

Homework Assignment #1: Semantic Sentence Realization**Due date: February 12, 2012**

This assignment illustrates knowledge representation, grammar, discourse structure, and non-learning rule-based language generation all together. It has four parts. In the first part you first define a knowledge representation scheme and create instance frames representing sentences. In part 2 you build a sentence generator that makes random sentences. In Part 3 you extend it to be able to convert your frames into English sentences. Part 4 of this assignment illustrates a computational approach to discourse (multi-sentence) text processing. This part has four sections. First you must connect the sentence-sized meaning frames you defined earlier. Then you must design the text plans (RST schemas). Then you create a new version of your sentence realizer to accept a communicative goal plus the network of frames, and finally from that you can plan and generate several coherent multi-sentence texts.

Please email all results SHOWING ALL THE REQUESTED DETAILS by the due date to dirkh@isi.edu.

Part 1. Representation (35 points)

As you know, representing meaning is tricky. Design a system for representing the meanings of simple sentences and then use it to represent the sentences given below. Try to be parsimonious if you can: create as few basic primitives ‘atoms’ of meaning as possible and then try to compose together these atoms into larger composites. Create representation frames/schemas for the following types of meaning:

- **Objects** (e.g., table, chair, wheel, car, human, child, cat, mountain, idea, justice, happiness). Use relations such as *:id*, *:type*, *:color*, *:age*, *:size*, *:quality*... and the quasi-syntactic slot *:identifiable*.
- **Events and actions** (e.g., knowing, playing, using, swimming, thinking, giving, taking, selling). Use relations such as *:id*, *:type*, *:agent*, *:patient*, *:information*, *:location*, *:instrument*, *:manner*, *:source*, *:destination*, *:beneficiary*, *:time*, *:location*, the pragmatic slots *:speechact* (for Assertion, Question, and Order) and *:polarity* (for Pos and Neg), and the inter-event connectives *:temp-after*, *:temp-during*, *:condition*, *:cause*... etc., as in:

He ate a peach hurriedly

((:id E11) (:type EAT) (:agent HE) (:patient PEACH) (:manner HURRIEDLY) (:time PAST))

- **Qualities** (e.g., red, happy, fast, wild, relaxed, large, small, old). Use relations such as *:id*, *:type*, *:weight*, *:age*, *:size*, *:polarity*, etc., as in:

The heavy old chair

((:id C0) (:type CHAIR) (:weight HEAVY) (:age OLD))

He ate a big peach

((:id E11) (:type EAT) (:agent HE) (:patient ((:id E3) (:type PEACH) (:size BIG)))) (:time PAST))

- **States** (e.g., having a headache, having a stomach ache, being happy, being very happy, wanting). Use relations such as *:id*, *:type*, *:experiencer*, *:attribute*, *:degree*, *:polarity*, as in:

John has a severe headache

((:id K0) (:type STATE) (:experiencer JOHN) (:attribute ((K1 :type HEADACHE :degree +7))))

- **State changes** (e.g., healing from the flu, becoming happy, and then very happy, growing tall). Use relations such as *:id*, *:type*, *:experiencer*, *:attribute*, *:old-degree*, *:new-degree*, *:old-value*, *:new-value*, *:polarity*

An example representation frame for *Did the man from Oklahoma eat a small red peach?* is:

```
((:id X15)
 (:type EAT)
 (:agent ((:id X16)
 (:type MAN)
 (:source OKLAHOMA)
 (:identifiable T)))
 (:patient ((:id X17)
 (:type PEACH)
 (:size SMALL)
 (:color RED)
 (:identifiable F)))
 (:time PAST)
 (:speechact QUESTION)
 (:polarity POS))
```

Look in the class handout for other examples, and choose how ‘deep’ (parsimonious) you want your representation primitives to be.

