Unit 1: Introduction to ACT-R

ACT-R is a cognitive architecture. It is a theory of the structure of the brain at a level of abstraction that explains how it achieves human cognition. That theory is instantiated in the ACT-R software which allows one to create models which may be used to explain performance in a task and also to predict performance in other tasks. This tutorial will describe how to use the ACT-R software for modeling and provide some of the important details about the ACT-R theory. Detailed information on the ACT-R theory can be found in the paper "An integrated theory of the mind", which is available on the ACT-R website and also in the book "How Can the Human Mind Occur in the Physical Universe?". More information on the ACT-R software can be found in the reference manual which is included in the docs directory of the ACT-R software distribution.

The goals of this unit are to introduce the basic components of the ACT-R architecture and show how those components are used to create and run a model in the ACT-R software.

1.1 Knowledge Representations

There are two types of knowledge representation in ACT-R -- **declarative** knowledge and **procedural** knowledge. Declarative knowledge corresponds to things we are aware we know and can usually describe to others. Examples of declarative knowledge include "George Washington was the first president of the United States" and "An atom is like the solar system". Procedural knowledge is knowledge which we display in our behavior but which we are not conscious of. For instance, no one can describe the rules by which we speak a language and yet we do. In ACT-R, declarative knowledge is represented in structures called **chunks** and procedural knowledge is represented as rules called **productions**. Chunks and productions are the basic building blocks of an ACT-R model.

1.1.1 Chunks in ACT-R

In ACT-R, elements of declarative knowledge are called **chunks**. Chunks represent knowledge that a person might be expected to have when they solve a problem. A chunk is a collection of attributes and values. The attributes of a chunk are called **slots**. Each slot in a chunk has a single value. A chunk also has a name which can be used to reference it, but that name is only a convenience for using the ACT-R software and is not considered to be a part of the chunk itself. Below are some representations of chunks that encode the facts that *the dog chased the cat* and that *4+3=7*. The chunks are displayed as a name and then slot and value pairs. The name of the first chunk is **action023** and its slots are **verb**, **agent,** and **object**, which have values of chase, dog, and cat respectively. The second chunk is named **fact3+4** and its slots are **addend1**, **addend2**, and **sum**, with values three, four, and seven.

```
Action023
    verb chase
    agent dog
    object cat
```

```
Fact3+4
    addend1 three
    addend2 four
    sum seven
```

1.1.2 Productions in ACT-R

A production is a statement of a particular contingency that controls behavior. They can be represented as if-then rules and some examples might be:

IF the goal is to classify a person

  and he is unmarried

THEN classify him as a bachelor

IF the goal is to add two digits d1 and d2 in a column

  and d1 + d2 = d3

THEN create a goal to write d3 in the column

The condition of a production (the IF part) consists of a conjunction of features which must be true for the production to apply. The action of a production (the THEN part) consists of the actions the model should perform when the production is selected and used. The above are informal English specifications of productions. They give an overview of when the productions apply and what actions they should perform, but do not specify sufficient detail to actually implement a production in ACT-R. You will learn the specific syntax for creating productions in ACT-R later in this unit.

1.2 The ACT-R Architecture

The ACT-R architecture consists of a set of **modules**. Each module performs a particular cognitive function and operates independently of other modules. We will introduce three modules in this unit and describe their basic operations. Later units will provide more details on the operations of these modules and also introduce other modules.

The modules communicate through an interface we call a **buffer**. Each module may have any number of buffers for communicating with other modules. A buffer relays requests to its module to perform actions, it responds to queries about the status of the module and the buffer itself, and it can hold one chunk at a time which is usually placed into the buffer as the result of an action which was requested. The chunk in a buffer is available for any module to see and modify and effectively works like a scratch-pad for creating and modifying chunks.

1.2.1 Goal Module

The goal module is the simplest of the modules in ACT-R. It has one buffer named **goal** which is used to hold a chunk which contains the current control information the model needs for performing its current task. The only request to which the module responds is for the creation of a new goal chunk. It responds to the request by immediately creating a chunk with the information contained in the request and placing it into the **goal** buffer.

1.2.2 Declarative Module

The declarative module stores all of the chunks which represent the declarative knowledge the model has which is often referred to as the model’s declarative memory. It has one buffer named **retrieval**. The declarative module responds to requests by searching through declarative memory to find the chunk which best matches the information specified in the request and then placing that chunk into the **retrieval** buffer. In later units we will cover that process in more detail to describe how it determines the best match and how long the process takes. For the models in this unit, there will never be more than one chunk which matches the request and the time cost will be fixed in the models at 50 milliseconds per request.

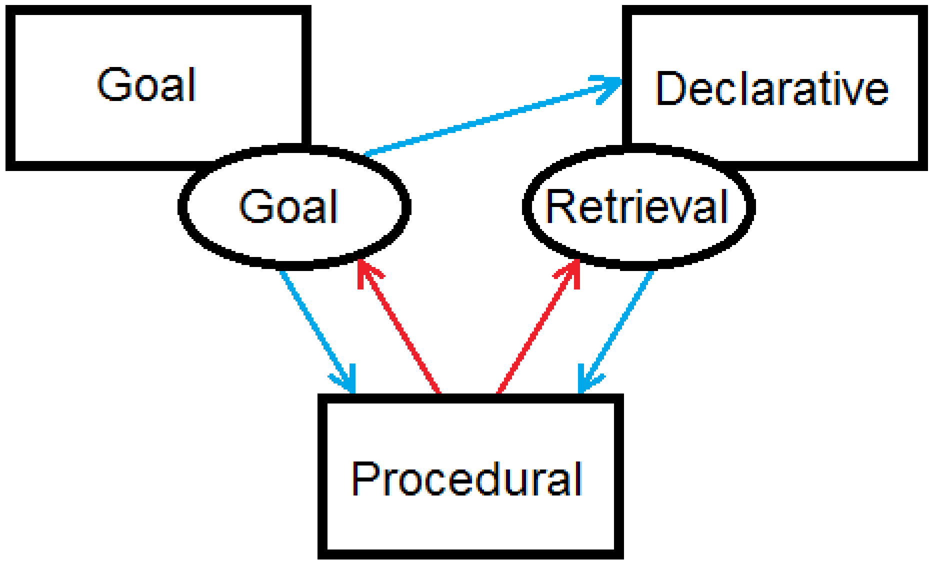
The declarative memory in a model consists of the chunks which are placed there initially by the modeler when defining the model and the knowledge which it learns as it runs. The learned knowledge is collected from the buffers of all of the modules. The declarative module monitors all of the buffers, and whenever a chunk is cleared from one of them the declarative module stores that chunk for possible later use.

1.2.3 Procedural Module

The procedural module holds all of the productions which represent the model’s procedural knowledge. It does not have a buffer of its own, and unlike other modules the procedural module does not take requests for actions. Instead, it is constantly monitoring the activity of all the buffers looking for patterns which satisfy the condition of some production. When it finds a production which has its condition met then it will execute the actions of that production, which we refer to as “firing” the production. Only one production can fire at a time and it takes 50 milliseconds from the time the production was matched to the current state until the actions happen. In later units we will look at what happens if more than one production matches at the same time, but for this unit all of the productions in the models will have their conditions specified so that at most one will match at any point in time.

1.2.4 Overview

These three modules are used in almost every model which is written in ACT-R, and older versions of ACT-R consisted entirely of just these three components. Here is a diagram showing how they fit together in the architecture with the rectangles representing the modules and the ovals representing the buffers:



The blue arrows show which modules read the information from another module's buffer, and the red arrows show which modules make requests to another module's buffer or directly modify the chunk it contains. As we introduce new modules and buffers in the tutorial, each of the buffers will have the same interface as shown for the goal buffer -- both the procedural and declarative modules will read the buffer information and the procedural module will modify and make requests to the buffer.

1.3 ACT-R Software and Models

Now that we have described the basic components in ACT-R, we will step back and describe the ACT-R software and how one creates and runs a model using it. This tutorial is written for version 7.12 (or newer) of the ACT-R software. Unlike previous versions, version 7.12 of the software is distributed primarily as applications and the tutorial instructions will assume that the user is running one of those applications, but like previous versions the source code is available for those that prefer to run from sources. The main ACT-R software is implemented in the ANSI Common Lisp programming language, but it is not necessary to know how to program in Lisp to be able to use ACT-R because it is essentially its own language. In prior versions of the software it was necessary to interact with ACT-R through Lisp code, and one had to write Lisp code to build experiments or other tasks for the model to perform. However, starting with version 7.6, ACT-R provides a remote interface that can be used to interact with ACT-R from essentially any programming language, and the tutorial materials include a Python module (in the Python use of the term module not the ACT-R use as a cognitive component of the architecture) which provides functions for accessing ACT-R through that remote interface. Using that Python module, all of the tasks for the models in the tutorial are implemented in Python as well as the original Lisp implementations of the tasks, and either version can be used with the models of the tutorial. There are also examples included with ACT-R for connecting other languages, but only Lisp and Python will be used in the tutorial.

1.3.1 Starting ACT-R

To run the ACT-R software you need to run the startup script named *run-act-r* that is included with the standalone version for your operating system (versions are available for Linux, macOS, and Windows). That will open two windows for the ACT-R software. The one titled "ACT-R" is an interactive Lisp session which contains the main ACT-R system and can be used to interact with ACT-R directly. The other window, titled "Control Panel", contains a set of GUI tools written in the Tcl/Tk programming language and is called the ACT-R Environment. The ACT-R Environment is an optional set of tools for interacting with the ACT-R system and the use of some of those tools will be described during the tutorial. Additional information on running the software can be found in the readme.txt file, and more information on the Environment tools can be found in the Environment's manual included in the docs directory of the software. You may close the ACT-R Environment window if you do not wish to use those tools (note however that for this unit the Environment tools are necessary and it should not be closed). Closing the ACT-R window will exit the software.

For this unit there are no tasks for the models to interact with, and all the interaction with the software can be done through the ACT-R Environment. For that reason, we will not describe how to use the Python module yet since there is no need for it, but it will be described in the next unit which includes tasks for the tutorial models to perform.

1.3.2 Interacting with Lisp

Although it is not necessary to use the Lisp interface for ACT-R, it can be convenient and some familiarity with Lisp syntax can be helpful since the syntax for creating ACT-R models is based on Lisp syntax. Lisp is an interactive language and provides a prompt at which the user can issue commands and evaluate code. The prompt in the ACT-R software window is the "?" character. To evaluate (also sometimes referred to as "calling") a command in Lisp at the prompt requires placing it between parentheses along with any parameters which it requires separated by whitespace and then hitting enter or return. As an example, to evaluate the command to add the numbers 3 and 4 requires calling the command named "+" with those parameters. That means one would type this at the prompt:

```
(+ 3 4)
```

and then hit the enter or return key. The system will then print the result of evaluating that command and display a new prompt:

```
? (+ 3 4)
7
?
```

One minor issue to note is that sometimes the output from the ACT-R system will overwrite the prompt character and it will not be visible in the ACT-R window. When that happens you can still evaluate commands by entering them at the bottom of the window and pressing enter or return.

