Unit 4: Complex Processing

This unit is concerned with how we engage in a complex stream of processing. We will discuss how ACT-R can read sentences, extract their meaning, and later retrieve these meanings. We will also cover the "buffer stuffing" mechanism as it relates to the visual system.

4.1 Sentence Verification

Anderson (1974) reported an experiment where subjects read a story consisting of sentences in either the active or passive voice and then had to verify whether various test sentences where true given what they had read. If you open the **demo4** model and set ***actr-enabled-p*** to **nil**, you can get a sense of what this experiment was like by calling **do-experiment**. You will see 8 sentences presented for 5 seconds each, then see the prompt "test" for 2 seconds, and then you have to judge whether 8 sentences follow from what you studied. If the sentence is true press the "k" key with your right hand and if the sentence is false press the "d" key with your left hand. The following is the outcome of one run I made through the material:

```
? (do-experiment)
CORRELATION:  0.243
MEAN DEVIATION:  0.491
            Active-Active Active-Passive Passive-Active Passive-Passive
True:         2.99 (T  )     2.37 (T  )     2.62 (T  )     3.17 (T  )
False:        1.69 (T  )     3.09 (T  )     2.15 (T  )     2.92 (T  )
```

The response time and correctness are reported classified by whether the sentence is true or false (by default the first 4 test sentences are true), by the voice of the study sentence, and by the voice of the test sentence on which it is based, represented as "study type"-"test type". Also printed out is the correlation of my data with the average from Anderson (1974) as well as the mean deviation. That average data are

```
          Active-active Active-passive Passive-active Passive-passive
True:         2.25           2.80           2.30           2.75
False:        2.55           2.95           2.55           2.95
```

Basically, subjects are slower to judge passive sentences than active sentences, show an effect of the truth of the judgment, but are not affected by the voice of the original sentence they studied. These data are typically interpreted as implying that the subjects had converted the study sentences into some form that was the same whether they had been active or passive. Thus, there is no effect of voice of study sentence. At test, they similarly convert the test sentence into such a format and try to retrieve a matching sentence. They take longer to process passive test sentences because they consist of more words.

Thus, to model these data we need to model how subjects parse sentences into an internal memory representation. The **demo4** model can read sentences (active or passive) and parse them into an internal representation. However, the model provided **does not** parse passive sentences correctly. It parses every sentence as if it is active. You will have to extend it to properly encode passive sentences. It also cannot respond to the test sentences. For the assignment you will have to

organize the process that compares the test sentences to the study sentences stored in memory and presses a key to respond.

This experiment differs from the previous experiments where each trial was independent, and the model could be reset before each trial. For this task, the model needs to be able to read and remember multiple sentences, so it will be run continuously through the entire task without being reset for each new display. This presents a couple of new challenges for the model that will also be addressed in this unit – detecting screen changes and repeated motor actions.