Create (for 8 points) on paper a small ontology of the various representational classes in your system, and list for each class the entities/events you need to use for your sentences:

TOP

OBJECT

PHYSICALOBJECT (e.g., CAR, PEACH)

NONPHYSICALOBJECT (e.g., IDEA, DEMOCRACY)

EVENT

QUALITY

STATE

STATE-CHANGE

SPEECHACT

RELATION

EVENTROLE

OBJECTRELATION

QUALITYRELATION

STATEANDSTATECHANGERELATION

Now represent the following sentences (for 3 points each):

1. At In-N-Out, we know that nothing complements a hamburger like tasty hot french-fries .
2. Regarding french-fries, we feel that fresh is the best [way] .
3. A french-fry is only as good as the potato [it comes from] .
4. So we use fresh large potatoes .
5. And we deliver them fresh to our stores .
6. Every day, we take new whole potatoes .
7. We prepare the potatoes individually .
8. Then we cook them in trans-fat-free vegetable oil .

9. We do not use a freezer .

*****To hand in for Part 1: The ontology of terms, and your representation(s) of these 10 sentences.**

Part 2. Random Sentence Realization (20 points)

Write a random sentence generator.

As explained in class, to make a realizer, we have the system produce outputs along the way, thereby changing the RTN into an Augmented Transition Network (an implementation is described in (Simmons & Slocum, 1972)). Whenever it crosses over an arc, the system has to generate the constituent stated on the arc. If the constituent is simple (just a word), the system simply outputs the word and continues. If it is complex (that is, another constituent, which itself may be a sequence of states), the system has to temporarily suspend the current ATN, push into the new constituent's ATN, and realize that. When it has successfully completed the new one, the system pops back to the old ATN, back to the sequence that initiated the push, and continues.

Algorithm: Use a stack to hold unprocessed material.

START: Place "S" on the stack

GEN:

1. pop the next item on the stack; call it I
2. if I is a lexical item, print it out
 else call EXPAND with I and push the results onto the top of the stack
3. unless the stack is empty, go to 1

EXPAND (X):

1. if X is a word class in the lexicon, return a word randomly selected from that class
2. if X is the head of a grammar rule, return an expansion randomly chosen from that rule
3. else there's an error

Doing this, you can generate grammatical English sentences, starting by loading "S" onto the stack, or noun phrases, starting by loading "NP" on the stack.

Create a grammar (with at least 10 rules) and a lexicon (with at least 5 word classes, each non-closed-class containing at least 10 words), and implement an algorithm to generate syntactically correct but possibly meaningless sentences. For this:

- lexicon: 3 points
- grammar: 3 points
- algorithm: 5 points
- run of system (with at least 2 traces showing interesting intermediate processing points): 4 points

*****To hand in for Part 2: mail the code, the data sources (lexicon and grammar), and program traces of the production of 2 random sentences.**

Part 3. Meaning-Based Sentence Realization (45 points)

Extend your realizer to accept the shallow semantic input representations you created in Part 1, and to generate the appropriate sentences. As discussed in class, you will have to augment the grammar rule representation to specify which portion of the input representation it addresses. You might also have to build a selection mechanism to decide which rule applies to the input, or else a backtracking mechanism to recover from incompletely expressed inputs.

Take as input a single sentence at a time.

In order to generate *meaningful* sentences, you can still use ATN traversal system outlined in class and developed in Part 2, but it has to be extended. While you still follow the arcs and output words, two things change:

- arc choice is no longer random,
- lexical choice is no longer random.

Both these decisions must somehow be guided by the input. Each arc will express some portion of the input. As you travel ‘down’ through the grammar, the portions of the input you consume get smaller and smaller, bottoming out in words. To do this, extend the grammar’s rules with links to the input:

```
1. S[head]           → NP[head:agent] VP[head]
                   → AUX[head:aux] NP[head:agent] VP[head]
                   → NP[head:agent] AUX[head:aux] VP[head]
```

and so on, for the other rules. Also extend the lexicon itself:

```
N[man]      → singular: man
            → plural: men
```

The first rule above for “S” means: when you want to make a sentence “S” with the input frame (linked to the variable “head”), then you must place on the stack two new things: “NP” with the piece of the frame that fills the :agent slot, and “VP” with the whole frame again. Then the “NP” rule will specify further how to decompose the :agent part of the input frame into smaller pieces, eventually into words.

