJECT (COMMITTED AGENT (CHARLES-1)
                            OBJECT (CRIME-1)))

(BACKWARD1   BFRAME   CSK-RULES)
returns:
(   (SEE AGENT (CHARLOTTE-1)
         OBJECT (THROW AGENT (CHARLES-1)
                       OBJECT (V OBJJ1)
                       INTO (V INTOO1)
                       MANNER (SECRETLY))
         LOC (V LOCC1))
    (MYSTERIOUS AGENT (CHARLES-1))
    (STORY OBJECT (V OBJJ1)
           ABOUT (CRIME-1))
    (IDENTITY AGENT (UNKNOWN)
              OF (CRIME-1))
)
```

Note:
Notice that, in what BACKWARD1 returns, only grounded terms in the THEN-part are substituted into the premises of the IF-part and so the variables OBJJ1, INTOO1 and LOCC1 remain unbound. If BACKWARD1 were to do more backward chaining, then it would attempt to unify those returned premises with other facts and also with other frames in the THEN-part of other rules.   However, BACKWARD1, as defined in HW4,   just performs the first step.

Now you are ready to answer questions about the MWM story, but we will skip the process of writing question-word demons (like "why", "who", "where" etc.) and we will skip the firing of those and other phrase demons), which would be required to map an English question into its conceptual representation. We will also skip a function needed to determine format answers, based on the type of question. For example, a "Why" question might require an answer format that begins "Because …".

We will also assume that a function (ANSWER question-concept) already exists and has called the function BACKWARD1 to answer the question "Why does Charlotte believe that Charles murdered Samantha?"

That is, we will assume that some function (ANSWER q-con1), in the case where:

```
q-con1 =
(REASON-FOR ANTE (V WHAT)
            CONSEQ (BELIEVES AGENT (CHARLOTTE-1)
                             OBJECT (COMMITTED AGENT (CHARLES-1)
                                               OBJECT (CRIME-1))))
```

...has already returned the conceptual answer:

```
C-ANS-1 =
(  (SEE AGENT (CHARLOTTE-1)
        OBJECT (THROW AGENT (CHARLES-1)
                      OBJECT (PRINTED-1)
                      INTO (TRASH)
                      MANNER (SECRETLY))
        LOC (HOME-1))

   (MYSTERIOUS AGENT (CHARLES-1))

   (STORY OBJECT (PRINTED-1)
          ABOUT (CRIME-1))