4.2 Parsing Active Sentences

The first sentence in the study set is "The painter visited the missionary". That is an active sentence, and here is the trace produced by studying it. (There is a function provided that will present single study trials called **study-sentence**. It takes a string to present and a time to present it for. The global variable ***demo4-study-set*** is a list of the sentences for the experiment.)

```
? (study-sentence (first *demo4-study-set*) 5)

 Time  0.000: Vision found LOC1265
 Time  0.000: Found-New-Word Selected
 Time  0.050: Found-New-Word Fired
 Time  0.050: Module :MOTOR running command CLEAR
 Time  0.050: Module :VISION running command MOVE-ATTENTION
 Time  0.100: Module :MOTOR running command CHANGE-STATE
 Time  0.135: Module :VISION running command ENCODING-COMPLETE
 Time  0.135: Vision sees TEXT1260
 Time  0.135: Read-Word Selected
 Time  0.185: Read-Word Fired
 Time  0.185: Skip-The Selected
 Time  0.235: Skip-The Fired
 Time  0.235: Find-Next-Word Selected
 Time  0.285: Find-Next-Word Fired
 Time  0.285: Module :VISION running command FIND-LOCATION
 Time  0.285: Vision found LOC1267
 Time  0.285: Attend-Word Selected
 Time  0.335: Failure Retrieved
 Time  0.335: Attend-Word Fired
 Time  0.335: Module :VISION running command MOVE-ATTENTION
 Time  0.420: Module :VISION running command ENCODING-COMPLETE
 Time  0.420: Vision sees TEXT1261
 Time  0.420: Read-Word Selected
 Time  0.470: Read-Word Fired
 Time  0.620: Painter Retrieved
 Time  0.620: Process-First-Noun Selected
 Time  0.670: Process-First-Noun Fired
 Time  0.670: Find-Next-Word Selected
 Time  0.720: Find-Next-Word Fired
 Time  0.720: Module :VISION running command FIND-LOCATION
 Time  0.720: Vision found LOC1269
 Time  0.720: Attend-Word Selected
 Time  0.770: Attend-Word Fired
 Time  0.770: Module :VISION running command MOVE-ATTENTION
 Time  0.855: Module :VISION running command ENCODING-COMPLETE
 Time  0.855: Vision sees TEXT1262
 Time  0.855: Read-Word Selected
 Time  0.905: Read-Word Fired
```

```
 Time  1.055: Visit Retrieved
 Time  1.055: Process-Verb Selected
 Time  1.105: Process-Verb Fired
 Time  1.105: Find-Next-Word Selected
 Time  1.155: Find-Next-Word Fired
 Time  1.155: Module :VISION running command FIND-LOCATION
 Time  1.155: Vision found LOC1271
 Time  1.155: Attend-Word Selected
 Time  1.205: Attend-Word Fired
 Time  1.205: Module :VISION running command MOVE-ATTENTION
 Time  1.290: Module :VISION running command ENCODING-COMPLETE
 Time  1.290: Vision sees TEXT1263
 Time  1.290: Read-Word Selected
 Time  1.340: Read-Word Fired
 Time  1.340: Skip-The Selected
 Time  1.390: Skip-The Fired
 Time  1.390: Find-Next-Word Selected
 Time  1.440: Find-Next-Word Fired
 Time  1.440: Module :VISION running command FIND-LOCATION
 Time  1.440: Vision found LOC1273
 Time  1.440: Attend-Word Selected
 Time  1.490: Failure Retrieved
 Time  1.490: Attend-Word Fired
 Time  1.490: Module :VISION running command MOVE-ATTENTION
 Time  1.575: Module :VISION running command ENCODING-COMPLETE
 Time  1.575: Vision sees TEXT1264
 Time  1.575: Read-Word Selected
 Time  1.625: Read-Word Fired
 Time  1.775: Missionary Retrieved
 Time  1.775: Process-Last-Word-Object Selected
 Time  1.825: Process-Last-Word-Object Fired
 Time  1.825: Study-Sentence-Read-Wait Selected
 Time  1.875: Study-Sentence-Read-Wait Fired
 Time  1.875: Module :VISION running command CLEAR
 Time  1.925: Module :VISION running command CHANGE-STATE
 Time  5.000: * Running stopped because time limit reached.
```

The first production that fires is **found-new-word:**

```
(P found-new-word
    =goal>
       ISA         comprehend-sentence
       state       nil
    =visual-location>
       ISA         visual-location
    =visual-state>
       ISA         module-state
       modality    free
==>
   -manual>
   =goal>
       state       attending
    +visual>
       ISA         visual-object
       screen-pos  =visual-location
)
```

This production has a couple of interesting features. Why is there a visual-location in the buffer at the start of the trial, and what is the purpose of the **–manual** request?

4.2.1 Visual Buffer Stuffing

In the last unit buffer stuffing was introduced as a simple bottom-up attention mechanism for the aural system. It also applies to the visual system. When the **visual-location** buffer is empty and the model processes the display it might automatically place the location of one of the visual objects into the **visual-location** buffer. For the aural system, because there was only one sound it was automatically "buffer stuffed". However, because the visual system is almost always presented with multiple objects there is a mechanism that allows you to specify how the object whose location is selected gets determined. You can specify the conditions for what gets buffer stuffed using the same conditions you would use to specify a regular +**visual-location** request. Thus, when the screen is processed, if there is a visual-location that matches that specification and the **visual-location** buffer is empty, then that location will be stuffed into the **visual-location** buffer.

The default specification for a visual-location to stuff is attended new and screen-x lowest. If you go back and run the previous units' models you can see that before the first production fires to request a visual-location there is in fact already one in the buffer, and it is the leftmost new item on the screen. To change the conditions for selecting the visual-location to be buffer stuffed you need to call the command **pm-set-visloc-default**. It takes keywords that specify the slots to test and the value to test for as would be used in a production. Here are a couple of examples:

```
(pm-set-visloc-default :attended new :screen-x lowest)
(pm-set-visloc-default :screen-x current :screen-y (within 100 230))
(pm-set-visloc-default :kind text :color red)
```

By using buffer stuffing the model can detect screen changes. The alternative method would be to continually request a location that was attended new, notice that there was a failure to find one, and request again until one was found.

One thing to keep in mind is that buffer stuffing only occurs if the buffer is empty. So if you want to take advantage of it you must make sure that the **visual-location** buffer is cleared before the update on which you want a location to be stuffed.