To do this, you can take the following approach. Instead of placing on the stack a single symbol like “S” or “NP”, create a structure or object to place on the stack. Inside that object, record the syntactic type (“S” or “NP”) but also include with it the appropriate piece of the representation that this symbol must work on (for “S”, include the whole sentence frame (or its pointer); for the NP, include just the :agent part of the frame (or its pointer), and so on.

Now, when you expand the stack top, you consider both the symbol and the part of the input frame that it is working on. You get the appropriate grammar rule, bind its left hand side variable to this frame part, and then decompose the frame as specified by the right hand side part to build new stack structures, each with its symbol and corresponding frame part. (You can at the same time handle agreement between subject and verb, like “singular” and “plural”, if you like, using this object.)

Now extend the basic algorithm:

Input: above shallow or deep semantics

Grammar: above rules, as extended

Lexicon: as above

Algorithm:

1. get the top of the stack, bind its head to the head of the input, and push the result onto stack
- 2a. pop the top of the stack
 - b. if it is a lexical item, print it
 - c. else: find an appropriate expanded rule for the top from the grammar
(if there are alternatives, execute some selection function)
for each portion of this rule, find the approp. portion of the input, as specified
in the portion (this may be a word) and bind it into the rule

- perform any side effects (number agreement)
- push the list of bound portions back onto the stack
- 3. unless stack is empty, go to 2a

You will note that some things that look simple are quite difficult to generate. It's ok to generate a paraphrase—a sentence that means the same but says it in a different way—but please explain what the trouble was in such cases.

*****To hand in for Part 3: mail in ALL the following:**

- the realizer code (20 points),
- the new grammar (5 points),
- the lexicon (5 points),
- the outputs generated for these inputs (and more, if you wish) (2 points each for the 10 sentences),
- a trace of the system running on at least two sentences, showing interesting processing stages (5 points),
- a discussion of what you found problematic and what not, and why (extra credit).

Extra credit (up to 10 points) if your generator can handle the last two sentences appropriately as mentioned in Part 1).

You may note that your system could say the same input in various ways; at the very least, you might have different words for the same thing, like *say* and *state* or *car* and *auto*. Why would you choose one over the other? How would you represent the information required to make the choice? You might also be worried by that fact that it will be hard to make the grammar more powerful without building in some kind of backtracking. For example, if your grammar had the two NP rules

NP → PRONOUN
NP → DET ADJ* N

Then every :agent will always become “he” or “she” or “it”, and you will never be able to say a full NP! Ideally, you'd build a backtracking mechanism into the loop, which would make sure that the whole input frame was actually said, or if not, would cause another grammar rule to be chosen as appropriate. If you can write your grammar simply enough to avoid this problem, good for you. But it's nicer (and you get even more extra credit, up to another 10 points) if you do implement some kind of backtracking or lookahead capability.

Part 4. Paragraph Planning

This part of the assignment illustrates a computational approach to discourse (multi-sentence) text processing. This part has four sections. First you must connect the sentence-sized meaning frames you defined earlier. Then you must design the text plans (RST schemas). Next, you create a new version of your sentence realizer to accept a communicative goal plus the network of frames, and finally from that generate several coherent multi-sentence texts.