1.3.3 ACT-R models

An ACT-R model is a simulated cognitive agent. A model is typically written in a text file that contains the ACT-R commands that specify how the model works, and that model file can be opened and edited in any application that can operate on text files. Because the ACT-R model syntax is based on Lisp syntax using an editor which provides additional support for Lisp formatting, like matching parentheses and automatic indenting, can be useful but is not necessary. There is a very simple text editor included with the ACT-R Environment tools which will do parenthesis matching, but beyond that it is a very limited text editor.

As we progress through the tutorial we will describe the ACT-R commands which one can use to create models. Later in this unit we will introduce the commands for initializing a model and creating the knowledge structures described above (chunks and productions).

1.3.4 Loading a model

To use an ACT-R model file it must be "loaded" into ACT-R. There are many ways to do so, but for this unit we will simply use the button in the ACT-R Environment labeled "Load ACT-R code". In the next unit we will show how to load a model using an ACT-R command that can be called from the Lisp prompt or from an external connection, like the Python module. When the model file is loaded, the commands it contains are evaluated in order from the top down.

1.3.5 Running a model

Once a model has been loaded, you can run it. Models that do not interact with a task can typically be run by just calling the ACT-R **run** command. The **run** command requires one parameter, which is the maximum length of simulated time to run the model measured in seconds. Again, for this unit we will be using the tool in the ACT-R Environment instead of using the command itself. That tool is the "Run" button in the Environment and the text entry box to its right is where the time to run the model can be entered. The default time in that box is 10.0 which means pressing the "Run" button will run the model for up to 10 simulated seconds.

In later units, where the models will be interacting with various tasks, just pressing the "Run" button or calling the **run** command may not be sufficient because one might also have to run the task itself. When that is the case the tutorial will describe what is necessary to run the model and the task, and there is an additional text included with each unit that describes how the tasks are implemented and the ACT-R commands involved (those are the texts with a name that ends with "_code").

1.4 Creating an ACT-R Model

Creating an ACT-R model requires writing the text file which contains the ACT-R commands to specify the details of the model and the initial knowledge it contains. Also, in addition to the model's details, one often includes commands for controlling the general state of the ACT-R system itself.

1.4.1 ACT-R control commands

When creating an ACT-R model, there are two ACT-R commands for controlling the system which will almost always occur in the model file, and we will describe those commands first.

1.4.1.1 clear-all

The **clear-all** command will usually occur at the top of every model file. This command requires no parameters and tells ACT-R that it should remove any models which currently exist and return ACT-R to its initial state. It is not necessary to call **clear-all** in a model file, but unless one is planning on running multiple models together it is strongly recommended that it occur as the first command to make sure that the model starts with the system in a properly initialized state.

1.4.1.2 define-model

The **define-model** command is how one actually creates an ACT-R model. Within the call to **define-model** one specifies a name for the model and then includes all of the calls to ACT-R commands that will provide the initial conditions and knowledge for that model. When the model file is loaded, define-model will create the model with the conditions specified, and then whenever ACT-R is reset that model will be returned to that same initial state.

1.4.2 Chunk-Types

Before describing the commands for creating the model's initial knowledge with chunks and productions, we will first describe an additional component of the software which can be useful when creating a model. There is an optional capability available in the ACT-R software called a **chunk-type**. A chunk-type is a way for the modeler to specify categories for the knowledge in the model by indicating a set of slots which will be used together in the creation and testing of chunks. A chunk-type consists of a name for the chunk-type and a set of slot names. That chunk-type name may then be used as a declaration when creating chunks and productions in the model.

The command for creating a chunk type is called **chunk-type**. It requires a name for the new chunk-type to create and then any number of slot names. The general chunk-type specification looks like this:

```
(chunk-type type-name slot-name-1 slot-name-2 … slot-name-n)
```

and here are some examples which could have been used in a model which created the example chunks shown earlier:

```
(chunk-type action verb agent object)
(chunk-type addition-fact addend1 addend2 sum)
```

The first creates a chunk-type named action which includes the slots verb, agent, and object. The other creates a chunk-type named addition-fact with slots addend1, addend2, and sum.

It is important to note that using a chunk-type declaration does not directly affect the operation of the model itself – the chunk-type is not a component of the ACT-R architecture. They exist in the software to help the modeler specify the model components. Creating and using meaningful chunk-types can make a model easier to read and understand. They also allow the ACT-R software to verify that the specification of chunks and productions in a model is consistent with the chunk-types that were created for that model which allows it to provide warnings when inconsistencies or problems are found relative to the chunk-types which are specified.

Although chunk-types are not required when writing an ACT-R model, most of the models in the tutorial will be written with chunk-type declarations included, and using chunk-types is strongly recommended.

1.4.3 Creating Chunks

The command to create a set of chunks and place those chunks into the model's declarative memory is called **add-dm**. It takes any number of chunk specifications as its arguments. As an example, we will show the chunks from the **count** model included with the tutorial that will be described in greater detail later in this unit. First, here are the chunk-type specifications used in that model:

```
(chunk-type number number next)
(chunk-type count-from start end count)
```

and here is the specification of the initial chunks which are placed into that model's declarative memory:

```
(add-dm
 (one ISA number number one next two)
 (two ISA number number two next three)
 (three ISA number number three next four)
 (four ISA number number four next five)
 (five ISA number number five)
 (first-goal ISA count-from start two end four))
```

Each chunk for **add-dm** is specified in a list – a sequence of items enclosed in parentheses. The first element of the list is the name of the chunk. The name may be anything which is not already used as the name of a chunk as long as it starts with an alphanumeric character and is a valid Lisp symbol (essentially a continuous sequence of characters which does not contain any of the symbols: period, comma, single quote, double quote, back quote, left or right parenthesis, backslash, or semicolon). In the example above the names are **one**, **two**, **three**, **four**, **five**, and **first-goal**. The purpose of the name is to provide a way for the modeler to refer to the chunk. The name is not considered to be a part of the chunk, and it can in fact be omitted, which will result in the system automatically generating a unique name for the chunk.

The next component of the chunk specification is the optional declaration of a chunk-type to describe the chunk being created. That consists of the symbol **isa** followed by the name of a chunk-type. Note that here we have capitalized the isa symbol to help distinguish it from the actual slots of the chunk, but that is not necessary and in most cases the symbols and names used in ACT-R commands are not case sensitive.

The rest of the chunk specification is pairs of a slot name and a value for that slot. The slot-value pairs can be specified in any order and the order does not matter. When a chunk-type declaration is provided, it is not necessary to specify a value for every slot indicated in that chunk-type, but if a slot which is not specified in that chunk-type is provided ACT-R will generate a warning to indicate the potential problem to the modeler.

1.4.4 Creating Productions

As indicated above, each production is a condition-action rule. Those rules are used by the procedural module to monitor the buffers of all the other modules to determine when to perform actions. The condition specifies tests for the contents of the buffers as well as the general state of the buffers and their modules. The action of a production specifies the set of operations to perform when the production is fired, and will consist of changes to be made to the chunks in buffers along with new requests to be sent to the modules.

The command for creating a production in ACT-R is **p**, and the general format for creating a production is:

```
(p Name "optional documentation string"
  buffer tests
==>
  buffer changes and requests
)
```

Each production must have a unique name and may also have an optional documentation string to describe it. That is followed by the condition for the production. The *buffer tests* in the condition of the production are patterns to match against the current buffers' contents and queries of the buffers for buffer and module state information. The condition of the production is separated from the action of the production by the three character sequence `==>`. The production's action consists of any buffer changes and requests which the production will make.

In separate subsections to follow we will describe the syntax involved in specifying the condition and the action of a production. In doing so we will use an example production that counts from one number to the next based on a chunk which has been retrieved from the model's declarative memory. It is similar to those used in the example models for this unit, but is slightly simpler than they are for example purposes. Here is the specification of the chunk-types used in the example production:

```
(chunk-type number number next)
(chunk-type count state current)
```

Here is the example production, which is named counting-example:

```
(P counting-example
   "example production for counting in tutorial unit 1 text"
  =goal>
   ISA      count
   state    incrementing
   current  =num1
  =retrieval>
   number   =num1
   next     =num2
==>
  =goal>
   ISA      count
   current  =num2
  +retrieval>
   ISA      number
   number   =num2
)
```

It is not necessary to space the production definition out over multiple lines or indent the components as shown above. It could be written on one line with only a single space between each symbol and result in the creation of the same production. Because of that, the condition of the production is also often referred to as the left-hand side (or LHS) and the action as the right-hand side (or RHS), because of their positions relative to the `==>` separator. However, since adding additional whitespace characters between the symbols does not affect the definition of a production, they are typically written spaced out over several lines to make them easier to read.

The symbols used in the production definition are also not case sensitive. The symbol ISA is only capitalized for emphasis in the example and it could have been written as isa, Isa, or any other combination of capital and lowercase letters with the same result.

1.4.4.1 Production Condition: Buffer Pattern Matching

The condition of the **counting-example** production specifies a pattern to match to the **goal** buffer and a pattern to match to the **retrieval** buffer. A buffer pattern begins with a symbol that starts with the character "=" and ends with the character ">". Between those characters is the name of the buffer to which the pattern is applied. Thus, the symbol =goal> indicates a pattern used to test the chunk in the **goal** buffer and the symbol =retrieval> indicates a pattern to test the chunk in the **retrieval** buffer. For a production's condition to match, the first thing that must be true is that there be a chunk in each of the buffers being tested. Thus, if there is no chunk in either the **goal** or **retrieval** buffer, often referred to as the buffer being empty or cleared, this production cannot match.

After indicating which buffer to test, an optional declaration may be made using the symbol isa and the name of a chunk-type to provide a declaration of the set of slots which are being used in the test. In the example production, the **goal** buffer pattern includes a declaration that the slots being tested are from the count chunk-type, but the **retrieval** buffer pattern does not declare a type for the set of slots specified. It is recommended that one create chunk-types and use the isa declarations when writing productions, but this one has been omitted for demonstration purposes. The important thing to remember is that the isa declaration is not a part of the pattern to be tested – it is only a declaration to allow the ACT-R software to verify that the slots used in the pattern are consistent with the chunk-type indicated.

The remainder of the pattern consists of slot tests for the chunk in the specified buffer. A slot test consists of an optional modifier (which is not used in any of the tests in this example production), the name of a slot for the chunk in the buffer, and a specification of the value that slot of the chunk in the buffer must have. The value may be specific as a constant value, a variable, or the Lisp symbol **nil**.

