Unit 8: Advanced Production Techniques

In this unit we will describe two additional mechanisms which can be used in the procedural system of ACT-R. They are called procedural partial matching and dynamic pattern matching. These capabilities allow for a lot more flexibility in the procedural system, and they can make it easier to create models which are able to work in situations where all of the details of the task are not know in advance and thus cannot be explicitly encoded within the model.

8.1 Procedural partial matching

Procedural partial matching is very similar to the partial matching of declarative memory as was described in unit 5 of the tutorial. When procedural partial matching is enabled by setting the **:ppm** parameter to a number, a production may be selected to fire even when its conditions do not perfectly match the current buffer contents. This mechanism applies to all productions being tested and it modifies the conflict resolution process used when multiple productions match under this looser definition of matching. Other than that however it does not change any of the other operations of the procedural system and all other procedural functionality is as described in the previous units.

8.1.1 Condition testing with procedural partial matching

When procedural partial matching is enabled the slot tests for the buffer chunks' values are slightly relaxed, but most of the conditions in a production are still tested explicitly. The following conditions must still be true for the production to match: if the production tests a buffer then that buffer must contain a chunk, all queries must be true as specified, all inequality tests on slots (negations, less-than, and greater-than comparisons) must be true, and all special conditions specified with !eval! or !bind! operators must be true. The only tests which do not have to be true are equality tests for the slots of the chunk in a buffer i.e. tests that specify a specific value for a slot or tests with variables which are comparing slot values. If all of the equality tests are true, then the production matches just as it would without procedural partial matching enabled. If some of the tests are not true the production may still be considered a match under procedural partial matching.

If a value specified for a slot in the production does not match the current value of that slot for the chunk in the buffer, then in order for it to be considered a match under procedural partial matching the mismatching values must be similar. Similarity here is defined in the same way that it is for declarative memory. Thus they must be chunks for which a similarity value has been specified through the set-similarities command or the modeler must provide a similarity hook function which specifies a similarity between the items whether or not they are chunks as was done for numbers in the unit 5 onehit blackjack game. The only difference between the procedural partial matching and declarative partial matching is that for procedural partial matching, items are only considered to be similar if they have a similarity value which is greater than the current maximum similarity difference (as set with the :md parameter). That additional constraint on the similarities is done for practical reasons because by default, all chunks have a similarity of the maximum difference to all other chunks. Without that constraint, any chunk could potentially be a partial match to any other in the production matching. By disallowing procedural partial matching from using the default similarity difference value it allows the modeler to control which items are allowed to be partially matched in productions. If one wants all chunks to potentially match to any

other chunk, then a simple similarity hook function can be defined that returns a value greater than :md for all chunk to chunk similarities.

8.1.2 Conflict resolution with procedural partial matching

If only one production matches then it is selected and fired. When there is more than one production which matches (including partially matched productions) the production with the highest utility is the one selected and fired as has been described before. The difference when procedural partial matching is enabled is that productions which are not a perfect match receive a penalty to their utility. The equation for the utility of production *i* when procedural partial matching is enabled is:

$$Utility_i(t) = U_i(t) + \varepsilon + \sum_j ppm * similarity(d_j, v_j)$$

Ui(t): Production *i*'s current true utility value.

ε: The noise which may be added to the utility.

j: The set of slots for which production *i* had a partially matched value.

ppm: The value of the :ppm parameter.

d_j**:** The desired value for slot *j* in production *i*.

v_j**:** The actual value in slot *j* of the chunk in the buffer.

A production which has one or more partially matched slot tests has its utility decreased by the similarity of each mismatch (since similarity values are negative) multiplied by the :ppm setting. That adjusted utility value is only used for the purposes of conflict resolution. It does not affect the $U_i(t)$ value used in the utility learning equation or the utility values used during production compilation learning.

8.1.3 Simple procedural partial matching model

The simple-ppm-model included with the tutorial unit shows a very simple example of procedural partial matching. It does not have a corresponding experiment and everything in it except the :ppm and :cst parameters has been described in previous units, so it should be easy to understand. The **goal** buffer starts with a chunk which looks like this:

```
GOAL: GOAL-CHUNK0 [TEST0]
GOAL-CHUNK0
   VALUE  SMALL
```

and there are three productions which simply test the value slot of the chunk in the **goal** buffer and then output the value from that slot:

```
(p small                (p medium               (p large
   =goal>                  =goal>                  =goal>
```

```
      isa test                isa test                isa test
      value small             value medium            value large
      value =val              value =val              value =val
    ==>                     ==>                     ==>
    !output! =val           !output! =val           !output! =val
    -goal>)                 -goal>)                 -goal>)
```

The starting model has procedural partial matching turned off. If you load that model and run it you will see a trace like this:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
(P SMALL
   =GOAL>
       VALUE SMALL
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production SMALL:
 :UTILITY  0.000
 :U  0.000
     0.000   PROCEDURAL             PRODUCTION-SELECTED SMALL
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED SMALL
SMALL
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   ------                 Stopped because no events left to process
```

As expected the production small is selected and fired. Before describing the results with procedural partial matching enabled the additional output in the trace will be described.

8.1.3.1 The conflict set trace parameter

The :cst parameter controls the conflict set trace, and it is set to **t** in the model to enable that trace. It is a model debugging aid similar to the activation trace parameter used with the declarative memory module. If it is set to **t** then during every conflict resolution event it will display the instantiation of each production which matches along with its current utility parameters. That set of matching productions is referred to as the conflict set. In the trace above that only shows the production small, and because utility learning is not turned on and there is no utility noise value set in the model that production has both a utility and U(t) of 0.

8.1.3.2 Turning on ppm

If you change the model to enable procedural partial matching by setting the :ppm parameter to 1 (you can make that change in the file and reload it or reset the model and then make the change interactively) then run it you will see a trace like this:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
(P SMALL
   =GOAL>
       VALUE SMALL
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
```

```
   -GOAL>
)
Parameters for production SMALL:
 :UTILITY  0.000
 :U  0.000
(P MEDIUM
   =GOAL>
       VALUE [MEDIUM, SMALL, -0.5]
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production MEDIUM:
 :UTILITY -0.500
 :U  0.000
     0.000   PROCEDURAL             PRODUCTION-SELECTED SMALL
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED SMALL
SMALL
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   ------                 Stopped because no events left to process
```

Again we see the production small being selected and fired, but now we see that the production medium was also a member of the conflict set. For a partially matched production the instantiation of the production will show additional information about the partial match. Here is the instantiation for the production medium from the trace:

```
(P MEDIUM
   =GOAL>
       VALUE [MEDIUM, SMALL, -0.5]
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
```

In the first comparison for the value slot, instead of just seeing medium as was specified in the production, we see a set of items in brackets. That indicates that the test was not a perfect one. The first item in the brackets was the value specified in the production. The second item is the buffer chunk's value for that slot, and the third item is the similarity between those items.

Another important thing to notice is that the binding for the variable =val in that production is still small (the other value slot test and the !output! which are specified with =val in the production). That is because that is the actual content of that slot in the chunk in the buffer – the variable bindings come from the slots of the buffer's chunk and not values specified in the production.

After the instantiation of medium we see its current utility values:

```
Parameters for production MEDIUM:
 :UTILITY -0.500
 :U  0.000
```

The U(t) value for medium is 0 just as it is for small. The utility used for deciding which production to fire however is not 0 because it was not a perfect match. The similarity between small and medium is set to -0.5 in the model and the :ppm value was set to 1. So, the utility of medium is: 0 + 1 * (-0.5) = -0.5.