   (IDENTITY AGENT (UNKNOWN)
             OF (CRIME-1))
)
```

The last thing to do for part-2 is to define a simple *conceptual generator* that can express a concept (an instantiated frame) as an English sentence.

Problem 6: (C-GEN   frame   eng-pats)

This function takes a **frame** and a list of associated English patterns **eng-pats** and uses those to generate each frame element into English.

eng-pats is an association list where each frame predicate has a list of slots and interleaved English phrases associated with it. Each phrase is preceded by PHR. C-GEN recursively calls upon itself to generate each slot specified and returns a list of English words. If a predicate to a frame is not in eng-pats, then C-GEN does not attempt to generate its slots and returns the predicate itself.

Note:
Slot names below are indicated in **bold**. English words are in a list preceded by PHR. Predicates are at the front of each association list.

ENG-PATS1 =
((SEE (**AGENT** (PHR SAW) **OBJECT AT** (PHR AT) **LOC**))
     (MYSTERIOUS (**AGENT** (PHR HAS AN UNKNOWN PAST)))
     (CRIME-1 ((PHR THE MURDER OF SAMANTHA)))
     (THROW (**AGENT MANNER** (PHR THROW) **OBJECT** (PHR INTO) **INTO**))
     (STORY ((PHR THE STORY TOLD IN) **OBJECT** (PHR IS ABOUT) **ABOUT**))
     (CHARLES-1 ((PHR UNCLE CHARLES)))
     (TRASH ((PHR A TRASH CAN)))
     (PRINTED-1 ((PHR A NEWSPAPER CLIPPING)))
     (CHARLOTTE-1 ((PHR CHARLOTTE)))
     (HOME-1 ((PHR HOME)))
     (RELATIVE-OF (**AGENT** (PHR IS A RELATIVE OF) **OBJECT**))
     (IDENTITY ((PHR THE IDENTITY OF PERSON INVOLVED IN) **OF** (PHR IS) **AGENT**))
     (BELIEVES (**AGENT** (PHR BELIEVES THAT) **OBJECT**))
     (COMMITTED (**AGENT** (PHR COMMITTED) **OBJECT**))
)

Examples:

(C-GEN '(THROW **AGENT** (CHARLES-1)
                    OBJECT (PRINTED-1)
                    INTO (TRASH)
                    MANNER (SECRETLY))

          ENG-PATS1)
returns:
(UNCLE CHARLES SECRETLY THROW A NEWSPAPER CLIPPING INTO A TRASH CAN)

(C-GEN '(CO-HABITATE **AGENT** (EMMA-1))   ENG-PATS1)
returns: (CO-HABITATE)

Problem 7: (C-GENS frames eng-pats)

This simple function calls on C-GEN to generate a sentence for each frame in **frames**, separated by the word AND.

Example 1:

The answer to: "Why does Charlotte believe that Charles murdered Samantha?"
is C-ANS-1 (above) which can now be translated into English by C-GENS.

(C-GENS C-ANS-1 ENG-PATS1) returns:

((CHARLOTTE SAW UNCLE CHARLES SECRETLY THROW A NEWSPAPER CLIPPING INTO A TRASH CAN AT HOME) AND (UNCLE CHARLES HAS AN UNKNOWN PAST) AND (THE STORY TOLD IN A NEWSPAPER CLIPPING IS ABOUT THE MURDER OF SAMANTHA) AND (THE IDENTITY OF PERSON INVOLVED IN THE MURDER OF SAMANTHA IS UNKNOWN))

Example 2:

Question: Why does Charlotte feel conflicted?

C-ANS-2 =

```
(   (BELIEVES AGENT (CHARLOTTE-1)
              OBJECT (COMMITTED AGENT (CHARLES-1)
                                OBJECT (CRIME-1)))
    (RELATIVE-OF AGENT (CHARLOTTE-1)
                 OBJECT (CHARLES-1))
)
```

(C-GENS C-ANS-2 ENG-PATS1) returns:

((CHARLOTTE BELIEVES THAT UNCLE CHARLES COMMITTED THE MURDER OF SAMANTHA) AND (CHARLOTTE IS A RELATIVE OF UNCLE CHARLES))

Assignment Restrictions & Tips

- **Start Early!**

- **Late Policy**
  Submit ~0 – 24 hours late and you'll lose 30% off the bat; submit more than a day late and it's a 0.

- **Using Previous HW Solutions**
  You may use any of the previous HW solutions we post on CourseWeb to assist your HW4 solutions, as these will be included for the grading tests. Any revisions to functions in HWs 1 – 3 that you might find useful to make will over-write the solution editions so long as you include the revisions in your HW4 submission.

- **Helper Functions Are OK!**
  Feel free to employ helper functions to be used within any of your homework solutions.

- **Make Unit Tests!**
  You should practice test-driven development for this homework, which might include a test suite that loads your function definitions and operates on some of the frames we provide.

- **Your Function Names Must Match These Exactly!**
  Failure to name your functions exactly as they appear in this homework will result in a grade of 0 for that problem!

- **Unfinished Functions**
  If you run out of time on a function, make sure you at least include that function definition and simply return 'UNIMPLEMENTED.

- **What to Submit**
  Submit one file, titled "*your-ID-number-here*.lsp" with all of your function definitions, and any helper functions you employed. DO NOT SUBMIT UNIT TESTS.

  E.g. I might submit a file named "555555555.lsp"