Here is the **goal** buffer pattern from the example production again for reference:

```
=goal>
   ISA      count
   state    incrementing
   current  =num1
```

The first slot test in the pattern is for a slot named **state** with a constant value of **incrementing**. Therefore for this production to match, the chunk in the **goal** buffer must have a slot named **state** which has the value **incrementing**. The next slot test in the pattern involves the slot named **current** and a variable value.

The "=" prefix on a symbol in a production is used to indicate a variable. The name of the variable can be any symbol, and it is recommended that the variable names be chosen to help make the production easier for a person reading it to understand. Variables are used in a production to generalize the condition and action, and they have two basic purposes. In the condition, variables can be used to compare the values in different slots, for instance that they have the same value or different values, without needing to know all the possible values those slots could have. The other purpose for a variable is to copy a value from a slot specified in the condition to another slot specified in the action of the production.

There are two properties of variables used in productions which are important. Every slot tested with a variable in the condition must exist in the chunk in the buffer for the pattern to match. Also, a variable is only meaningful within a specific production -- using the same variable name in different productions does not create any relation between those productions.

Given that, we could describe this production's test for the **goal** buffer like this:

There must be a chunk in the goal buffer. It must have a slot named state with the value incrementing, and it must have a slot named current whose value we will refer to using the variable =num1.

Now, we will look at the **retrieval** buffer's pattern in detail:

```
   =retrieval>
      number      =num1
      next        =num2
```

The first slot test it has tests the slot named **number** with the variable **=num1**. Since the variable **=num1** was also used in the **goal** buffer test, this is testing that the **number** slot of the chunk in the **retrieval** buffer has the same value as the **current** slot of the chunk in the **goal** buffer. The other slot test for the **retrieval** buffer is for the slot named **next** using a variable named **=num2**.

Therefore, the test for the **retrieval** buffer can be described as:

There must be a chunk in the retrieval buffer. It must have a slot named number which has the same value as the slot named current of the chunk in the goal buffer, and it must have a slot named next whose value we will refer to using the variable =num2.

There is one other detail to note about the buffer pattern tests on the LHS of a production. Notice that the specification of a buffer pattern begins with the character "=" which is also used to indicate a variable in a production. The start of a buffer pattern is also specifying a variable using the name of the buffer. In this production that would be the variables =goal and =retrieval. Those variables will refer to the chunk that is in the **goal** buffer and the chunk that is in the **retrieval** buffer respectively, and those variables can be used just like any other variable to test a value in a slot or to place that chunk into a slot as an action.

This example production did not include all of the possible items which one may need in writing the condition for a production, but it does cover the basics of the pattern matching applied to chunks in buffers. We will describe how one queries the buffer and module state later in this unit, and additional production condition details will be introduced in future units.

1.4.4.2 Production Action

The action of a production consists of a set of operations which affect the buffers. Here is the RHS from the example production again:

```
  =goal>
   ISA      count
   current  =num2
  +retrieval>
   ISA      number
   number   =num2
```

The RHS of a production is specified much like the LHS with actions to perform on particular buffers. To indicate a particular action to perform with a buffer a symbol is used which starts with a character that indicates the action to take, followed by the name of a buffer, and then the character ">". That is followed by an optional chunk-type declaration using isa and a chunk-type name and then the specification of slots and values to detail the action to perform. There are five different operations that can be performed with a buffer. The three most common will be described in this unit. The other operations will be described later in the tutorial.

1.4.4.2.a Buffer Modifications, the = action

If the buffer name is prefixed with the character "=" then the action will cause the production to immediately modify the chunk currently in that buffer. Each slot specified in a buffer modification action indicates a change to make to the chunk in the buffer. If the chunk already has such a slot its value is changed to the one specified. If the chunk does not currently have that slot then that slot is added to the chunk with the value specified.

Here is the action for the **goal** buffer from the example production:

```
=goal>
 ISA      count
 current  =num2
```

It starts with the character "=" therefore this is a modification to the buffer. It will change the value of the **current** slot of the chunk in the **goal** buffer (since we know that it has such a slot because it was tested in the condition of the production) to the value referred to by the variable **=num2** (which is the value that the **next** slot of the chunk in the **retrieval** buffer had in the condition). This is an instance of a variable being used to copy a value from one slot to another.

1.4.4.2.b Buffer Requests, the + action

If the buffer name is prefixed with the character "+", then the action is a request to that buffer's module, and we will often refer to such an action as a "*buffer* request" where *buffer* is the name of the buffer indicated e.g. a retrieval request or a goal request. Typically this results in the module replacing the chunk in the buffer with a different one, but not all modules handle requests the same way. As was noted above, the goal module handles requests by creating new chunks and the declarative module uses the request to find a matching chunk in the model's declarative memory to place into the buffer. In later units of the tutorial we will also describe modules that perform perceptual and motor actions in response to requests.

Here is the retrieval request from the example production:

```
+retrieval>
 ISA         number
 number      =num2
```

It is asking the declarative module to find a chunk that has a slot named **number** that has the same value as **=num2**. If such a chunk exists in the model's declarative memory it will be placed into the **retrieval** buffer.

1.4.4.2.c Buffer Clearing, the - action

A third type of action that can be performed with a buffer is to explicitly clear the chunk from the buffer. This is done by placing the "-" character before the buffer name in the action. Thus, this action on the RHS of a production would clear the chunk from the **retrieval** buffer:

```
  -retrieval>
```

Clearing a buffer occurs immediately and results in the buffer being empty and the declarative module storing the chunk which was in the buffer in the model’s declarative memory.

1.4.4.2.d Implicit Clearing

In addition to the explicit clearing action one can make, there are situations which will implicitly clear a buffer. Any buffer request with a “+” action will also cause that buffer to be cleared. Therefore, the retrieval request from the example production will also result in the **retrieval** buffer being automatically cleared at the time the request is made. Later units will describe other places where implicit buffer clearing will occur.

1.5 The Count Model

The first model we will run is a simple one that counts up from one number to another, for example it will count up from 2 to 4 -- 2,3,4. It is included with the tutorial files for unit 1 in the file named “count.lisp”. You should now start the ACT-R software if you have not done so already, and then load the count model by pressing the “Load ACT-R code” button on the ACT-R Environment Control Panel and selecting the “count.lisp” file.

When you do that, you should see a window open up which says “Successful Load”, with no other text in the window. You should press the “Ok” button on that window to continue. If there had been any problems when loading the file then details about those issues would have been shown in that window.

Now you should run the model by pressing the “Run” button on the ACT-R Environment Control Panel. That will run the model for up to 10 seconds, since the default time shown next to the button is 10.0. When you do that, you should see the following output in the ACT-R window:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL FIRST-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.000   PROCEDURAL             PRODUCTION-SELECTED START
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED START
     0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK TWO
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   PROCEDURAL             PRODUCTION-SELECTED INCREMENT
     0.100   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT
TWO
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK THREE
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   PROCEDURAL             PRODUCTION-SELECTED INCREMENT
     0.200   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.200   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT
THREE
     0.250   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.250   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK FOUR
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FOUR
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   PROCEDURAL             PRODUCTION-SELECTED STOP
     0.300   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.300   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.350   PROCEDURAL             PRODUCTION-FIRED STOP
FOUR
     0.350   PROCEDURAL             CLEAR-BUFFER GOAL
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   ------                 Stopped because no events left to process
```

This output is called the trace of the model. Each line of the trace represents one event that occurred during the running of the model. Each event shows the time in seconds at which it happened, the ACT-R module that generated the event, and some details describing the event. Any output generated by the model is also shown in the trace. The level of detail provided in the trace can be changed to see more or less information as needed, and this model is set to show the most information possible. How to change the amount of detail shown in the trace is described in the unit 1 code description document, named “unit1_code”.

You should now open the count model in a text editor (if you have not already) to begin looking at how the model is specified. Since we will not be editing the file, the simple editor provided by the ACT-R Environment is sufficient to see the model definition, and you can use that by pressing the “Open File” button on the Control Panel.

The first two commands used in the model file are **clear-all** and **define-model** as described above, and the rest of the file contains the model definition within the **define-model** call.

The first item in the model definition is a call to the **sgp** command which is used to set parameters for the model. We will not describe that here, but it is covered in the code description document. The rest of the model definition contains the chunks and productions for performing this task, and we will look at those in detail.

1.5.1 Chunk-types for the Count model

The model definition has two specifications for chunk-types used by this model:

```
(chunk-type number number next)
(chunk-type count-from start end count)
```

The **number** chunk-type specifies the slots that will be used to encode the ordering of numbers. It contains a slot named number which indicates the current number and a slot named next which indicates the next number in order.[1] The **count-from** chunk type specifies the slots that will be used for the **goal** buffer chunk of the model, and it has slots to hold the starting number, the ending number, and the current count.

1.5.2 Declarative Memory for the Count model

After the chunk-types we find the initial chunks placed into the declarative memory of the model using the **add-dm** command:

```
(add-dm
 (one ISA number number one next two)
 (two ISA number number two next three)
 (three ISA number number three next four)
 (four ISA number number four next five)
 (five ISA number number five)
 (first-goal ISA count-from start two end four))
```

Each of the lists in the add-dm command specifies one chunk. The first five define chunks named **one**, **two**, **three**, **four**, and **five**. Each chunk represents a number, and contains a slot indicating the next number in order. This is the knowledge that enables the model to count.

The last chunk created, **first-goal**, encodes the goal of counting from two (in slot **start**) to four (in slot **end**). Note that the chunk-type **count-from** has another slot called **count** which is not used when creating the chunk **first-goal**. Because the **count** slot is not included in the definition of the chunk **first-goal** that chunk does not have a slot named **count**.

1.5.3 Setting the Initial Goal

The next thing we see is a call to the command **goal-focus** with the chunk name first-goal:

```
(goal-focus first-goal)
```

[1] There are many ways which one could represent that information using a single chunk or spread across multiple chunks. We have chosen a single chunk representation of numbers for the tutorial to keep things simple, and we will use that same representation throughout the tutorial (except for the final model used in this unit), extending it when necessary to include more information.

The **goal-focus** command is provided by the goal module to allow the modeler to specify a chunk to place into the **goal** buffer when the model starts to run. Therefore, in this model the chunk named **first-goal** will be placed into the **goal** buffer when the model starts to run, and the results of that command can be seen in the first line of the trace shown above and copied here with color coding for reference:

```
 0.000   GOAL      SET-BUFFER-CHUNK GOAL FIRST-GOAL NIL
```

It shows that at time 0.000 the goal module performed the set-buffer-chunk action (the ACT-R command for placing a chunk into a buffer) for the goal buffer with the chunk named **first-goal**. It also indicates that this action was not requested by a production (the "nil" at the end of the event details). That last part is not an important detail for this model, but we will describe places where that distinction may be important in later units.

1.5.4 The Productions

The rest of the model definition is the productions which can use the chunks from declarative memory to count, and we will look at each of the production specifications in detail along with the selection and firing of those productions as shown in the trace of the model run above.

1.5.4.1 The Start Production