8.1.3.3 The large production

Why isn't the production large also considered in the conflict set as a partial match? The reason is because there is no similarity set between the chunks small and large in the model. Thus, those chunks have the maximum similarity difference value and a test for the chunk large will not be considered a partial match to the chunk small in a production.

8.1.3.4 Adding noise

If the :egs parameter is set to a value greater than 0 then occasionally the medium production will be selected over the small production because of the noise added to the utilities. With the :egs value set to 1 we should see medium chosen almost 40% of the time with the difference of .5 between their utilities. Here is a run where medium was selected:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
(P SMALL
   =GOAL>
       VALUE SMALL
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production SMALL:
 :UTILITY -1.534
 :U  0.000
(P MEDIUM
   =GOAL>
       VALUE [MEDIUM, SMALL, -0.5]
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production MEDIUM:
 :UTILITY  3.296
 :U  0.000
     0.000   PROCEDURAL             PRODUCTION-SELECTED MEDIUM
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED MEDIUM
SMALL
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   ------                 Stopped because no events left to process
```

8.1.4 Building Sticks Task alternate model

To see procedural partial matching used in an actual task this unit contains a slightly different model of the Building Sticks Task from unit 6 of the tutorial. The main difference between this model and the one

presented in unit 6 is that instead of having separate productions to force and decide for both the over and under strategies, this model only has a decide production for each strategy.

There are several minor differences between this model and the previous version with respect to how the encoding is performed to enable the simpler test, but we will only be looking in detail at the two major differences between them. Those differences are in how the strategy is initially chosen and how the model computes the difference between the current stick's length and the goal stick's length.

The model is in the bst-ppm-model.lisp file with the unit 8 materials, and the corresponding experiment code is found in the bst-ppm.lisp and bst_ppm.py files (which will load the model automatically). The model can be run through the experiment using the bst-ppm-experiment function in Lisp and the experiment function from the bst_ppm module in Python.

8.1.4.1 Strategy choice

In this model these are the productions which decide between the overshoot and undershoot strategies:

```
(p decide-over                          (p decide-under
    =goal>                                  =goal>
      isa         try-strategy                isa         try-strategy
      state       choose-strategy             state       choose-strategy
     - strategy   over                       -  strategy   under
    =imaginal>                              =imaginal>
      isa         encoding                    isa         encoding
      goal-length =d                          goal-length =d
      b-len       =d                          c-len       =d
   ==>                                     ==>
    =imaginal>                              =imaginal>
    =goal>                                  =goal>
      state    prepare-mouse                 state    prepare-mouse
      strategy over                          strategy under
    +visual-location>                       +visual-location>
      isa      visual-location               isa      visual-location
      kind     oval                          kind     oval
      value    "b")                          value    "c")
```

They are essentially a combination of the force and decide productions from the previous model for each strategy. Because they test the current strategy value they still operate as a forcing production when the model fails with its first choice and has to choose the other strategy, but when there is no current strategy either one can match and the utilities will determine which one is selected.

Procedural partial matching is enabled in this model and a similarity between numbers is provided using the sim-hook as it was for the 1hit blackjack model using these parameter settings:

```
(sgp :ppm 40 :sim-hook "bst-number-sims")
```

The :ppm value of 40 was estimated to produce a good fit to the data. The similarities between numbers (the line lengths in pixels) are computed by the bst-number-sims command which is provided in the experiment file and are set using a simple linear function to scale the possible differences into the default similarity range of 0 to -1:

$$Similarity(a,b) = -\frac{abs(a-b)}{300}$$

Unlike the previous model of this task, this one does not need to compute the difference between the goal stick's length and the b and c sticks' lengths explicitly to determine whether overshoot or undershoot should be used. Now that happens as a result of the partial matching in the **imaginal** buffer test of those productions. Whichever stick is closer to the goal stick's length will bias the utility toward that choice.

The initial utilities of the decide-over and decide-under productions are set at 10 in the model. Given that, we can look at how utility is determined when those productions are competing on the first problem which has stick lengths of a=15, b=250, c=55, and a goal of 125. The chunk in the **imaginal** buffer at that time will look like this:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   A-LOC  VISUAL-LOCATION8-0
   B-LOC  VISUAL-LOCATION9-0
   C-LOC  VISUAL-LOCATION10-0
   GOAL-LOC  VISUAL-LOCATION11-0
   GOAL-LENGTH  125
   B-LEN  250
   C-LEN  55
```

The length of the goal, b, and c sticks are the important tests for the decide productions in the **imaginal** buffer tests. Each production is checking to see if the desired stick and the goal stick are the same length, and the procedural partial matching will adjust the utilities of those productions based on the similarity between the compared sticks. Here are the instantiations of those productions in the conflict set trace showing the similarities as computed from the similarity function for the mismatching and the resulting utilities (with the utility noise turned off for this example):

```
     1.005   PROCEDURAL             CONFLICT-RESOLUTION
(P DECIDE-OVER
   =GOAL>
       STATE CHOOSE-STRATEGY
    -  STRATEGY OVER
   =IMAGINAL>
       GOAL-LENGTH 125
       B-LEN [125, 250, -0.41666666]
 ==>
   =IMAGINAL>
   =GOAL>
       STATE PREPARE-MOUSE
       STRATEGY OVER
   +VISUAL-LOCATION>
       KIND OVAL
       VALUE "b"
)
Parameters for production DECIDE-OVER:
 :UTILITY -6.667
 :U 10.000
(P DECIDE-UNDER
   =GOAL>
       STATE CHOOSE-STRATEGY
```

```
    -  STRATEGY UNDER
   =IMAGINAL>
       GOAL-LENGTH 125
       C-LEN [125, 55, -0.23333333]
 ==>
   =IMAGINAL>
   =GOAL>
       STATE PREPARE-MOUSE
       STRATEGY UNDER
   +VISUAL-LOCATION>
       KIND OVAL
       VALUE "c"
)
Parameters for production DECIDE-UNDER:
 :UTILITY  0.667
 :U 10.000
```

The effective utility of decide-over is -6.667 (10 + 40 * -0.41666666) and the effective utility of decide-under is 0.667 (10 + 40 * -0.23333333). If the noise for utilities is set at 3 in the model we can compute the probabilities of choosing over and under on the initial trial using the equation presented in unit 6:

$$\Pr obability(i) = \frac{e^{U_i / \sqrt{2}s}}{\sum_j e^{U_j / \sqrt{2}s}}$$

$$\Pr obability(over) = \frac{e^{-6.666/4.24}}{e^{-6.666/4.24} + e^{.666/4.24}}$$

$$\approx .15$$

$$\Pr obability(under) = \frac{e^{0.666/4.24}}{e^{-6.666/4.24} + e^{.666/4.24}}$$

$$\approx .85$$

Thus, the model will pick undershoot about 85% of the time for the first problem.