4.2.2 Repeated Motor Actions

On the RHS of the **found-new-word** production there is a request to clear the manual buffer:

```
(P found-new-word
…
==>
   -manual>
 …
)
```

This is done to keep the modeling task for this unit simple. By default, the time it takes to perform a motor action is faster if it shares features with the previous motor action. The

following production presses a key repeatedly, and the trace shows the speedup between the first and second presses:

```
(p press-keys
     =goal>
     isa goal
     =manual-state>
     isa module-state
     modality free
     ==>
     +manual>
     isa press-key
     key "d")


> (pm-run 3.0)
 Time  0.000: Press-Keys Selected
 Time  0.050: Press-Keys Fired
 Time  0.050: Module :motor running command press-key
 Time  0.200: Module :motor running command preparation-complete
 Time  0.250: Module :motor running command initiation-complete
 Time  0.260: Device running command output-key
 Time  0.350: Module :motor running command finish-movement
 Time  0.350: Press-Keys Selected
 Time  0.400: Press-Keys Fired
 Time  0.400: Module :motor running command press-key
 Time  0.400: Module :motor running command preparation-complete
 Time  0.450: Module :motor running command initiation-complete
 Time  0.460: Device running command output-key
 Time  0.550: Module :motor running command finish-movement
```

The first key press takes .21 seconds – the time between **press-keys** firing and the output-key action. The second press however only takes .06 seconds because the .15 seconds of preparation time is not necessary because it is the same action that was previously prepared.

By clearing the manual buffer at the start of a trial we eliminate the previous actions features, and then do not benefit from the speed up. In this way the model will always take the same amount of time to respond to a test sentence, no matter which answer it had given previously. A better way to handle the situation would be to allow the speed up and run the model many times over the task with the presentation order randomized and then average the results, but that makes it more difficult for the assignment, so for simplicity this model does not speed up for repeated responses.

4.2.3 More about Found-New-Word

Back to the first production that fires - **found-new-word:**

```
(P found-new-word
    =goal>
       ISA         comprehend-sentence
       state       nil
    =visual-location>
       ISA         visual-location
    =visual-state>
       ISA         module-state
       modality    free
```

```
==>
   -manual>
   =goal>
       state       attending
    +visual>
       ISA         visual-object
       screen-pos  =visual-location
)
```

Why does it use buffer stuffing to get the first location on the screen instead of just requesting one as was done with the first productions from the previous units? The difference is that this model runs continuously over multiple sentences whereas the previous units were run on each trial independently. Previously the models were reset to initial conditions before each trial, so the first production to fire was always at the start of the trial. However, with this task the model must read multiple sentences without being reset between trials and thus it needs to be able to detect when a new sentence is presented so that it can start processing it. Buffer stuffing provides the model with the ability to detect the new sentence and start processing it.

4.2.4 Reading the Sentence

Throughout the processing of this sentence, we see a triple of productions apply to process the next word in the sequence (with the exception of the first word which only needs 2 productions – **found-new-word** and **read-word** because it can take advantage of buffer stuffing). This is the same sort of triple as we have seen apply in past exercises:

```
(P find-next-word
    =goal>
       ISA         comprehend-sentence
       state       find
==>
    +visual-location>
       ISA         visual-location
       screen-x    greater-than-current
       nearest     current
    =goal>
       state       looking
)



(P attend-word
    =goal>
       ISA         comprehend-sentence
       state       looking
    =visual-location>
       ISA         visual-location
    =visual-state>
       ISA         module-state
       modality    free
==>
    =goal>
       state       attending
    +visual>
       ISA         visual-object
       screen-pos  =visual-location
)
```

```
(P read-word
    =goal>
       ISA         comprehend-sentence
       state       attending
       word        nil
    =visual>
       ISA         text
       value       =word
==>
    -visual-location>
    =goal>
       word        =word
       state       read
    +retrieval>
       ISA         meaning
       word        =word
)
```

The first production requests the location of the next word to the right of the currently attended word. The second switches attention to the location of that word. The third identifies the word and makes a request to retrieve the meaning of the word. Content words like "missionary" and "chase" have meanings but functor words like "the" do not. While a more complex model might do more we simply skip the word “the”:

```
(P skip-the
    =goal>
       ISA         comprehend-sentence
       word        "the"
       state       read
==>
    =goal>
       state       find
       word        nil
   )
```

which sets the word slot to nil and sets the state to find so that it will look for the next word. There are also similar productions that skip the words “was” and “by” so that the model can read the passive sentences even though it does not properly encode them. You will have to replace or change one or both of the skip-was and skip-by productions to enable your model to properly encode passive sentences.

With respect to the content words, we need to extract their meaning and build up a representation of the sentence's meaning. One way to create new internal chunks in ACT-R is by means of past goal chunks. The goal chunk to comprehend the sentence will serve as the repository of the sentence's meaning. The chunk-structure for the goal is:

```
(chunk-type comprehend-sentence agent action object purpose word state)
```

The **agent**, **action**, and **object** slots will hold the meanings of the content words. The **purpose** slot keeps track of whether we are processing the sentence for purpose of study or test. It will help us take a different path at the end of test sentences and not confuse study sentences with test

sentences when we try to recall the studied material. The **word** slot holds the current word being processed, and **state** keeps track of the current state.