```
(p start
   =goal>
      ISA         count-from
      start       =num1
      count       nil
 ==>
   =goal>
      ISA         count-from
      count       =num1
   +retrieval>
      ISA         number
      number      =num1
   )
```

The LHS of the start production tests the **goal** buffer. It tests that there is a value in the start slot which it now references with the variable **=num1**. This is often referred to as binding the variable, as in, **=num1** is bound to the value that is in the start slot. It also tests the count slot for the value **nil**. The value **nil** is a special value that can be used when testing and setting slots. **Nil** is the Lisp symbol which represents both the boolean false and null (the empty list). Its use in a slot test is to check that the slot does not exist in the chunk. Thus, that is testing that the chunk in the **goal** buffer does not have a slot named count.

The RHS of the start production performs two actions. The first is a modification of the chunk in the **goal** buffer. That modification will add the slot named count to the chunk in the **goal** buffer (since the condition tested that it does not have such a slot) with the value bound to **=num1**. The other action is a retrieval request. That request asks the declarative module to find a chunk that has the value which is bound to **=num1** in its number slot and place that chunk into the **retrieval** buffer.

Looking at the model trace, we see that the first production that gets selected and fired by the model is the production **start**:

```
   0.000   PROCEDURAL             CONFLICT-RESOLUTION
   0.000   PROCEDURAL             PRODUCTION-SELECTED START
   0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
   0.050   PROCEDURAL             PRODUCTION-FIRED START
```

The first line shows that the procedural module is performing an action called conflict-resolution. That action is the process which the procedural module uses to select which of the productions that matches the current state (if any) to fire next. The next line shows that among those that matched, the **start** production was selected. The important thing to remember is that the production was selected because its condition was satisfied. It did not get selected because it was the first production listed in the model or because it has the name start. The third line shows that the selected production's condition tested the chunk in the **goal** buffer. That last line, which happens 50 milliseconds later (time 0.050), shows that the **start** production has now fired and its actions will take effect. The 50ms time is a parameter of the procedural module, and by default every production will take 50ms between the time it is selected and when it fires.

The actions of the production are seen as the next two lines of the trace:

```
   0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
   0.050   PROCEDURAL             MODULE-REQUEST RETRIEVAL
```

Mod-buffer-chunk is the action which modifies a chunk in a buffer, in this case the **goal** buffer, and module-request indicates that a request is being sent to a module through the indicated buffer.

The next line of the trace is also a result of the RHS of the **start** production:

```
   0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
```

That is the implicit clearing of the buffer which happens because of the request that was made.

The next line in the trace is a notification from the declarative module that it has received a request and has started the chunk retrieval process:

```
     0.050   DECLARATIVE            start-retrieval
```

That is followed by the procedural system now trying to find a new production to fire:

```
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
```

However, it is not followed by a notification of a production being selected, because there is no production whose condition is satisfied at this time.

The following two lines show the successful completion of the retrieval request by the declarative module and then the setting of the **retrieval** buffer with that chunk after another 50ms have passed.

```
     0.100   DECLARATIVE            RETRIEVED-CHUNK TWO
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
```

Now we see conflict resolution occurring again and this time the **increment** production is selected.

```
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   PROCEDURAL             PRODUCTION-SELECTED INCREMENT
```

1.5.4.2 The Increment Production

```
 (P increment
   =goal>
      ISA         count-from
      count       =num1
    - end         =num1
   =retrieval>
      ISA         number
      number      =num1
      next        =num2
 ==>
   =goal>
      ISA         count-from
      count       =num2
   +retrieval>
      ISA         number
      number      =num2
   !output!       (=num1)
   )
```

On the LHS of this production we see that it tests both the **goal** and **retrieval** buffers. In the test of the **goal** buffer it uses a modifier in the testing of the **end** slot:

```
   =goal>
      ISA         count-from
      count       =num1
    - end         =num1
```

The “-” in front of the slot is the negative test modifier. It means that the following slot test must **not** be true for the test to be successful. The test is that the **end** slot of the chunk in the **goal** buffer have the value bound to the **=num1** variable (which is the value from the **count** slot of the chunk in the buffer). Thus, this test is true if the **end** slot of the chunk in the **goal** buffer does **not** have the same value as the **count** slot since they are tested with the same variable **=num1**. Note that the negation of the **end** slot test would also be true if the chunk in the **goal** buffer did not have a slot named **end** because a test for a slot value in a slot which does not exist is false, and thus the negation of that would be true.

The **retrieval** buffer test checks that there is a chunk in the **retrieval** buffer which has a value in its **number** slot that matches the current **count** slot from the **goal** buffer chunk and binds the variable **=num2** to the value of its **next** slot:

```
   =retrieval>
      ISA         number
      number      =num1
      next        =num2
```

We can see that these two buffers were tested by the production in the next two lines of the trace:

```
     0.100   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
```

Now we will look at the RHS of this production:

```
  =goal>
      ISA         count-from
      count       =num2
   +retrieval>
      ISA         number
      number      =num2
   !output!       (=num1)
```

The first two actions are very similar to those in the **start** production. It modifies the chunk in the **goal** buffer to change the value of the **count** slot to the next number, which is the value of the **next** slot of the chunk in the **retrieval** buffer, and it makes a retrieval request to get the chunk representing that next number. The third action is a special command that can be used in the actions of a production:

```
   !output!       (=num1)
```

!output! (pronounced bang-output-bang) can be used on the RHS of a production to display information in the trace. It may be followed by a single item or a list of items, and the given item(s) will be printed in the trace when the production fires. In this production it is used to display the numbers as the model counts. The results of the **!output!** in the first firing of **increment** can be seen in the next two lines of the trace:

```
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT
TWO
```

The output is displayed on one line in the trace after the notice that the production has fired. The items in the list to output can be variables as is the case here (=num1), constant items like (stopping), or a combination of the two e.g. (the number is =num). When a variable is included in the items to output the value to which it was bound in the production will be used in the output which is displayed. That is why the trace shows TWO instead of =num1.

The next few lines of the trace show the actions initiated by the **increment** production and they look very much like the actions that the **start** production generated. The **goal** buffer is modified, a retrieval request is made, a chunk is retrieved, and then that chunk is placed into the **retrieval** buffer:

```
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK D
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL D
```

We then see that the **increment** production is selected again:

```
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   PROCEDURAL             PRODUCTION-SELECTED INCREMENT
```

It will continue to be selected and fired until the value of the **count** and **end** slots of the goal chunk are the same, at which time its test of the **goal** buffer will fail. We see that it fires twice in the trace and then a different production is selected at .25 seconds into the run:

```
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-SELECTED STOP
```

1.5.4.3 The Stop Production

```
(P stop
   =goal>
      ISA         count-from
      count       =num
      end         =num
   =retrieval>
      ISA         number
      number      =num
 ==>
```

```
   -goal>
   !output!       (=num)
   )
```

The stop production matches when the values of the **count** and **end** slots of the chunk in the **goal** buffer are the same and they also match the value in the **number** slot of the chunk in the **retrieval** buffer. The action it takes is to again print out the current number and to also clear the chunk from the **goal** buffer:

```
     0.350   PROCEDURAL             PRODUCTION-FIRED STOP
FOUR
     0.350   PROCEDURAL             CLEAR-BUFFER GOAL
```

The final event that happens in the run is conflict-resolution by the procedural module. No productions are found to match and no other events of any module are pending at this time (for instance a retrieval request being completed) so there is nothing more for the model to do and it stops running.

```
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   ------                 Stopped because no events left to process
```

1.6 Pattern Matching Exercise

To show you how ACT-R goes about matching a production, you will work through an exercise where you will manually fill in the bindings for the variables of the productions as they are selected. This is called instantiating the production. You need to do the following:

1. Load the count model if it is not currently loaded or reset it to its initial conditions if it has been loaded and already run. To reset the model you should press the "Reset" button on the Control Panel.

2. Press the "Stepper" button in the Control Panel to open the Stepper tool. This tool will pause the model before every operation it performs. For each event that shows as a line in the trace, the stepper will force the model to wait for your confirmation before handling that event. That provides you with the opportunity to inspect all of the components of the model as it progresses and can be a very valuable tool for debugging models. After each time a production is selected the Stepper will show the text of the production, the bindings for the variables as they matched that production, and a set of parameters for that production (which we will describe later in the tutorial). After the production is fired it will show the same information except that the production text will have the variables replaced with their bound values (referred to as being instantiated). After a retrieval request completes, the Stepper will show the details of the chunk it retrieved and the corresponding parameters that lead to that chunk being the one retrieved (more on that in a later unit).

You are going to use a feature of this window that forces you to manually instantiate the productions before it will step past a production-selected event. That is, you will assign all of the variables the proper values when the production is selected. To enable this functionality of the Stepper, click the “Tutor Mode” checkbox at the top of the Stepper window.

3. Press the “Buffers” button in the Control Panel to bring up a new buffer inspection window. That will display a list of all the buffers for the existing modules. Selecting one from the list will display the chunk that is in that buffer in the text window to the right of the list. At this point all of the buffers are empty, but that will change as the model runs.

4. Now you should run the model by pressing the “Run” button in the Control Panel. The first action that occurs is the setting of the chunk in the **goal** buffer as was seen in the first line of the trace above:

```
0.000   GOAL    SET-BUFFER-CHUNK GOAL FIRST-GOAL NIL
```

That line should be displayed at the top of the Stepper window where it says “Next Step:” indicating the next action which will occur. Hitting the “Step” button in the Stepper window will allow that action to occur, and you will see that line printed out in the model trace and now shown as the “Last Stepped:” event in the Stepper window. You can now inspect the chunk in the **goal** buffer using the buffer inspection window which you opened previously. When using the Stepper, the inspection windows will automatically update their contents when a step is taken, but when the Stepper is not being used the windows will only update when they are selected which allows one to compare values from different times in a model’s run and improves the performance of the system since it does not have to update the information as it runs.

1.6.1 The first-goal-0 Chunk and Buffer Chunk Copying

When you inspect the **goal** buffer you will see output that looks like this:

```
GOAL: FIRST-GOAL-0 [FIRST-GOAL]
FIRST-GOAL-0
   START  TWO
   END  FOUR
```

The goal-focus command in the model made the goal module set the **goal** buffer to the chunk **first-goal**. The trace indicated that the chunk **first-goal** was used to set the buffer, but this output is displaying that a chunk named **first-goal-0** is currently in the **goal** buffer. Why is that? When a chunk is placed into a buffer the buffer always makes its own new copy of that chunk which is then placed into the buffer. The name of the chunk that was copied is shown in the buffer inspection window in square brackets after the name of the chunk actually in the buffer. The reason for copying the chunk is to prevent the model from being able to alter the original chunk -- it cannot directly manipulate the information that it has learned previously, but it may use the contents of that knowledge to create new chunks.

The buffers can be thought of as note pads for manipulating and creating new chunks. Placing a chunk into a buffer corresponds to writing the contents of that chunk on the top page of the note pad. The information on the top page of the note pad can be manipulated by productions or any of the modules. Clearing the chunk from the buffer corresponds to tearing off the page with the current information and revealing a new empty page underneath. That torn off page containing the chunk is then filed away in the model's declarative memory and can no longer be changed, but it can be retrieved and copied into the retrieval buffer for use in the future.

1.6.2 Pattern Matching Exercise Continued

5. If you Step past the production-selected event you will come to the first production to match, **start**. It is shown in the bottom right pane of the Stepper window with all of the variables highlighted. Your task is to replace all of the variables with the values to which they are bound in the matching of this production. When you click on a highlighted variable in the production display of the stepper a new window will open in which you can enter the value for that variable. You must enter the value for every variable in the production (including multiple occurrences of the same variable) before it will allow you to progress to the next event.

Here is how you can determine the variable bindings:

- **=goal** will always bind to the name of the chunk in the **goal** buffer. This can be found with the Buffers viewer.
- **=retrieval** will always bind to the name of the chunk in the **retrieval** buffer. This can also be found with the Buffers viewer.
- To find the binding for other variables, you will use the Buffers viewer to look at the buffer which is being tested to find the value for the slot that is being tested with the variable.
- A variable will have the same value everywhere in a production that matches, and the bound values are displayed in the Stepper window as you enter them. Thus once you find the binding for a variable you can just refer to the Stepper to get the value for other occurrences of the variable in that production.

At any point in time, you can ask the tutor for help in binding a variable by hitting either the Hint or Help button of the entry dialog. A hint will instruct you on where to find the correct answer and help will just give you the correct answer.

6. Once the production is completely instantiated, you can continue stepping through the model run with the Step button. You should step the model to the next production that matches and watch how the contents of the **goal** and **retrieval** buffer change based on the actions taken by the **start** production.

7. To finish the run of the model you will need to instantiate two instances of the production i**ncrement** being selected and one instance of the **stop** production matching.

8. When you have completed this example and explored it as much as you want, go on to the next section of this unit, in which we will describe another example model.

1.7 The Addition Model

The second example model uses a larger set of counting facts to do a somewhat more complicated task. It will perform addition by counting up. Thus, given the goal to add 2 to 5 it will count 5, 6, 7, and report the answer 7. You should now load the **addition** model, found in the "addition.lisp" file of unit 1, the same way you loaded the **count** model.

The initial chunks for the **addition** model are the same as those used for the **count** model with the inclusion of a chunk that encodes one follows zero and chunks that encode counting from seven up to ten. The chunk-type created for the goal information now has slots to hold the starting number (**arg1**), the number to be added (**arg2**), a slot for holding the currently counted value (**sum**), and a slot for holding the amount that has been added so far (**count**):

```
(chunk-type add arg1 arg2 sum count)
```

Here is the initial chunk created for the **goal** buffer which indicates that it should add 2 to 5:

```
(second-goal ISA add arg1 five arg2 two)
```

If you run this model (without having the Stepper open) you will see this trace:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL SECOND-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FIVE
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FIVE
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK ZERO
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK SIX
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SIX
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            start-retrieval
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK ONE
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   DECLARATIVE            start-retrieval
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   DECLARATIVE            RETRIEVED-CHUNK SEVEN
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SEVEN
     0.500   PROCEDURAL             CONFLICT-RESOLUTION
     0.550   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
SEVEN
     0.550   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.550   PROCEDURAL             CONFLICT-RESOLUTION
     0.550   ------                 Stopped because no events left to process
```

The first thing you may notice is that there is less information in the trace of this model than there was in the trace of the **count** model. This model is set to show the default trace detail level which should make it easier to read the sequence of productions that fire.

In this sequence we see that the model alternates between incrementing the count from zero to two and incrementing the sum from five to seven. The production **initialize–addition** starts things going and makes a retrieval request for the chunk which has the number found in arg1. **Increment-sum** then uses that retrieved chunk to update the current sum to the next number and requests a retrieval of a chunk to determine the next count value. That production fires alternately with **increment-count**, which processes the retrieval of the counter increment and requests a retrieval to use to increment the sum. **Terminate-addition** recognizes when the counter equals the second argument of the addition and it modifies the **goal** to make the model stop and outputs the answer.

1.7.1 The initialize-addition and terminate-addition productions

The production **initialize-addition** initializes an addition process whereby the model tries to count up from the first digit a number of times that equals the second digit and the production **terminate-addition** recognizes when this has been completed.

```
(P initialize-addition
   =goal>
      ISA         add
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal>
      ISA         add
      sum         =num1
      count       zero
   +retrieval>
      ISA         number
      number      =num1
)
```

When the chunk in the **goal** buffer has values for the **arg1** and **arg2** slots but does not have a **sum** slot the **initialize-addition** production matches. Its action modifies the **goal** buffer chunk to add a **sum** slot with the first digit and a **count** set to zero, and it makes a retrieval request for a chunk that has a value in the number slot matching **=num1**.

Pairs of productions will apply after this to keep incrementing the **sum** and the **count** slots until the **count** slot equals the **arg2** slot, at which time **terminate-addition** matches:

```
(P terminate-addition
   =goal>
      ISA         add
      count       =num
      arg2        =num
      sum         =answer
   =retrieval>
      ISA         number
      number      =answer
  ==>
   =goal>
      ISA         add
      count       nil
   !output!       =answer
)
```

This production removes the **count** slot from the **goal** buffer chunk by setting it to the symbol **nil**. Previously we saw the special symbol **nil** used to test that a slot does not exist in a chunk, and here we see the reciprocal modification for removing a slot. This causes the model to stop because other than **initialize-addition**, (which requires that the sum slot not exist) all of the other productions require the goal to have a **count** slot. So, after this production fires none of the productions will match the chunk in the **goal** buffer.

1.7.2 The increment-sum and increment-count productions

The two productions that apply repeatedly between the previous two are **increment-sum**, which uses the retrieval of the sum increment and requests a retrieval of the count increment, and **increment-count**, which uses the retrieval of the count increment and requests a retrieval of the sum increment.

```
(P increment-sum
   =goal>
      ISA         add
      sum         =sum
      count       =count
    - arg2        =count
   =retrieval>
      ISA         number
      number      =sum
      next        =newsum
```

```
  ==>
   =goal>
      ISA         add
      sum         =newsum
   +retrieval>
      ISA         number
      number      =count
)

(P increment-count
   =goal>
      ISA         add
      sum         =sum
      count       =count
   =retrieval>
      ISA         number
      number      =count
      next        =newcount
  ==>
   =goal>
      ISA         add
      count       =newcount
   +retrieval>
      ISA         number
      number      =sum
)
```

Some terminology which is often used when discussing productions which use a chunk that was placed into a buffer and then immediately clear that chunk from the buffer is to say that it "harvests" the chunk. Both of these productions harvest the chunk from the **retrieval** buffer which resulted from the request of the other because the retrieval request action automatically clears the buffer. We would not say that they harvest the **goal** buffer's chunk because it is modified and remains in the buffer.

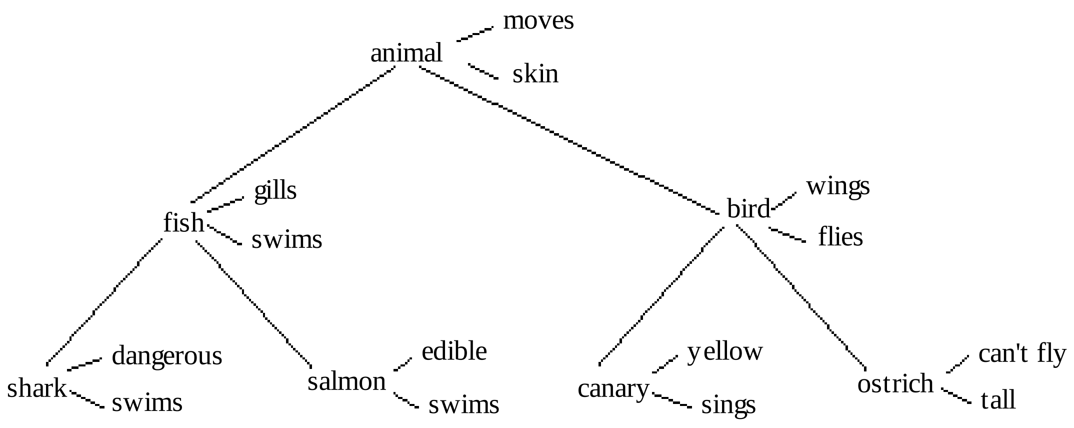
1.7.3 The Addition Exercise

Now, as you did with the **count** model, you should use the tutor mode of the Stepper to step through the matching of the four productions for the **addition** model. Once you have completed that you should move on to the next model.

1.8 The Semantic Model

The last example model for this unit is the **semantic** model, found in the "semantic.lisp" file of tutorial unit 1. You should load that model file now. When you do so, you may notice that there is some text displayed in the window that indicates it loaded successfully and that text is also displayed in the ACT-R window. For now, you can ignore that output and just press "Ok" as you have for the previous models. We will come back to that after describing the operation of the model.

This model contains chunks which encode the following network of categories and properties.



The productions it has are capable of searching this network to make decisions about whether one category is a member of another category.

1.8.1 Encoding of the Semantic Network

All of the links in this network are encoded by chunks with the slots **object**, **attribute**, and **value**. For instance, the following three chunks encode the links involving shark:

```
(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)
```

p1 encodes that a shark is dangerous by encoding the value as **true** for an attribute with the value **dangerous**. **p2** encodes that a shark can swim by encoding the value **swimming** with the attribute of **locomotion**. **p3** encodes that a shark is a fish by encoding **fish** as the value and the attribute as **category**.

There are of course many other ways one could encode this information, for example instead of using slots named attribute and value it seems like one could just use the slot as the attribute and the value as its value for example:

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

or perhaps even collapsing all of that information into a single chunk describing a shark:

```
(shark dangerous true locomotion swimming category fish)
```

How one chooses to organize the knowledge in a model can have an effect on the results, and that can be a very important aspect of the modeling task. For example, choosing to encode the properties in separate chunks vs a single chunk will affect how that information is reinforced (all the items together vs each individually) as well as how it may be affected by the spreading of activation for related information (both topics which will be discussed in later units). Typically, there is no one “right way” to write a model and one should make the choices necessary based on the objectives of the modeling effort and any related research which provides guidance.

For this model we have chosen the representation for practical reasons – it provides a useful example. It might seem that the second representation would still be better for that, and such a representation could have been used for the category searching model described below since we are only searching for the category attribute which could have been encoded directly into the productions. However, with what has been discussed so far in the tutorial, it would be difficult to use that representation to perform a more general search for information in the network. When we get to unit 8 of the tutorial we will see some more advanced aspects of the pattern matching allowed in productions which could be used with such a representation to make the searching more general.