Part 4.1: Text meaning network (10 points)

To the sentences given in the previous assignment, add the appropriate inter-frame links to connect your representation frames into a network. You were given 9 sentences; let's call them E1 to E9 here. To link

frames, invent the appropriate semantic relation(s) and add it/them to each representations. For example, if you believe sentence 3 (and following) provides more details about (i.e., elaborates) sentence 2, create a *:details* link between sentences 2 and 3:

2. Regarding french-fries, we feel that fresh is the best [way] .
3. A french-fry is only as good as the potato [it comes from] .

which can be represented as

```
((:id E2)
 (:type FEEL)
 (:agent ((:id X16)
 (... ..)))
 (:patient ((:id X17)
 (... ..)))
 (:details E3))          ← new DETAILS relation
```

and

```
((:id E3)
 (:type BE-EQUIVALENT)
 (:agent ((:id X22)
 (... ..)))
 (:patient ((:id X23)
 (... ..)))
 (:causes E4))          ← new CAUSES relation
 (:causes E9))          ← new CAUSES relation
```

(Note that perhaps the DETAILS relation should be *inside* X17, not parallel to it.)

At least the following links seem reasonable, given the story and what we know about the world:

```
E1 DETAILS E2
E2 DETAILS E3
E3 CAUSES E4-E8      E3 CAUSES E9
E4 NEXT E5           E5 NEXT E6           E6 NEXT E7           E7 NEXT E8
E1 DETAILS E3       E1 DETAILS E8
E4 OPPOSES E9
```

You can probably imagine more as well.

Even with these relations added to the frames, your sentence realizer should still work; it will simply ignore the new links.

*****To hand in for 4.1: Email a drawing or printout of the network of linked frames. If you feel something needs to be explained or justified, please add that.**

Part 4.2: Text plans (15 points)

Now we have a network of frames. How can we tell coherent stories as subsets of this network of sentences? One (short) story is simply the sequence of the events E4 to E8, following the *:next* links, which of course represent *temporal sequence*. For this, you will need a very simple RST relation/plan:

```
TemporalSequence (?X)
Goal: (tell-next-action ?X)
Nucleus: (make-sentence ?X)
Satellite: (tell-next-action ?Y)
Nucleus+Satellite: (?X :next ?Y)
Nucleus Growth Points: nil
Satellite Growth Points: (tell-next-action ?Y)
Order: (Nucleus Satellite)
```

Cue word: “then” or “next” or “after that”

Another way to write the above plan is like a grammar rule: (make-sentence E4) — position 1
tell-next-action[head] → make-sentence[head] "then" tell-next-action[head:next].
(tell-next-action E5) — position 2

So you can use almost the same realizer that you built before, now as a paragraph planner. Assuming the TemporalSequence goal is loaded to the stack with E4 matched to ?X, the planner will then match E5 to ?Y (because of the *:next* link between them); cause E4 to be sent to the realizer (in the Nucleus action); (perhaps) add “then” or “next” to be realized; and (in the recursive Satellite action) will re-load a new TemporalSequence goal onto the stack, now with E5 bound to ?X. So with just this plan defined, starting your planner with

TemporalSequence (E4)

should result in a chain of

(make-sentence E4) “then” (make-sentence E5) “next” ... (make-sentence E8)

statements, each ready for the realizer.

For each relation you have defined, you will need to define an RST plan operator to handle it. At least, you will need RST relation/plans for **Cause** (signaled in English by, for example, “therefore”) and **Elaborate** (signaled by, for example, “so”). What Nucleus or Satellite growth points do you need, if any?

*****To hand in for 4.2: Represent all the plans you need, in your own formalism, and mail them in.**

Part 4.3: RST text structure planner (10 points)

Now you need to make a fresh copy of the sentence realizer you built earlier and adapt it to plan a tree structure of sentence frames (not words). Mainly you must change three things:

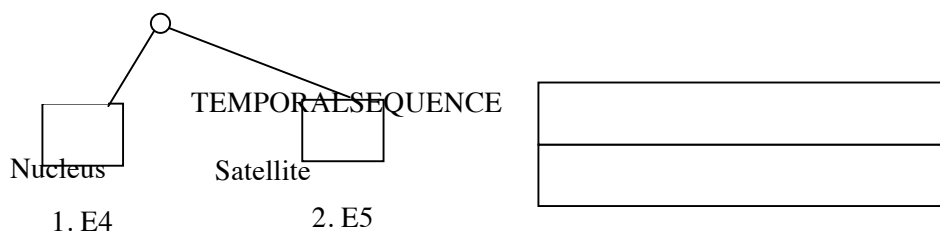
- the system’s rules: from grammar rules to RST plans,
- its input: from a sentence frame to an RST communicative goal,
- its output: no longer a lexical item, but sequence of calls to the realizer, each one pointing to the appropriate frame(s), and interspersed with the appropriate cue phrases/words.

*****To hand in for 4.3: Create a planner by copying and appropriately changing your realizer. Print out and email in the code.**

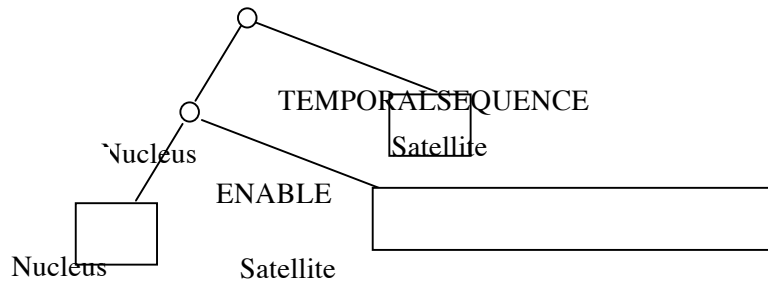
Part 4.4: Demo time! (15 points)

Finally, you are ready to demonstrate the planner.

Start the simplest story by giving the text planner the goal (*tell-next-action E4*). This goal is loaded onto the stack. The planner’s loop starts; it pops off this goal, and searches for a text plan that can achieve the *tell-next-action* goal. Of the text plans available, the Goal field of the plan TemporalSequence matches, with ?X bound to E4. So it instantiates this plan on the Discourse Structure (the evolving tree), binds E4 to ?X in the Nucleus position, finds that E4 is linked to sentence E5 through the *:next* link, binds E5 to ?Y, and places that in the Satellite position. The Discourse Structure and the stack now look like this:



Since there is a Satellite Growth Point, it is added to the stack as (*tell-next-action E5*). . (NOTE: If there *had* been a Nucleus Growth Point, for example (*tell-enabled ?X*), then this growth point's goal would also have been added to the stack, also pointing to position 1. If later the loop had expanded this goal, using the Enable plan, then its material would also have grown out of position 1 of the Discourse Structure, pushing down the original E4 to position 3:



This kind of internal growth does occur with the choice on the Enable plan (to w.)

The Satellite Growth Point will cause (*tell-next-action E5*) to be loaded onto the stack, and the whole process will start again, now with E5 as the Nucleus. The output will eventually read something like:
 “We use.... Then we deliver.... Then we start...”

*****To hand in for 4.4:**

- Generate the above short story starting with E4. Email in a trace, showing some intermediate steps (say, the next goals from the growth points), and the final tree, and of course the final story.
- Generate the whole network story, starting with E1. Mail this in.
- Generate another story, starting somewhere else. This might be an even shorter story, and have a different order to sentences—perhaps only two sentences, linked by an appropriate cue phrase. Mail this in too.

EXTRA CREDIT: You will notice that a lot remains to be done—in particular, duplicated sentences, choosing pronouns (he/she), fixing tenses, deciding the placement of PPs and relative clauses, etc. If you feel like it, implement some microplanner rules to do the right things, and include it before the sentence realizer. Its input should be a frame of representation and its output should be one or more frames containing the same representation, augmented by details. This will be worth up to 10 points of extra credit.

This assignment has two main goals: to teach you about knowledge representation and how to write a planner and generator, including the grammar and lexicon, and to prompt you to *be creative!* The more you think about the questions, and the more you describe what you did and why you did it, the better.