At the end of the run shown above there will be two chunks of type comprehend-sentence. The one in the goal buffer looks like this:

```
**Goal1   0.000
    isa COMPREHEND-SENTENCE
    agent nil
    action nil
    object nil
    purpose Study
    word nil
    state nil
```

It is a new goal constructed by the **study-sentence-read-wait** production which prepares the model for the next trial. The previous goal now encodes the sentence that was processed:

```
Goal    0.000
    isa COMPREHEND-SENTENCE
    agent Painter
    action Visit
    object Missionary
    purpose Study
    word "missionary"
    state Sentence-Complete
```

An active sentence reflects the default pattern in English which is that the first word is an agent, the second the action, and the third the object. The productions in the **demo4** model reflect this default assumption. For instance, the production that processes the second content word is:

```
(P process-verb
    =goal>
       ISA         comprehend-sentence
       agent       =val
       action      nil
       word        =word
       state       read
    =retrieval>
       ISA         meaning
       word        =word
==>
    =goal>
       action      =retrieval
       word        nil
       state       find
)
```

This responds to the fact that a word meaning has already been assigned to the **agent** slot and no word has been assigned to the **action** slot. It then takes the retrieved meaning of word **=word** and assigns it to the action slot. As in the production **skip-the**, this production also sets the word slot to nil and the state to find to enable the processing of the next word.

The last production that fires above, **study-sentence-read-wait**, creates a new goal in preparation for the next study sentence.

```
(P study-sentence-read-wait
    =goal>
       ISA         comprehend-sentence
       state       sentence-complete
       purpose     study
==>
    -visual>
    +goal>
       ISA         comprehend-sentence
       purpose     study
)
```

It also requests the clearing of the **visual** buffer to suppress the re-encoding discussed in unit 2.

After the model reads the word “test” the following production will apply

```
(P respond-to-test
    =goal>
       ISA         comprehend-sentence
       word        "test"
    =retrieval>
       ISA         meaning
       word        "test"
==>
   -visual>
   +goal>
       ISA         comprehend-sentence
       purpose     test
)
```

which creates a new goal whose **purpose** is **test** in preparation for the first test sentence. When extending the model so that it can respond to the test sentences you will probably want your last production that fires before the next presentation to also create a new goal with the **purpose** slot set to **test**.

4.3 Extending the Assignment

You should probably first try to extend this program to parse passive sentences correctly. When the first noun comes in you cannot tell whether the sentence is active or passive. Therefore, your program will probably still assign that to the agent slot. When the passive structure is signaled by the "was" and "by" you will need to switch what slot the meaning of the first noun is assigned to. You can test your passive parser by calling **study-sentence** with some passive sentence like the third sentence in study-set:

```
(study-sentence (third *demo4-study-set*) 5)
```

There is also a function provided that presents the sentences for testing called **test-sentence**. It requires one of the test specifications from the global variable ***demo4-test-set*** as a parameter:

```
> (test-sentence (first *demo4-test-set*))
(1 30.0 nil)
```

It returns a list of three items. The first is the position of the test probe in the global list, the second is the response time to the question in seconds, and the third is t if the response was correct and nil if it was not.

It is recommended that you test your model with these functions on individual sentences until your model is capable of doing the whole task. Remember that the model needs to remember the sentences from study to test and thus is not reset for each trail. Of the provided functions, only **do-experiment** resets the model, so you may need to reset your model manually while testing on individual sentences if you want it to "forget" what it has seen or start over.

The productions that you started with will parse active sentences whether they are presented as study sentences or test sentences and your extension to passives should do the same. However, in the case of test sentences you need to go on and verify the sentence after parsing it. Remember that the comprehension goal has a slot called purpose that will be set to study in the case of study sentences and to test in the case of test sentences.

It is not really possible, given what we have discussed about ACT-R, to account for why subjects are slower on the false sentences. This would require going down to the subsymbolic level of declarative retrieval. However, it is possible to make the production that responds negative take longer by setting an effort longer than the default .05 seconds for a production to fire. This is done using the spp command to change the effort parameter of the production that responds no. This is the setting I used in my solution:

```
(spp answer-no :effort .2)
```

That sets the time taken by the answer-no production to be .2 seconds instead of .05 seconds.

As a reference point, the following is the performance of my model on this task:

```
CORRELATION:  0.972
MEAN DEVIATION:  0.077
          Active-Active Active-Passive Passive-Active Passive-Passive
True:      2.29 (T  )     2.85 (T  )     2.29 (T  )     2.85 (T  )
False:     2.44 (T  )     3.00 (T  )     2.44 (T  )     3.00 (T  )
```