1.8.2 Testing for Category Membership

This model performs a test for category membership when a chunk in the **goal** buffer has object and category slot values and no slot named judgment. There are 3 different starting goals provided in the initial chunks for the model. The one initially placed in the **goal** buffer is **g1**:

```
(g1 ISA is-member object canary category bird)
```

which represents a test to determine if a canary is a bird. That chunk does not have a judgment slot, and the model will add that slot to indicate a result of yes or no. If you run the model with **g1** in the **goal** buffer (which is placed there by the call to goal-focus in the model definition) you will see the following trace:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL G1 NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK P14
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED DIRECT-VERIFY
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   ------                 Stopped because no events left to process
```

If you inspect the goal chunk after the model stops it will look like this:

```
GOAL: G1-0
G1-0
   OBJECT  CANARY
   CATEGORY  BIRD
   JUDGMENT  YES
```

This is among the simplest cases possible for this network of facts and only requires the retrieval of this property to determine the answer:

```
(p14 ISA property object canary attribute category value bird)
```

There are two productions involved. The first, **initial-retrieve**, requests the retrieval of categorical information and the second, **direct-verify,** harvests that information and sets the **judgment** slot to **yes**:

```
(p initial-retrieve
   =goal>
      ISA         is-member
      object      =obj
      category    =cat
      judgment    nil
==>
   =goal>
      judgment    pending
   +retrieval>
      ISA         property
      object      =obj
      attribute   category
)
```

Initial-retrieve tests that there are object and category slots in the **goal** buffer's chunk and that it does not have a judgment slot. Its action is to modify the chunk in the **goal** buffer by adding a judgment slot with the value pending and to request the retrieval of a chunk from declarative memory which indicates a category for the current object.

After that production fires and the chunk named p14 is retrieved the **direct-verify** production matches and fires:

```
(P direct-verify
   =goal>
      ISA         is-member
      object      =obj
      category    =cat
      judgment    pending
   =retrieval>
      ISA         property
      object      =obj
      attribute   category
      value       =cat
==>
   =goal>
      judgment    yes
)
```

That production tests that the chunk in the **goal** buffer has values in the object and category slots and a judgment slot with the value pending, and that the **retrieval** buffer has an object slot with the same value as the object slot of the chunk in the **goal** buffer, an attribute slot with the value category, and a value slot with the same value as the category slot of the chunk in the **goal** buffer. Its action is to modify the chunk in the **goal** buffer by setting the judgment slot to the value yes.

You should now work through the pattern matching process in the tutor mode of the Stepper with the goal set to **g1**.

1.8.3 Chaining Through Category Links

A slightly more complex case occurs when the category is not an immediate super ordinate of the indicated object and it is necessary to chain through an intermediate category. An example where this is necessary is in the verification of whether a canary is an animal, and such a test is created in chunk **g2**:

```
(g2 ISA is-member object canary category animal)
```

You should change the goal-focus call in the model file to set the goal to **g2** instead of **g1**. After you save that change to the file you must load the model again. That can be done using the "Reload" button on the Control Panel which will load the last model file that was loaded, or you can use the "Load ACT-R code" button to select the file and load it. [Note: it is also possible to update the **goal** buffer chunk without editing the model file and loading it again, and the unit1_code text shows how that could be done.]

Running the model with chunk **g2** as the goal will result in the following trace:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL G2 NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK P14
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK P20
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P20
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED DIRECT-VERIFY
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   ------                 Stopped because no events left to process
```

This trace is similar to the previous one except that it involves an extra production, **chain-category**, which retrieves the category in the case that an attribute has been retrieved which does not immediately allow a decision to be made.

```
(P chain-category
   =goal>
      ISA         is-member
      object      =obj1
      category    =cat
      judgment    pending
   =retrieval>
      ISA         property
      object      =obj1
      attribute   category
      value       =obj2
    - value       =cat
==>
   =goal>
      object      =obj2
   +retrieval>
      ISA         property
      object      =obj2
      attribute   category
)
```

This production tests that the chunk in the **goal** buffer has values in the object and category slots and a judgment slot with the value of pending, and that the chunk in the **retrieval** buffer has an object slot with the same value as the object slot of the chunk in the **goal** buffer, an attribute slot with the value category, a value in the value slot, and that the value in the value slot is not the same as the value of the category slot of the chunk in the **goal** buffer. Its action is to modify the chunk in the **goal** buffer by setting the object slot to the value from the value slot of the chunk in the **retrieval** buffer and to request the retrieval of a chunk from declarative memory which has that same value in its object slot and the value category in its attribute slot.

You should now go through the pattern matching exercise in the tutor mode of the Stepper with **g2** set as the goal.

1.8.4 The Failure Case

Now change the initial goal to the chunk **g3** and reload the model file.

```
(g3 ISA is-member object canary category fish)
```

If you run the model with this goal, you will see what happens when the chain reaches a dead end:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL G3 NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK P14
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK P20
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P20
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVAL-FAILURE
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   PROCEDURAL             PRODUCTION-FIRED FAIL
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   ------                 Stopped because no events left to process
```

Here we see the declarative memory module reporting that a retrieval-failure occurred at time 0.3 which is then followed by the production **fail** firing. The **fail** production uses a test on the LHS that we have not yet seen.

1.8.5 A Query in the Condition

In addition to testing the chunks in the buffers as has been done in all of the productions we have seen to this point, it is also possible to query the status of the buffer itself and the module which controls it. This is done using a "**?**" instead of an "**=**" before the name of the buffer. There is a fixed set of queries which can be made of the buffer itself. The buffer's module must respond to some specific queries, but may have any number of additional queries for its specific operations to which it will respond. The result of a query will be either true or false. If any query tested in a production has a result which is false, then the production does not match.

1.8.5.1 Querying the buffer itself

When querying the buffer itself, it can be in one of three mutually exclusive situations: there can be a chunk in the buffer, a failure can be indicated, or the buffer can be empty with no failure noted. If there is a chunk in the buffer or a failure has been indicated, then it is also possible to test whether that was the result of a requested action or not. Here are examples of the possible queries which can be made to test the status of a buffer, using the **retrieval** buffer for the examples.

This query will be true if there is a chunk in the **retrieval** buffer and false if there is not:

```
 ?retrieval>
     buffer      full
```

This query will be true if a failure has been noted for the **retrieval** buffer and false if not:

```
 ?retrieval>
     buffer      failure
```

This query will be true if there is not a chunk in the **retrieval** buffer and there is not a failure indicated and false if there is either a chunk in the buffer or a failure has been indicated:

```
  ?retrieval>
      buffer     empty
```

This query will be true if there is either a chunk in the **retrieval** buffer or the failure condition has occurred for the **retrieval** buffer, and that chunk or failure was the result of a request made to the buffer’s module. Otherwise, it will be false:

```
  ?retrieval>
      buffer     requested
```

This query will be true if there is either a chunk in the **retrieval** buffer or the failure condition has occurred for the **retrieval** buffer, and that chunk or failure happened without a request being made to the buffer’s module (later units will describe modules which can act without being requested to do so). Otherwise, it will be false:

```
  ?retrieval>
      buffer     unrequested
```

1.8.5.2 Querying a buffer’s module

The queries that are available for every module allow one to test for one of three possible states of the module: free, busy, or error. Below are examples of those queries again using the **retrieval** buffer.

This query will be true if the **retrieval** buffer’s module (the declarative module) is not currently performing an action and will be false if it is currently performing an action:

```
?retrieval>
   state       free
```

This query will be true if the **retrieval** buffer's module is currently performing an action and will be false if it is not currently performing an action:

```
?retrieval>
   state       busy
```

This query will be true if an error has occurred in the declarative module since the last request made to the **retrieval** buffer and false if there has been no error:

```
?retrieval>
   state       error
```

The module's state error query is related to the buffer failure query because when the module encounters an error it will note that as a failure in the buffer, but they may not always both be true. That is because the buffer's failure state is cleared whenever the buffer is cleared, either explicitly with a buffer clear action or implicitly when a request is made to the buffer. However, the module's error state will remain set until the module clears it, and the conditions for clearing the module's error will vary depending on the operation of the particular module. Testing the buffer failure is usually a better choice because of the consistency of its operation across buffers and will be used for the models in the tutorial, but in some circumstances one may find it useful to query a module's internal error state instead.

Unlike the buffer conditions of empty, full, and failure, a module's states of free, busy, and error are not mutually exclusive e.g. a module can be both free and indicate that an error occurred.

Modules can also provide other queries that are specific to their operation. The declarative module for example provides a query called recently-retrieved which can be tested to determine if the chunk that was retrieved had also been retrieved previously:

```
?retrieval>
   recently-retrieved  t
```

1.8.5.3 Using queries in productions

When specifying queries in a production multiple queries can be made of a single buffer. This query checks if the **retrieval** buffer is currently empty and that the declarative module is not currently handling a request:

```
?retrieval>
```

```
      buffer      empty
      state       free
```

One can also use the optional negation modifier “-” before a query to test that such a condition is not true. Thus, either of these tests would be true if the declarative module was not currently handling a request:

```
   ?retrieval>
      state       free
```

or

```
   ?retrieval>
     - state      busy
```

1.8.6 The fail production

Here is the production that fires in response to a category request not being found.