As in the unit 6 model of the task, this model is also learning utilities based on the rewards it gets. So, the base utility values of the strategies will be adjusted as it progresses. Here are the results from running the unit 6 model of the task showing the average learned utility values for the four productions it needs to make the decision:

```
CORRELATION:  0.772
MEAN DEVIATION: 18.155
```

```
Trial 1     2     3     4     5     6     7     8     9     10    11    12    13    14    15
    22.0  50.0  61.0  77.0  94.0  32.0  85.0  83.0  61.0  42.0  30.0  21.0  63.0  63.0  54.0

DECIDE-OVER : 13.2118
DECIDE-UNDER: 10.8572
FORCE-OVER  : 12.0410
FORCE-UNDER : 6.6625
```

Here are the results of running this model of the task:

```
CORRELATION:  0.876
MEAN DEVIATION: 13.619

Trial 1     2     3     4     5     6     7     8     9     10    11    12    13    14    15
    11.0  62.0  45.0  72.0  87.0  28.0  75.0  88.0  66.0  30.0  31.0  18.0  62.0  67.0  36.0

DECIDE-OVER : 11.8611
DECIDE-UNDER: 7.2320
```

We see a similar shift in the utilities with over gaining utility and under losing utility and overall this model produces a slightly better fit to the data.

8.1.4.2 Attributing actions to the imaginal module

Before moving on to the other new procedural mechanism for this unit there is one other new capability to look at in this Building Sticks model. In the previous model of the task the productions computed the difference between the current stick length and the goal stick length to decide whether to select stick a or stick c for each of the actions after the strategy choice had been made using a !bind! action in the production to do the math and then place that value into a slot of the **imaginal** buffer:

```
(p calculate-difference
   =goal>
      isa      try-strategy
      state    calculate-difference
   =imaginal>
      isa      encoding
      length   =current-len
   =visual>
      isa      line
      width    =goal-len
  ==>
   !bind! =val (abs (- =current-len =goal-len))
   =imaginal>
      length   =val
   =goal>
      state    consider-next)
```

In many models abstractions like that are easy to justify and do not cause any issues for data comparison. However, if one is interested in more low-level details, often when comparing model data to human data collected from fMRI or EEG studies, then accounting for the timing of such actions and attributing them to an appropriate module can be important. In the case of the imaginal module, there is a built in mechanism that allows the modeler to assign user defined functionality to the imaginal module and this version of the Building Sticks task does so instead of using a !bind! action for computing the difference. The details of how that is done are in the code text for this unit.

8.2 Dynamic Pattern Matching

Dynamic pattern matching is a powerful mechanism which allows the model to test and extend its representations based on the current context without the specific information having to be explicitly encoded into the model. It is particularly important for tasks where instruction or example following are involved, but can also be useful in other situations where flexibility or context dependence are necessary.

8.2.1 Basic Operation

The distinguishing feature between dynamic pattern matching and the pattern matching which has been used so far in the tutorial is that with dynamic pattern matching one can use variables to specify the slots in the conditions and actions of the productions. In fact, dynamic pattern matching is really part of the normal pattern matching process for productions and does not require any changes to enable it. To demonstrate dynamic pattern matching we will be using a very simple model found in the "simple-dynamic-model.lisp" file. It has three productions which demonstrate the typical dynamic uses, and it is a very simplified version of something that is often done in a model that is following a set of declarative instructions and updating a chunk in the **imaginal** buffer based on information it retrieves from declarative memory. Because all of the components of this model are known in advance it is not actually necessary to use dynamic pattern matching, but that should make it easier to understand since it can be compared to the static productions which would perform the same operations.

8.2.2 Arbitrary slots in conditions

The first thing that dynamic pattern matching allows is the ability to test arbitrary slots in the conditions of a production. The conditions of the productions start and retrieve-first-step from the example model show this:

```
(p start
   =goal>
     isa      fact
     context  =context
     =context =x
   ?imaginal>
     state    free
     buffer   empty
  ==>
   =goal>
     context  destination
   +imaginal>
     isa      result
     data1    =x
   +retrieval>
     isa      step
     step     first)
```

```
(p retrieve-first-step
   =goal>
     isa     fact
     context =slot
     data    =x
   =imaginal>
     isa     result
     data1   =x
   =retrieval>
     isa     step
     =slot   =target
     step    first
  ==>
   =goal>
     data    11
   =imaginal>
     =target =x
   +retrieval>
     isa     step
     step    second)
```

The start production uses a dynamic test in the **goal** buffer pattern and the retrieve-first-step production uses a dynamic test in the **retrieval** buffer pattern. By using a variable to name a slot in a condition, that test now depends on the contents of a buffer at the time of the test instead of specifying the slot name in advance. That means that the same production could test different slots based on the current context.

First we will look at the production start. The only thing different in that production relative to those seen previously is the last test in the **goal** buffer's condition:

```
     =goal>
       isa      fact
       context  =context
       =context =x
```

That means that the slot being tested will be the one which corresponds to the binding of the variable =context. The variable =x will be bound to the value of that slot. The =context variable is bound to the value of the context slot of the chunk in the **goal** buffer in the same way that you have seen in other productions.

Before running the model we will look at the contents of the **goal** buffer and see how this will apply for pattern matching. [If you want, you could run the model using the Stepper tool in tutor mode as was used in unit 1 and try to instantiate the production yourself before continuing.]

Here is the initial chunk placed in the **goal** buffer in the model:

```
GOAL: GOAL-CHUNK0 [FACT0]
GOAL-CHUNK0
   CONTEXT  DATA
   DATA  10
```

The **goal** buffer pattern in the start production binds the =context variable to the value in the context slot, which is data. Then the slot which corresponds to the binding of =context, which is data, is used to bind the variable =x. Thus, =x gets bound to 10 since that is the value of the data slot of the chunk in the **goal** buffer. If the binding of =context did not correspond to a slot of the chunk in the buffer then the production would not match. The rest of the start production operates just like all of the other productions that have been seen previously in the tutorial and thus should not need further explanation.

When we run the model with the :cst parameter on to show the instantiation of matching productions this is what the start production's instantiation looks like:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
(P START
   =GOAL>
       CONTEXT DATA
       DATA 10
   ?IMAGINAL>
       STATE FREE
       BUFFER EMPTY
 ==>
   =GOAL>
       CONTEXT DESTINATION
   +IMAGINAL>
       DATA1 10
   +RETRIEVAL>
       STEP FIRST
)
Parameters for production START:
 :UTILITY  0.000
 :U  0.000
```

That instantiation looks just like productions that you have seen previously since the variables have been replaced with their values, but the important thing to remember is that the instantiation of a dynamic production may have instantiated variables in both the slot value and slot name positions.

The retrieve-first-step production also has a dynamic test in its condition:

```
  =goal>
   isa     fact
   context =slot
   data    =x
  =imaginal>
   isa     result
   data1   =x
  =retrieval>
   isa     step
   =slot   =target
   step    first
```

In this case, that test involves multiple buffers. The value for the =slot variable is bound from one buffer and used to test a slot in a different buffer. Again, you may want to step through that in tutor mode and instantiate it yourself to get a better feel for how this works before looking at the instantiation below.

After the start production fires, and the imaginal module has created the new chunk these are the chunks in the buffers at the time retrieve-first-step matches:

```
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   CONTEXT  DESTINATION
   DATA  10

IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   DATA1  10

RETRIEVAL: RETRIEVAL-CHUNK0 [A]
RETRIEVAL-CHUNK0
   STEP  FIRST
   DESTINATION  DATA2
```

Thus, in the **goal** buffer matching of retrieve-first-step the =slot variable will be bound to the value destination and the =x variable will be bound to 10. That **imaginal** buffer pattern matches since =x is bound to 10. Since the =slot value is bound to destination, the **retrieval** buffer pattern will bind the =target variable to the value in the destination slot, which is data2.

Here is the instantiation of that production from the trace:

```
(P RETRIEVE-FIRST-STEP
   =GOAL>
       CONTEXT DESTINATION
       DATA 10
   =IMAGINAL>
       DATA1 10
   =RETRIEVAL>
       DESTINATION DATA2
       STEP FIRST
 ==>
   =GOAL>
       DATA 11
   =IMAGINAL>
       DATA2 10
   +RETRIEVAL>
       STEP SECOND
)
```

Though not shown in these examples, the dynamic conditions may also be used with any of the modifiers for a slot test as well as multiple times within or across buffer conditions. Thus, this set of conditions would be valid in a production:

```
(P EXAMPLE
 =GOAL>
  CONTEXT =CONTEXT
 =IMAGINAL>
  > =MIN-S 0
  =MIN-S =MIN
  > =MAX-S =MIN
  =CONTEXT CURRENT
 =RETRIEVAL>
  - =CONTEXT COMPLETE
  MAX-SLOT =MAX-S
  MIN-SLOT =MIN-S
...)
```

However, there are some constraints imposed on the use of dynamic conditions and those will be discussed in a later section.

8.2.3 Arbitrary slots in actions

The actions of the retrieve-first-step production show how dynamic pattern matching can be used to specify an action based on the current context. In its modification of the **imaginal** buffer we see this:

```
 =goal>
   data    11
 =imaginal>
   =target =x
 +retrieval>
   isa     step
   step    second
```

As with the condition, that means that the slot of the chunk in the **imaginal** buffer that will be modified will be the one bound to the variable =target. From the previous section we saw that that was the value data2, and that the =x variable was bound to 10. Thus this action will modify the chunk in the **imaginal** buffer so that its data2 slot has the value 10.

Here is what the chunk in the **imaginal** buffer looks like after retrieve-first-step fires.

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   DATA1  10
   DATA2  10
```

A variable may be used to specify a slot in any modification, modification request, or request action of a production. However, because such values are only determined when the production actually matches, it is possible that the production may attempt to make a modification or request with a slot that was not previously defined for the indicated chunk-type (or any chunk-type if the production does not declare a chunk-type in the action). For the buffer modification actions there is a special mechanism which handles that which is described in the next section. For requests, if a previously unspecified slot is used that will usually lead to a warning and the failure to execute the request, but could also result in errors occurring during the run depending on the details of the module which handles the request. Thus, if slots of

requests are specified dynamically, care should be taken to ensure the productions perform other tests, either in that production, or elsewhere in the sequence of productions, to protect against invalid requests.

8.2.4 Extending chunks with new slots

The final production in the test model, retrieve-second-step, shows the final new capability which dynamic pattern matching allows. Here is the production:

```
(p retrieve-second-step
   =goal>
     isa     fact
     context =slot
     data    =x
   =imaginal>
   =retrieval>
     isa     step
     =slot   =target
     step    second
  ==>
   =imaginal>
     =target =x)
```

The condition of this production is very similar to the retrieve-first-step production, and the action of this production is a modification to the chunk in the **imaginal** buffer which looks exactly the same as the one from the retrieve-first-step production.

The significant difference between retrieve-first-step and retrieve-second-step is not in the specifications of the productions, but the contents of the buffers when they are selected. Retrieve-first-step was described in the previous section, and essentially the same matching process holds true for retrieve-second-step. However, the bindings for the variables are now different. Here are the chunks from the **goal** and **retrieval** buffers after retrieve-first-step fires which match the retrieve-second-step production:

```
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   CONTEXT  DESTINATION
   DATA  11

RETRIEVAL: RETRIEVAL-CHUNK0 [B]
RETRIEVAL-CHUNK0
   STEP  SECOND
   DESTINATION  DATA3
```

The variable bindings from the conditions of the retrieve-second-step production are: =slot bound to destination, =x bound to 11, and =target bound to data3. Given those bindings, the instantiation of the action of this production will look like this:

```
   =imaginal>
      data3 11
```

While that doesn't look all that different from the modification performed by the previous production the interesting thing to note is that none of the chunk-types specified for this model have a slot named data3:

```
(chunk-type fact context data)
```

```
(chunk-type step step destination)
(chunk-type result data1 data2)
```

As we have seen previously in the tutorial, modifications to a buffer add slots when the chunk does not have the slot indicated. When a dynamically determined modification action specifies a slot which does not exist in any of the chunk-types for the model that will be indicated in the trace by an extend-buffer-chunk action before the mod-buffer-chunk action:

```
0.350   PROCEDURAL              PRODUCTION-FIRED RETRIEVE-SECOND-STEP
0.350   PROCEDURAL              EXTEND-BUFFER-CHUNK IMAGINAL
0.350   PROCEDURAL              MOD-BUFFER-CHUNK IMAGINAL
```

Here is the result for the chunk in the **imaginal** buffer after that occurs:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   DATA1  10
   DATA2  10
   DATA3  11
```

This mechanism allows the model to build its chunk representations as needed instead of requiring all possible slots be specified up front.

It is also possible to extend the available slots without dynamic pattern matching in a modification action, but such a change will happen as part of the model definition instead of at run time and will also generate a warning. For example, if we were to add this production to this model:

```
(p change-new-slot
   =goal>
  ==>
   =goal>
    new-slot 10)
```

That will result in this warning when the model is loaded:

```
#|Warning: Production CHANGE-NEW-SLOT uses previously undefined slot NEW-SLOT. |#
```

Which indicates that a slot which was not specified in the chunk-type definitions is used. That will add a slot named new-slot to the chunk in the **goal** buffer if it does not already have one when the production fires, but it does not report the extend-buffer-chunk action since the extension of the chunk-types occurred at model loading time. Such a practice is not recommended and all slots specified explicitly in productions should be declared in chunk-type definitions.

8.2.5 Constraints on dynamic pattern matching

Dynamic pattern matching adds a lot of flexibility to the specification of productions, and this flexibility allows for more powerful productions to be created. However, since ACT-R is intended to be a model of human cognition, there are two constraints imposed upon how dynamic pattern matching works to maintain the plausibility of the system.

8.2.5.1 No Search

The first constraint on dynamic pattern matching is that it does not require searching to find a match. All productions which use dynamic pattern matching must be specified so that all variables are bound directly based on the contents of slots. The procedural module will not allow the creation of a production which has a pattern like this:

```
(p not-allowed
  =goal>
    isa example
    =some-slot desired-value
 ==>
)
```

which would require finding a slot based on its value. If search like that were allowed then the matching of a single production would be able to solve NP-hard problems which is not a plausible mechanism.

8.2.5.2 One level of indirection

The other constraint on dynamic pattern matching is that it only allows one level of indirection. All of the variables which are used as slot indicators must be bound to the values of slots which are specified as constants. Thus, one cannot use a dynamically matched slot's value as a slot indicator like this:

```
(p also-not-allowed
  =goal>
    isa example
    first-slot =s1
    =s1        =next-slot
    =next-slot value
 ==>
 )
```

Allowing an unbounded level of indirection is not plausible, thus some constraint needed to be determined for the indirection. A single level was chosen as the constraint because that was all that was necessary to provide the abstractions needed and such a mechanism appears to be realizable within the basal ganglia of the human brain as shown by Stocco, Lebiere, and Anderson.

8.2.6 Example Models

There are two example models which use dynamic pattern matching included with this unit and both are variations of previous models seen in the tutorial. The first is an expanded version of the semantic model from unit 1 and the other is a refinement of the paired associate experiment model which performed the task based on instructions as presented in unit 7. Both models operate similarly to how they did before, and therefore will not be described in full detail here. Only the significant differences related to dynamic pattern matching will be discussed.

8.2.6.1 Semantic Model

In unit 1 the model used a somewhat cumbersome representation of the knowledge in declarative memory specifying slots named attribute and value instead of just using slots and values:

```
(p1 ISA property object shark attribute dangerous value true)
(p2 ISA property object shark attribute locomotion value swimming)
(p3 ISA property object shark attribute category value fish)
```

The reason for that representation at the time was because without dynamic pattern matching the model would have required specifying separate productions for each of the attributes which one wanted to lookup. The version in the semantic-dynamic-model.lisp file of this unit uses a more natural representation of the chunks like this which can be searched using dynamic pattern matching:

```
(p1 ISA property object shark dangerous true)
(p2 ISA property object shark locomotion swimming)
(p3 ISA property object shark category fish)
```

To go along with that the model specifies the property chunk-type as having all the possible attributes used:

```
(chunk-type property object category dangerous locomotion edible
            breathe moves skin color sings flies height wings)
```

That is not necessary, because just as with productions, specifying new slots when creating chunks in the model definition will automatically extend them and print a warning. For example if we add this property to the model's add-dm call:

```
(p25 isa property object shark teeth sharp)
```

We get this warning when the model is loaded and when it is reset:

```
#|Warning: Invalid slot TEETH specified when creating chunk P25 with type PROPERTY.  Extending
chunks with slot named TEETH. |#
```

However, as was recommended with productions, all slot names used as constants in the model should be declared in advance. Of course, if this model also included a means to acquire new facts in some way instead of having them all specified in the definition it could create those new slots through dynamic modifications in the productions.

Instead of only being able to chain through category tests this model is now able to search for any attribute and value and will search back through the categories as necessary to try and find it. Here are some goal chunks which the model starts out with as options to test:

```
(g1 ISA test-attribute object canary attribute category value bird)
(g2 ISA test-attribute object canary attribute wings value true)
(g3 ISA test-attribute object canary attribute dangerous value true)
```

The goal chunks now contain slots for the attribute and value to indicate what is being searched for and here are the productions which use that information dynamically to request a retrieval of the desired fact and to verify when it has been successfully retrieved:

```
(p retrieve-attribute
   =goal>
     ISA         test-attribute
     object      =obj
     attribute   =attr
     value       =val
     state       nil
  ==>
```

```
(P direct-verify
   =goal>
     ISA         test-attribute
     original    nil
     object      =obj
     attribute   =attr
     value       =val
     state       attr-check
```

```
   =goal>                                  =retrieval>
     state       attr-check                  ISA         property
   +retrieval>                               object      =obj
     ISA         property                    =attr       =val
     object      =obj                    ==>
     - =attr     nil)                     =goal>
                                            answer      yes
                                            state       done)
```

The retrieve-attribute production requests the retrieval of a chunk which has a value in the slot with the named attribute, and then the direct-verify production matches if that retrieved chunk has the requested value in that slot. You may wonder why we don't just request the retrieval of the specific slot value desired in retrieve-attribute, and the reason is that the model may contain a contradictory fact which would fail to be retrieved if that were the case and it would then search through the other categories to try and find the matching chunk. For example, if asked whether ostriches fly with a goal like this:

```
(g1 ISA test-attribute object ostrich attribute flies value true)
```

and given the fact in declarative memory which indicates they do not fly:

```
(p15 ISA property object ostrich flies false)
```

A model which requested that specific slot value combination of "flies true" would fail to retrieve that chunk about ostriches. This is very similar to the fan model from unit 5 which only requested a fact based on one of the items it contained, and as a general strategy it is almost always better for the model to retrieve something and then test that result than to rely on retrieval failure.

The other productions in this model which handle failure to retrieve, retrieving a contradictory fact, and chaining up through the categories to continue searching are similar to those from the unit 1 model, but now use dynamic pattern matching since the attribute is provided in the **goal** buffer chunk. You should be able to understand how those work now and we will not describe them.

One final thing to note about this model is that when it finds a matching fact in a parent category it adds that specific fact to the model so that future searches will not require going through all of the categories again. Here are the traces of two successive runs asking for the same fact specified in the g2 goal provided:

```
? (goal-focus g2)
G2
? (run 1)
     0.000   GOAL                 SET-BUFFER-CHUNK GOAL G2 NIL
     0.000   PROCEDURAL           CONFLICT-RESOLUTION
     0.050   PROCEDURAL           PRODUCTION-FIRED RETRIEVE-ATTRIBUTE
     0.050   PROCEDURAL           CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE          start-retrieval
     0.050   PROCEDURAL           CONFLICT-RESOLUTION
     0.100   DECLARATIVE          RETRIEVAL-FAILURE
     0.100   PROCEDURAL           CONFLICT-RESOLUTION
     0.150   PROCEDURAL           PRODUCTION-FIRED MISSING-ATTR
     0.150   PROCEDURAL           CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE          start-retrieval
     0.150   PROCEDURAL           CONFLICT-RESOLUTION
     0.200   DECLARATIVE          RETRIEVED-CHUNK P14
```

```
     0.200   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL P14
     0.200   PROCEDURAL              CONFLICT-RESOLUTION
     0.250   PROCEDURAL              PRODUCTION-FIRED CHAIN-FIRST-CATEGORY
     0.250   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.250   PROCEDURAL              CONFLICT-RESOLUTION
     0.300   PROCEDURAL              PRODUCTION-FIRED RETRIEVE-ATTRIBUTE
     0.300   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.300   DECLARATIVE             start-retrieval
     0.300   PROCEDURAL              CONFLICT-RESOLUTION
     0.350   DECLARATIVE             RETRIEVED-CHUNK P18
     0.350   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL P18
     0.350   PROCEDURAL              CONFLICT-RESOLUTION
     0.400   PROCEDURAL              PRODUCTION-FIRED INDIRECT-VERIFY
     0.400   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.400   PROCEDURAL              CONFLICT-RESOLUTION
     0.450   PROCEDURAL              PRODUCTION-FIRED RECORD-RESULT
     0.450   PROCEDURAL              CLEAR-BUFFER IMAGINAL
     0.450   PROCEDURAL              CONFLICT-RESOLUTION
     0.650   IMAGINAL                SET-BUFFER-CHUNK-FROM-SPEC IMAGINAL
     0.650   PROCEDURAL              CONFLICT-RESOLUTION
     0.700   PROCEDURAL              PRODUCTION-FIRED CREATE
     0.700   PROCEDURAL              CLEAR-BUFFER IMAGINAL
     0.700   PROCEDURAL              CONFLICT-RESOLUTION
     0.750   PROCEDURAL              PRODUCTION-FIRED RESPOND
CANARY WINGS TRUE ANSWER IS YES
     0.750   PROCEDURAL              CLEAR-BUFFER GOAL
     0.750   PROCEDURAL              CONFLICT-RESOLUTION
     0.750   ------                  Stopped because no events left to process
0.75
84
NIL
? (goal-focus g2)
G2
? (run 1)
     0.750   GOAL                    SET-BUFFER-CHUNK GOAL G2 NIL
     0.750   PROCEDURAL              CONFLICT-RESOLUTION
     0.800   PROCEDURAL              PRODUCTION-FIRED RETRIEVE-ATTRIBUTE
     0.800   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.800   DECLARATIVE             start-retrieval
     0.800   PROCEDURAL              CONFLICT-RESOLUTION
     0.850   DECLARATIVE             RETRIEVED-CHUNK IMAGINAL-CHUNK0-0
     0.850   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL IMAGINAL-CHUNK0-0
     0.850   PROCEDURAL              CONFLICT-RESOLUTION
     0.900   PROCEDURAL              PRODUCTION-FIRED DIRECT-VERIFY
     0.900   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.900   PROCEDURAL              CONFLICT-RESOLUTION
     0.950   PROCEDURAL              PRODUCTION-FIRED RESPOND
CANARY WINGS TRUE ANSWER IS YES
     0.950   PROCEDURAL              CLEAR-BUFFER GOAL
     0.950   PROCEDURAL              CONFLICT-RESOLUTION
     0.950   ------                  Stopped because no events left to process
```

8.2.6.2 Paired-learning Model

A new version of the paired associate model is included with this unit in the paired-dynamic-model.lisp file. This model is similar to the one from the previous unit in that it starts with instructions for doing the task, but the representation of the chunks which it uses differ from those used in the last unit. For the operator chunks, instead of having two generic slots named arg1 and arg2 which get used differently for different subtasks it now uses appropriately named slots for each operation:

```
 (op1 isa operator pre start         action read     label word    post stimulus-read)
 (op2 isa operator pre stimulus-read action retrieve required word label number post recalled)
 (op3 isa operator pre recalled      slot   number   success respond            failure wait)
 (op4 isa operator pre respond       action type     required number           post wait)
 (op5 isa operator pre wait          action read     label number  post new-trial)
 (op6 isa operator pre new-trial     action complete-task          post start)
```

It also does not specify a chunk-type for the result. Instead, it just creates an empty chunk that gets the appropriate slots added to it such that the resulting chunks for the trials will look like this:

```
IMAGINAL-CHUNK0-0
   WORD  "zinc"
   NUMBER  "9"
```

Those slot names are encoded in the instructions in the label slot of operators op1 and op5 above. The label slot of the action specifies the name of the slot into which the item should be stored in the **imaginal** buffer and is setup by the **read** production storing the label into the context slot of the chunk in the **goal** buffer:

```
(p read
   =goal>
     isa         task
     step        retrieving-operator
   =retrieval>
     isa         operator
     action      read
     label       =destination
     post        =state
   =visual-location>
   ?visual>
     state       free
   ?imaginal>
     buffer      full
  ==>
   +visual>
     isa         move-attention
     screen-pos =visual-location
   =goal>
     context     =destination
     step        process
     state       =state)
```

and then the **encode** production modifies the chunk in the **imaginal** buffer using that value to indicate the slot:

```
(p encode
   =goal>
     isa          task
     step         process
     context      =which-slot
   =visual>
     isa          text
     value        =val
   =imaginal>
   ?imaginal>
     state        free
  ==>
   *imaginal>
     =which-slot =val
   =goal>
     step         ready)
```

That makes this set of instruction following productions more general than the previous ones because the representation does not have to be encoded in the model's chunk-types and could have any number of slots in the representation based on the instructions. The model also does not need to have a separate production to set each possible slot that is used in the representation. Of course, since the instructions are specifically encoded in this particular model's definition that generality is not necessary since the representation is known in advance. However, these same productions could be used for other tasks without changing them, or they could be used with an even more general version of the model which had other productions for reading the instructions themselves.

Similarly, we see that by using dynamically matched productions this new version of the model can also perform the retrieval based on a slot specified in the instructions:

```
(p retrieve-associate
   =goal>
     isa      task
     step     retrieving-operator
   =imaginal>
     =target  =stimulus
   =retrieval>
     isa      operator
     action   retrieve
     required =target
     label    =other
     post     =state
  ==>
   +retrieval>
     =target  =stimulus
   =goal>
     step     retrieving-result
     context  =other
     state    =state)
```

and also respond using an arbitrary slot specified in the instructions:

```
(p type
   =goal>
     isa      task
     step     retrieving-operator
   =imaginal>
     =slot    =val
   =retrieval>
     isa      operator
     action   type
     required =slot
     post     =state
   ?manual>
     state    free
  ==>
   +manual>
     cmd      press-key
     key      =val
   =goal>
     state    =state
     step     ready)
```

8.2.6.3 Dynamic matching and production compilation

As in the previous version of this task, this model uses production compilation to learn specific productions for doing the task as it is repeatedly following the instructions. There is no difference in how the production compilation mechanism works with dynamically matched productions. It still combines successive productions which are fired into a single production when possible using the rules described in unit 7. However, it is worth looking at a couple of examples to see how that applies when dynamically matched productions are involved. When one or both of the productions being composed contain dynamically matched components, the resulting production may retain some of those dynamic components or they may be replaced with statically matched values. They will be replaced by static values in the same way that other variables are replaced – when a retrieval request and harvesting are removed between the productions the removed variables' bindings are instantiated.

We can see examples of this very early in the model's trace since it has the :pct parameter set to t. [To run this model you will need to load/import the paired.lisp or paired.py file before loading this model file, and then use the functions described in unit 4 to run the experiment or a subset of it.] When the encode and retrieve-operator productions are combined the result still has the dynamic component from the encode production:

```
     0.400   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-OPERATOR
Production Compilation process started for RETRIEVE-OPERATOR
  Production ENCODE and RETRIEVE-OPERATOR are being composed.
  New production:

(P PRODUCTION1
  "ENCODE & RETRIEVE-OPERATOR"
   =GOAL>
       CONTEXT =WHICH-SLOT
       STEP PROCESS
       STATE =STATE
   =IMAGINAL>
   =VISUAL>
       TEXT T
       VALUE =VAL
   ?IMAGINAL>
       STATE FREE
 ==>
   =GOAL>
       CONTEXT NIL
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE =STATE
   *IMAGINAL>
       =WHICH-SLOT =VAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-OPERATOR.
```

However, when the retrieve-operator and retrieve-associate productions are combined the instantiation of the variables used from the retrieved chunk results in a production with no dynamic components:

```
     0.600   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-ASSOCIATE
Production Compilation process started for RETRIEVE-ASSOCIATE
  Production RETRIEVE-OPERATOR and RETRIEVE-ASSOCIATE are being composed.
  New production:

(P PRODUCTION2
  "RETRIEVE-OPERATOR & RETRIEVE-ASSOCIATE - OP2"
   =GOAL>
       STATE STIMULUS-READ
       STEP READY
   =IMAGINAL>
       WORD =STIMULUS
 ==>
   =GOAL>
       CONTEXT NUMBER
       STATE RECALLED
       STEP RETRIEVING-RESULT
   +RETRIEVAL>
       WORD =STIMULUS
)
Parameters for production PRODUCTION2:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-ASSOCIATE.
```

8.2.6.4 Data fit

Despite the model being more general, it still performs the task the same way as the previous version of the model, and with production compilation enabled provides a similar fit to the data using the same parameter settings as the model from the previous unit:

```
Latency:
CORRELATION:  0.987
MEAN DEVIATION:  0.112
Trial    1       2       3       4       5       6       7       8
        0.000   2.032   1.929   1.817   1.716   1.671   1.614   1.610

Accuracy:
CORRELATION:  0.992
MEAN DEVIATION:  0.045
Trial    1       2       3       4       5       6       7       8
        0.000   0.430   0.710   0.800   0.870   0.900   0.900   0.930
```

8.3 Assignment

The assignment for this unit will be to use both of the new production techniques described above to create a model which can perform a simple categorization task. The experiment this model will be performing is a simplification of an experiment which was performed by Robert M. Nosofsky (which was itself based on an experiment performed by Stephen K. Reed) that required participants to classify schematic face drawings into one of two learned categories.

In the experiment, the participants first trained on learning 10 faces each of which belonged to one of two categories. The faces themselves were varied along four features: eye height, eye separation, nose length, and mouth height. Then there was a testing phase in which they were presented with both old and new faces and asked to specify to which category the face belonged. The data collected was the probability of classifying the face as a member of each category in the testing phase. For this assignment we will not be modeling the whole task, nor will we be trying to fit all of the data from the experiment because a thorough model of this task would require a lot more work than is reasonable for an assignment.

Here is the general description of the task which the model for this assignment will have to perform. It will be presented with the attributes of a stimulus one at a time, indicating the name of a feature and its value (it will not be visually interpreting a face image). It must collect those attributes into a single chunk which represents the current stimulus. Using that chunk, it can then retrieve a best matching example from declarative memory. Based on that retrieved chunk, it will make a category choice for the current stimulus. The model will not be performing the initial training phase, thus it will not require using any of the ACT-R learning mechanisms. Each trial will be completed separately with the model being reset before each one. This is similar to how the fan experiment model from unit 5 worked, with the training information pre-encoded in declarative memory and the model only needing to perform one trial of the task. The details of how those steps are to be performed are described below.

The primary part of the exercise is the first step – creating the stimulus representation from the individual attributes. The important aspect of the assignment will be on how to perform this task in a general way because that sort of encoding is something which can be applicable to many different tasks. Overall, this should be a very small model, only requiring around 5 productions to do the task, but will require using both of the new mechanisms described in this unit.

8.3.1 The Stimulus Attributes

The attributes of the stimulus to categorize will be presented to the model one at a time through the **goal** buffer. The experiment code will set the **goal** buffer to a chunk which uses the slots specified for the chunk-type named goal:

```
(chunk-type goal state name value)
```

The state slot will be set to the value add-attribute, the name slot of the chunk will contain a symbol specifying the name of the attribute being presented. The value slot will hold a value for that attribute in the current stimulus which will be a number. Thus, a goal chunk at the start of a trial may look like this:

```
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   NAME  EH
   STATE  ADD-ATTRIBUTE
   VALUE  0.704
```

What the model must do is convert that numeric value into a symbolic description which will be stored in a slot of the **imaginal** buffer. The reason for doing so is because the example chunks which the model has stored in declarative memory that it will be retrieving look like this:

```
EXAMPLE1
   CATEGORY  1
   EH  MEDIUM
   ES  SMALL
   NL  LARGE
   MH  MEDIUM
```

Those chunks provide a general representation of the stimuli instead of just recording explicit measurements or distance values. By converting the attributes as they are encoded the model will be able to create a similarly structured chunk in the **imaginal** buffer. In a more complete model of the task a similar process would take place during the training phase of the experiment to strengthen and learn the examples, and we want to maintain that same representational aspect even if it is not currently being modeled.

For this task, all of the attributes can be classified as either small, medium, or large, and to make things easier, the numeric values of the attributes used have all been scaled to the same range based on data also collected by Nosofsky for this task. To perform that conversion from numeric value to descriptive name, procedural partial matching should be used by the model. The starting model has a similarity function provided that assigns a similarity score between the attribute values and the labels small, medium, and large. Thus, one can have competing productions which test the value for each of those labels and the production which specifies the most similar label will be the one selected.

The specific range of values used should not be explicitly encoded into the model, and in fact this unit will not describe how the similarity values are computed between the labels and the numbers. Thus, your model should not explicitly test the values against any particular numbers, but should be able to work with any value given relying on the similarity function to provide the comparison to the appropriate labels. Thus, a production like this is ***not*** a good thing to use in this model:

```
(p very-bad-way-to-test-for-medium
   =goal>
     isa attribute
    > value -.5
    < value .5
  ...
)
```

After determining what the appropriate label for the attribute is, the model must record that value in the chunk in the **imaginal** buffer. When the first attribute is presented, the **imaginal** buffer will be set to a chunk which only has the value unknown in the category slot:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   CATEGORY  UNKNOWN
```

It does not have any slots for holding the specific attributes of the task. This is where dynamic pattern matching will need to be used. The model will need to add a slot to the chunk based on the name of the attribute provided. Thus, the complete encoding of the attribute shown above would result in the **imaginal** buffer looking like this when done if that value were mapped to a large result:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   CATEGORY  UNKNOWN
   EH  LARGE
```

Again, it is important that the model be general in how it performs that modification to the **imaginal** buffer's chunk. The model should be able to encode any attribute name which is provided, and should not have any specific attribute names mentioned in the productions. Thus this action in a production is ***not*** the way that it should be handled:

```
(p bad-imaginal-attribute-encoding
...
==>
  =imaginal>
     eh  =value
...)
```

As with the values, the unit will not be specifying the names that the attributes will have. The experiment code will guarantee that the example chunks created in declarative memory have the same attributes as those that are presented to the model, and the model should be able to work with any attribute names it is given.

After updating the **imaginal** buffer, the model should stop and wait for the next attribute to be provided. An easy way to do that would be to make sure that all of the productions which are processing the attribute test the **goal** buffer chunk in their conditions (which is likely to occur since it contains the information needed) and then clear the chunk from the **goal** buffer after the attribute has been processed. Thus, the model should be able to process any number of attributes for a stimulus. The result will be a chunk in the **imaginal** buffer which has a slot for each of the attributes that was provided and the value in each of those slots is a label (small, medium, or large) as determined through procedural partial matching based on the numeric value from the attribute.

8.3.2 Model Response

After all of the attributes have been provided to the model a different goal chunk will be set to indicate that it is time for the model to retrieve an example and classify the stimulus that it has encoded in the **imaginal** buffer. That **goal** buffer's chunk will have the state slot set to the value categorize:

```
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   STATE  CATEGORIZE
```

To make a response, the model needs to do two things. First, it must retrieve a chunk from declarative memory which is similar to the chunk in the **imaginal** buffer. The model will have 10 examples already encoded in declarative memory which have their features set based on the examples from the original experiment. Half of the examples are in category 1 and the other half are in category 2, and here are two examples which were created with the default attribute names:

```
EXAMPLE4
   CATEGORY  1
   EH  SMALL
   ES  MEDIUM
   NL  LARGE
   MH  LARGE
```

```
EXAMPLE5
   CATEGORY  2
   EH  LARGE
   ES  SMALL
   NL  SMALL
   MH  SMALL
```

Because the names of the slots to specify in the retrieval request are not known in advance the easiest way to perform the retrieval request is using an indirect request with the chunk from the **imaginal** buffer, as was shown in the siegler model from unit 5 of the tutorial. That will look like this as an action in the production:

```
(p indirect-retrieval-request
  =imaginal>
 ...
 ==>
  +retrieval> =imaginal
 ...)
```

and is equivalent to specifying all of the slots and values from the chunk that is in the **imaginal** buffer explicitly in the request.

Declarative partial matching is also enabled for this model, and that will allow for the retrieval of a chunk which has values close to the requested values in the event that there is not a perfectly matching example. The other declarative parameters for the model are set such that if the chunk in the **imaginal** buffer is created correctly there should always be a chunk retrieved in a reasonable amount of time.

After the model retrieves a chunk, the final step it needs to do is to set the category slot of the chunk in the **imaginal** buffer to be the same as the category value of the chunk that was retrieved. After that happens the model should again stop because it is then done with the trial.

8.3.3 Running the experiment

There are three commands provided for running the task. The first one performs one step of the attribute presentation. The categorize-attribute function in Lisp and the attribute function in the categorize module of Python each take two parameters. The first is the name for an attribute (which should be a symbol in Lisp and a string in Python) and the second is the numeric value for that attribute (which should be between -1 and 1). It will generate an attribute goal chunk for the model with the name and value slots set using the values provided and a state slot value of add-attribute then run the model to perform the encoding. Here are some example calls and the goal chunk that will be created:

```
? (categorize-attribute size -.9)
>>> categorize.attribute('size',-.9)

GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   NAME  SIZE
   STATE  ADD-ATTRIBUTE
   VALUE  -0.9
```

That should lead to the model creating a chunk like this in the **imaginal** buffer when it is done:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   SIZE  SMALL
   CATEGORY  UNKNOWN
```

That's because a value of -.9 is most similar to the small label with the default settings (although noise in the utility calculations may cause one of the other labels to be applied instead). You should work with this command until your model is able to do that part of the task reliably. Note that this command does not reset the model, so if you call it again the chunk in the **imaginal** buffer should contain both of the slots specified. Therefore, if after the previous call you then called one of these:

```
? (categorize-attribute s2 1.0)
>>> categorize.attribute('s2',1.0)
```

The chunk in the **imaginal** buffer should look like this:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   SIZE  SMALL
   CATEGORY  UNKNOWN
   S2  LARGE
```

Both slots should have a descriptive value, and the second one will most likely be large since 1.0 is most similar to large with the default settings. In the full task the model will be required to do that process four times, but the model should be capable of handling any number of attributes presented to it, each time adding the new slot to the chunk in the **imaginal** buffer with its corresponding description.

After the model is able to properly encode attributes, you can test its ability to retrieve examples based on a stimulus with multiple attributes using the categorize-stimulus function in Lisp or the stimulus function from the categorize module in Python. It requires four parameters which should each be a number in the range of -1 to 1. It will reset the model, generate the four attribute goal chunks for the model (using a default set of names for the attributes) like the attribute action above, and then it will generate a categorize goal and run the model to make the response. The goal chunk will look like this when the model needs to categorize the item that has been encoded in the **imaginal** buffer:

```
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   STATE  CATEGORIZE
```

To categorize the item it must set the category slot of the chunk in the **imaginal** buffer to the category value of the item which it retrieves from declarative memory that is most similar to the chunk in the **imaginal** buffer. The default chunks created for the features in declarative memory have category values of 1 and 2, but the model should not make any assumptions about those values and just use the value from the chunk which it retrieves. The return value from the stimulus function will be the category which the model set. If this stimulus description were used:

```
? (categorize-stimulus 1 -1 -1 -1)
>>> categorize.stimulus(1,-1,-1,-1)
```

The model should end up with a chunk like this in the **imaginal** buffer indicating that it most closely matches an instance from category 2, and the functions will return a value of 2 (again because of noise it may not always produce that same description and categorization):

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   CATEGORY  2
   EH  LARGE
   ES  SMALL
   NL  SMALL
   MH  SMALL
```

Once your model can reliably encode and categorize individual stimuli you can move on to try it on the whole experiment using the categorize-experiment function in Lisp or the experiment function from the categorize module in Python. It requires one parameter which is the number of times to repeat the experiment. It will run the model over the 14 stimuli from the experiment as many times as specified and print out the proportion of times that each item was classified as being in category 1 along with the experimental data, number of responses the model made, and the model's fit to the experimental data. A call like this:

```
? (categorize-experiment 10)
>>> categorize.experiment(10)
```

Should result in output similar to this showing that the model responded 10 times on each trial (the number in parentheses):

```
CORRELATION:  0.922
MEAN DEVIATION:  0.165
P(C=1)
        ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10) ( 10)
data   0.975 0.850 0.987 1.000 0.963 0.075 0.138 0.087 0.050 0.025 0.937 0.544 0.988 0.087
model  1.000 0.700 1.000 0.900 0.700 0.100 0.000 0.100 0.000 0.000 0.800 0.800 1.000 0.500
```

8.3.4 Fitting the data

If your model works as described above, then it should produce a fit to the data similar to this using the default parameters in the model when run over many trials:

```
CORRELATION:  0.955
MEAN DEVIATION:  0.175
P(C=1)
        (100) (100) (100) (100) (100) (100) (100) (100) (100) (100) (100) (100) (100) (100)
data   0.975 0.850 0.987 1.000 0.963 0.075 0.138 0.087 0.050 0.025 0.937 0.544 0.988 0.087
model  0.910 0.740 0.930 0.890 0.830 0.270 0.220 0.380 0.210 0.130 0.860 0.770 0.970 0.480
```

If the correlation and deviation are significantly worse than that, then you will want to go back and make sure your model is doing all of the steps described above correctly.

Because fitting the data is not the focus of this exercise it should not be necessary to adjust the parameters to improve that fit, but if you would like to explore the parameters for improving the fit then the ones that are recommended to be changed would be these four from the model:

```
(sgp :mp 1 :ppm 1 :egs .25 :ans .25)
```

Those are the partial matching scale parameters for declarative and procedural partial matching and the noise values for those mechanisms. Adjusting those parameters should allow you to decrease the mean deviation from the data.

8.3.5 Generality

If your model fits the data adequately then the final thing to test is to make sure that it is doing the task in a general way. The function for running the experiment takes optional parameters which can be used to modify the attribute names and the range of values it uses. Providing a second parameter which is a number will shift the attribute values provided to the model by that amount and also shift the similarities to the labels accordingly. Thus, the model should be unaffected by that change and produce the same fit to the data regardless of the number provided for the shift:

```
? (categorize-experiment 100 4.5)
>>> categorize.experiment(100,4.5)
```

In addition to providing an offset value for the experiment, one can also specify the four names to use for the attributes. Those provided names will be used to generate the examples in declarative memory and to provide the attributes to the model. The names can be anything that is valid for a slot name in ACT-R except for the name category since that slot is already used to specify the categorization of the items. Here are examples that specify names of a, b, c, and d along with an offset of -3:

```
? (categorize-experiment 100 -3 a b c d)
>>> categorize.experiment(100,-3,'a','b','c','d')
```

Changing the attribute names and the offset should not affect the model's ability to do the task if it has been written to perform the task generally.

References

Nosofsky, R. M. (1991). Tests of an exemplar model for relating perceptual classification and recognition memory. *Journal of Experimental Psychology: Human Perception and Performance. 17*, 3-27.

Reed, S. K. (1972). Pattern recognition and categorization. *Cognitive Psychology. 3*, 382-407.

Stocco, A., Lebiere, C., & Anderson, J. R. (2010). Conditional routing of information to the cortex: A model of the basal ganglia's role in cognitive coordination. *Psychological Review, 117*(2), 540-574.