```
(P fail
   =goal>
      ISA         is-member
      object      =obj1
      category    =cat
      judgment    pending

   ?retrieval>
      buffer      failure
==>
   =goal>
      judgment    no
)
```

Note the query for a **retrieval** buffer failure in the condition of this production. When a retrieval request does not succeed, in this case because there is no chunk in declarative memory which matches the specification requested, the buffer will indicate that as a failure. In this model, this will happen when one gets to the top of a category hierarchy and there are no super ordinate categories.

You should now make sure your goal is set to **g3**, and then go through the pattern matching exercise using the tutor mode of the Stepper tool one final time.

1.8.7 Model Warnings

Now that you have worked through the examples we will examine another detail which you might have noticed while working on this model. When you load or reload the **semantic** model there are the following warnings displayed in the Successful load window and the ACT-R window:

```
#|Warning: Creating chunk CATEGORY with no slots |#
#|Warning: Creating chunk PENDING with no slots |#
#|Warning: Creating chunk YES with no slots |#
#|Warning: Creating chunk NO with no slots |#
```

Output that begins with ”#|Warning:” is a warning from ACT-R, and indicates that there is something in the model that may need to be addressed. This differs from warnings or errors which may be reported by the Lisp which implements the ACT-R system that can occur because of problems in the syntax or structure of the code in the model file. If you see ACT-R warnings when loading a model you should always read through them to make sure that there is not a serious problem in the model.

In this case, the warnings are telling you that the model uses chunks named **category, pending, yes,** and **no,** but does not explicitly define them and thus they are being created automatically. That is fine in this case. Those chunks are being used as explicit markers in the productions and there are no problems caused by allowing the system to create them automatically because they are arbitrary names that were used when writing the productions.

If there had been a typo in one of the productions however, for instance misspelling pending in one of them as “pneding”, the warnings may have shown something like this:

```
#|Warning: Creating chunk PNEDING with no slots |#
```

That would provide you with an opportunity to notice the discrepancy and fix the problem before running the model and then trying to determine why it did not perform as expected.

There are many other ACT-R warnings that may be displayed and you should always read the warnings which occur when loading a model to make sure that there are no serious problems before you start running the model. Most warnings which are not a serious problem can be eliminated with some additional declarations or changes to the model. For example, if we wanted to eliminate these warnings we could explicitly create those chunks in the model. With what we have seen so far that would be done by adding them to declarative memory with the other chunks that are created, but later in the tutorial we will show a better approach that does not require adding them to the model’s memory.

1.9 Building a Model

We would like you to now construct the pieces of an ACT-R model on your own. The “tutor-model.lisp” file included with unit 1 contains the basic code necessary for a model, but does not have any of the declarative or procedural elements defined. The instructions that follow will guide you through the creation of those components. You will be constructing a model that can perform the addition of two two-digit numbers using this process to add the numbers (which is of course not the only way one could implement the addition process):

> To add two two-digit numbers start by adding the ones digits of the two numbers. After adding the ones digits of the numbers determine if there is a carry by checking if that result is equal to 10 plus some number. If it is, then that number is the answer for the ones column and there is a carry of 1 for the tens column. If it is not, then that result is the answer for the ones column and there is no carry. Then add the digits in the tens column. If there is no carry from the ones column then that sum is the answer for the tens column and the task is done. If there is a carry from the ones column then add the carry to that sum and make that the answer for the tens column to finish the task.

Once all of the pieces have been created as described below to implement that process, you should be able to load and run the model to produce a trace that looks like this:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED PROCESS-CARRY
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK FACT34
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT34
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-CARRY
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            start-retrieval
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK FACT17
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT17
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-DONE
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   ------                 Stopped because no events left to process
```

There is a working solution model included with the unit 1 files, and it is also described in the code description text for this unit. Thus, if you have problems you can consult that for help, but you should try to complete these tasks without looking first.

You should now open the tutor-model.lisp file in a text editor if you have not already. The following sections will describe the components that you should add to the file in the places indicated by comments in the model (the lines that begin with the semicolons) to construct a model for performing this task. There are of course many ways one could represent the addition facts and process them using productions to perform the addition, but the description below corresponds to the solution model which is provided for reference and thus should be followed explicitly if you want to be able to compare to that as you go along.

1.9.1 Chunk-types

The first thing we should do is define the chunk-types that we will use in writing the model. There are two chunk-types which we will define for doing this task. One to represent addition facts and one to represent the goal chunk which holds the components of the task for the model and the correct answer when it is done. These chunk types will be created with the **chunk-type** command as described in section 1.4.2.

1.9.1.1 Addition Facts

The first chunk-type you should create is one to represent the addition facts. It should be named **addition-fact** and have slots named **addend1**, **addend2,** and **sum**. Chunks with these slots will represent the basic addition facts for addends from 0-10.

1.9.1.2 The Goal Chunk Type

The other chunk type you should create is one to represent the goal of adding two two-digit numbers. It should be named **add-pair.** It will have slots to encode all of the necessary components of the task. It should have two slots to represent the ones digit and the tens digit of the first number called **one1** and **ten1** respectively. It will have two more slots to hold the ones digit and the tens digit of the second number called **one2** and **ten2,** and two slots to hold the answer, called **one-ans** and **ten-ans**. It will also need a slot to hold any information necessary to process a carry from the addition in the ones column to the tens column which should be called **carry**.

1.9.2 Chunks

We are now going to define the chunks that will allow the model to solve the problem 36 + 47 and place them into the model's declarative memory. This is done using the **add-dm** command which was described in section 1.4.3.

1.9.2.1 The Addition Facts

You need to create addition facts which encode the following math facts in the model's declarative memory to be able to solve this problem with the productions which will be described later:

```
3+4=7
6+7=13
```

```
10+3=13
1+7=8
```

They will use the slots of the type addition-fact and should be named based on their addends. For example, the fact that 3+4=7 should be named **fact34**. The addends and sums for these facts will be the corresponding numbers. Thus, **fact34** will have slots with the values 3, 4, and 7. Note that for simplicity we are just using the actual numbers as the values of the slots instead of creating separate number chunks like the addition and count models used.

1.9.2.2 The Initial Goal

You should now create a chunk named **goal** which encodes that the goal is to add 36+47 which will use slots from the add-pair chunk-type. This should be done by specifying the values for the ones and tens digits of the two numbers and leaving all of the other slots empty.

1.9.2.3 Checking the results so far

Once you have completed adding the chunk-types and chunks to the model you should be able to save the file and then load it to inspect the components you have created. To see the chunks you have created you can press the "Declarative" button on the Control Panel. That will open a declarative memory viewer which allows you to see all of the chunks in the model's declarative memory (every time you press that button a new declarative viewer window will be opened so that you can view different chunks at the same time when needed). To view a particular chunk with that tool select it in the list of chunks on the left of the window. The window on the right will then show the declarative parameters for that chunk (which will be described in later units) along with the slots and values for that chunk.

Once you are satisfied with the chunks that you have created you can move on to creating the productions which will use those chunks.

1.9.3 Productions

So far, we have been looking mainly at the individual productions in the models. However, a model really only functions because of the interaction of the productions it contains. Essentially, the action of one production will set the state so that the condition for another production can match, which has an action that sets the state to allow another to match, and so on. It is that sequence of productions firing, each of which performs some small step, which leads to performing the entire task, and one of the challenges of cognitive modeling is determining how to perform those small steps in a way that is both capable of performing the task and consistent with human performance on that task – just "doing the task" is not usually the objective for creating a model with a cognitive architecture like ACT-R.

Your task now is to write the ACT-R productions which perform the steps described in English above to do multi-column addition. To do that you will need to use the ACT-R **p** command to specify the productions as described in section 1.4.4, and to help with that we will further detail six productions which will implement that process using the chunk-types and chunks created previously.

START-PAIR

If the goal buffer contains slots holding the ones digits of two numbers and does not have a slot for holding the ones digit of the answer then modify the goal to add the slot one-ans and set it to a value of **busy** and make a retrieval request for the chunk indicating the sum of those ones digits.

ADD-ONES

If the chunk in the goal buffer indicates that it is **busy** waiting for the answer for the addition of the ones digits and a chunk has been retrieved containing the sum of the ones digits then modify the goal chunk to set the one-ans slot to that sum and add a slot named carry with a value **busy**, and make a retrieval request for an addition fact to determine if that sum is equal to 10 plus some number.

PROCESS-CARRY

If the chunk in the goal buffer indicates that it is **busy** processing a carry, a chunk has been retrieved which indicates that 10 plus a number equals the value in the one-ans slot, and the digits for the tens column of the two numbers are available in the goal chunk then set the carry slot of the goal to 1, set the one-ans slot of the goal to the other addend from the chunk in the retrieval buffer, set the ten-ans slot to the value **busy**, and make a retrieval request for an addition fact of the sum of the tens column digits.

NO-CARRY

If the chunk in the goal buffer indicates that it is **busy** processing a carry, a failure to retrieve a chunk occurred, and the digits for the tens column of the two numbers are available in the goal chunk then remove the carry slot from the goal by setting it to **nil**, set the ten-ans slot to the value **busy**, and make a retrieval request for an addition fact of the sum of the tens column digits.

ADD-TENS-DONE

If the chunk in the goal buffer indicates that it is **busy** computing the sum of the tens digits, there is no carry slot in the goal chunk, and there is a chunk in the retrieval buffer with a value in the sum slot then set the ten-ans slot of the chunk in the goal buffer to that sum.

ADD-TENS-CARRY

If the chunk in the goal buffer indicates that it is **busy** computing the sum of the tens digits, the carry slot of the goal chunk has the value 1, and there is a chunk in the retrieval buffer with a value in the sum slot then remove the carry slot from the chunk in the goal buffer, and make a retrieval request for the addition of 1 plus the current sum from the retrieval buffer.

1.9.4 Running the model

When you are finished entering the productions, save your model, reload it, and then run it.

If your model is correct, then it should produce a trace that looks like the one shown above, and the correct answer should be encoded in the **ten-ans** and **one-ans** slots of the chunk in the **goal** buffer:

```
GOAL-0
   ONE1  6
   TEN1  3
   ONE2  7
   TEN2  4
   TEN-ANS  8
   ONE-ANS  3
```

1.9.5 Incremental Creation of Productions

It is also possible to write just one or two productions and test them out first before you go on to try to write the rest – to make sure that you are on the right track. For instance, this is the trace you would get after successfully writing the first two productions and then running the model:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   ------                 Stopped because no events left to process
```

The first production, **start-pair**, has fired and successfully requested the retrieval of the addition fact **fact67**. The next production, **add-ones**, then fires and makes a retrieval request to determine if there is a carry. That chunk is retrieved and then because there are no productions which match the current state of the system and no actions remaining to perform it stops. You may find it helpful to try out the productions occasionally as you write them to make sure that the model is working as you progress instead of writing all the productions and then trying to debug them all at once.

1.9.6 Debugging the Productions

In the event that your model does not run correctly you will need to determine why that is so you can fix it. One tool that can help with that is the “Why not?” button in the Procedural viewer. To use that, first press the “Procedural” button on the Control Panel to open a viewer for the productions in the model. That tool is very similar to the Declarative viewer described previously – there is a list of productions on the left and selecting one will show the parameters and text of the production on the right. Pressing the "Why not?" button at the top of the Procedural viewer window when there is a production selected in the list will open another window. If the chosen production matches the current state of the buffers then the new window will indicate that it matches and display the instantiation of that production. If the chosen production does not match the current state then the text of the production will be printed and an indication of the first mismatching item from the LHS of the production will be indicated, for example, when the start-pair production described above does not match it might say "The chunk in the GOAL buffer has the slot ONE-ANS" since that production specifies that the **goal** buffer should not have a slot named one-ans.

The Stepper is also an important tool for use when debugging a model because while the model is stopped you can inspect everything in the model using all of the other tools. For more information on writing and debugging ACT-R models you should also read the "Modeling" text which accompanies this unit. That file is the "unit1_modeling" document, and each of the odd numbered units in the tutorial will include a modeling text with details on potential issues one may encounter and ways to detect and fix those issues.

Unit 1 Code Description

This document (and the corresponding documents in future units) will describe any ACT-R commands used in the unit's models which are not described in the main unit text, the code that implements the task/experiment for the models in that unit, how ACT-R is interfaced with the task/experiment, and describe additional commands that one can use interactively when working with ACT-R (most of the information available through the tools of the Environment can also be obtained directly from commands).

For this unit there are no tasks for the models and all of them run simply by using the ACT-R run command. They start with a predefined goal that is specifically placed into the goal buffer, they have a given set of declarative memories, and the result is a modification of the goal chunk representing the processing that took place. Later units will include tasks which have much more involved perception and action, but for this unit all of the models have the same basic structure as shown here in a solution to the multi-column addition model one was to write for unit1:

```
(clear-all)

(define-model tutor-model

(sgp :esc t :lf .05 :trace-detail medium)

;; Add Chunk-types here

(chunk-type addition-fact addend1 addend2 sum)
(chunk-type add-pair one1 ten1 one2 ten2 ten-ans one-ans carry)

;; Add Chunks here

(add-dm
 (fact17 isa addition-fact addend1 1 addend2 7 sum 8)
 (fact34 isa addition-fact addend1 3 addend2 4 sum 7)
 (fact67 isa addition-fact addend1 6 addend2 7 sum 13)
 (fact103 isa addition-fact addend1 10 addend2 3 sum 13)
 (goal isa add-pair ten1 3 one1 6 ten2 4 one2 7))

;; Add productions here

(p start-pair
  =goal>
    ISA add-pair
    one1 =num1
    one2 =num2
    one-ans nil
==>
  =goal>
    one-ans busy
  +retrieval>
```

```
    ISA addition-fact
    addend1 =num1
    addend2 =num2)

(p add-ones
  =goal>
    ISA add-pair
    one-ans busy
    one1 =num1
    one2 =num2
  =retrieval>
    ISA addition-fact
    addend1 =num1
    addend2 =num2
    sum =sum
==>
  =goal>
    one-ans =sum
    carry busy
  +retrieval>
    ISA addition-fact
    addend1 10
    sum =sum)

(p process-carry
  =goal>
    ISA add-pair
    ten1 =num1
    ten2 =num2
    carry busy
    one-ans =ones
  =retrieval>
    ISA addition-fact
    addend1 10
    sum =ones
    addend2 =remainder
==>
  =goal>
    carry 1
    ten-ans busy
    one-ans =remainder
  +retrieval>
    ISA addition-fact
    addend1 =num1
    addend2 =num2)

(p no-carry
  =goal>
    ISA add-pair
    ten1 =num1
    ten2 =num2
    one-ans =ones
    carry busy
  ?retrieval>
    buffer failure
==>
  =goal>
    carry nil
    ten-ans busy
  +retrieval>
    ISA addition-fact
    addend1 =num1
    addend2 =num2)

(p add-tens-done
  =goal>
    ISA add-pair
    ten-ans busy
    carry nil
  =retrieval>
    ISA addition-fact
    sum =sum
==>
  =goal>
    ten-ans =sum)

(p add-tens-carry
  =goal>
    ISA add-pair
    carry 1
    ten-ans busy
  =retrieval>
    ISA addition-fact
    sum =sum
==>
  =goal>
    carry nil
  +retrieval>
    ISA addition-fact
    addend1 1
    addend2 =sum)

(goal-focus goal)
)
```

It starts with a call to a function called **clear-all** followed by a call to the command **define-model**. Inside the call to **define-model** there is a call to a command called **sgp**, then all of the components of the model are defined (chunk-types, chunks, and productions) and finally the starting goal chunk gets set using the **goal-focus** command.

Most of these commands were described in the main unit text, which also described some tools in the ACT-R Environment for inspecting and debugging models. Here we will provide some additional details for some of those commands, describe the sgp command, and show how one can also perform some of the actions that were done using the Environment tools from the prompt in the ACT-R window. In later units, we will also show how those commands can be accessed remotely using the included Python module as an example.

Additional ACT-R command details

Clear-all

The **clear-all** command is used to set the ACT-R software to its initial state. It removes all of the models that are currently defined, returns the clock to time 0, and removes any events which are on the event queue. If a model (or set of models) is contained within a single file, then one probably should call **clear-all** in that file to make sure ACT-R is initialized before defining the model components.

An additional side effect of the **clear-all** command is that it records the name of the file that contains it when it is loaded. That is how the **reload** command and "Reload" button in the ACT-R Environment know which file to load.

Define-model

The **define-model** command takes one required parameter which is the name of a new model to create and then an arbitrary number of other parameters which are the commands that create the initial conditions for the model. The name should be a symbol and must be unique with respect to other models that are currently defined. When a **reset** happens, all of the commands specified inside of the **define-model** are reevaluated for that model in the order they were specified. Each call to **define-model** creates a new model which is independent of the other models, but all of the models will run in parallel when the **run** command is called.

SGP

The **sgp** command is used to set or show the model specific parameters (it stands for set/show general parameters). The parameters for a model control many different things. Some are used in equations that control the performance of the model's cognitive modules, others are there to help the modeler with debugging by changing the outputting of information or the seed of the pseudorandom number generator, and others are available to provide ways that the modeler can extend or modify internal ACT-R mechanisms. The details of all of the parameters can be found in the reference manual.

When using **sgp** to set parameters the syntax is to specify a parameter and then the new value you want to assign to that parameter. Any number of parameters and values may be specified in a single call to **sgp**. All of the parameters in ACT-R begin with a ":" (in Lisp syntax they are keywords). All of the unit1 models have a call to **sgp** similar to this:

```
(sgp :esc t :lf .05 :trace-detail medium)
```

That is setting three parameters**:** **:esc**, **:lf**, and **:trace-detail**. The first two together are specifying that retrieval requests will always take 50ms to complete, but further details on those are beyond the scope of this unit and will be discussed fully in later units. The third parameter being set, **:trace-detail**, controls how much information is shown in the trace when a model runs. The default value is **medium**, and that is also how it is being set in the example above. The other values that it can have are **high** and **low**. When it is set to **high**, effectively every action the model does shows in the trace. Here is the trace of the two-digit addition model with **:trace-detail** set to **high**:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.000   PROCEDURAL             PRODUCTION-SELECTED START-PAIR
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            START-RETRIEVAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   PROCEDURAL             PRODUCTION-SELECTED ADD-ONES
     0.100   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            START-RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   PROCEDURAL             PRODUCTION-SELECTED PROCESS-CARRY
     0.200   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.200   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.250   PROCEDURAL             PRODUCTION-FIRED PROCESS-CARRY
     0.250   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.250   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            START-RETRIEVAL
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK FACT34
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT34
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   PROCEDURAL             PRODUCTION-SELECTED ADD-TENS-CARRY
     0.300   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.300   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.350   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-CARRY
     0.350   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.350   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            START-RETRIEVAL
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK FACT17
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT17
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   PROCEDURAL             PRODUCTION-SELECTED ADD-TENS-DONE
     0.400   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.400   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.450   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-DONE
     0.450   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   ------                 Stopped because no events left to process
```

That can be very useful when debugging a model but it can be a bit too much at other times. Here is the same model running with a **medium** level of **:trace-detail** (which is the default value):

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            START-RETRIEVAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            START-RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED PROCESS-CARRY
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            START-RETRIEVAL
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK FACT34
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT34
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-CARRY
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            START-RETRIEVAL
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK FACT17
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT17
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-DONE
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   ------                 Stopped because no events left to process
```

In that trace we no longer see the individual condition tests and only some of the actions of the productions. Now, here is the same model run with a **low** setting for **:trace-detail**:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.250   PROCEDURAL             PRODUCTION-FIRED PROCESS-CARRY
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT34
     0.350   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-CARRY
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT17
     0.450   PROCEDURAL             PRODUCTION-FIRED ADD-TENS-DONE
     0.450   ------                 Stopped because no events left to process
```

At **low** we only see production firings and buffer settings. The setting of **:trace-detail** does not change how the model actually runs. It only affects how the trace is displayed to the modeler and which events are available to "step" to using the Stepper tool in the ACT-R Environment.

If one wants to completely turn off the model trace, there is another parameter which can be set to do so, and that will be described in a future unit.

To get a parameter's current value using **sgp** only the name of the parameter (or parameters) should be specified. When all of the values passed to **sgp** are the names of parameters it will print out the details of those parameters and return a list of their current values. Typically that is not necessary when defining a model (with one exception which can be very helpful for debugging a model and that will be described in a later unit), but it can be used at the ACT-R prompt to inspect the model settings (there is also an inspector in the Environment for doing so). Here is an example which is checking the values of the **:trace-detail** and **:lf** parameters:

```
? (sgp :trace-detail :lf)
:TRACE-DETAIL MEDIUM (default MEDIUM) : Determines which events show in the trace
:LF 0.05 (default 1.0) : Latency Factor
(MEDIUM 0.05)
```

If no parameters are provided to **sgp** then it will print out all of the parameters and their details.

Interacting with a model

In this unit we used the ACT-R Environment tools to load, run, reset, and inspect the model components. Most of the GUI tools in the Environment are using commands which are also available to the modeler for interacting with the model from the command prompt or in experiment or task code. Here we will describe some of the commands that correspond to the Environment tools used for this unit with respect to using them from the ACT-R command prompt (as was shown above for **sgp**). In future units we will discuss how those commands can also be used in experiments or tasks written in Lisp or Python[1].

Loading a model

To load the models in this unit we used the “Load ACT-R code” button to pick a file to load. There is a command in ACT-R which can also be used to load a model file which is called **load-act-r-model**. It requires one parameter which is a string specifying the pathname of the file to load. Typically, the full pathname of the file must be specified, but it does accept a simplified specification for files located in the directory containing the ACT-R files which is based on the Lisp logical pathname convention. The format for those relative pathnames is to start with ACT-R: and then follow that with the file name or subdirectories separated by semicolons and then the file name. Here is an example of loading the count.lisp model from the default ACT-R distribution:

```
? (load-act-r-model "ACT-R:tutorial;unit1;count.lisp")
T
```

The **T** printed after the call is the return value from the command, and **T** is the Lisp symbol for true, which for this command means it was successful in loading the file. If the file is not found or there is an error in loading it then it will print a warning indicating the issue and the return value would be **nil** instead.

Resetting and Reloading

Instead of pressing the buttons to reset or reload a model one can call the corresponding commands, which are named reset and reload. They require no parameters and will return **T** if successful:

```
? (reset)
```

[1] There are examples of connecting other languages included in the examples/connections directory, but those are just simple examples and would need to be extended to provide sufficient support to be able to implement the tutorial tasks. The documentation on the protocol for remote connections is in the docs directory if one is interested in extending those examples.

```
T
? (reload)
T
```

Running the model

There are actually multiple commands which can be used to run a model based on how it should determine when to stop running. The one that corresponds to the button on the Control Panel is called **run**. It requires one parameter which is the maximum number of seconds to run the model, but it will stop earlier if the model has no more actions to perform. In later units we will introduce more of the running commands and show them being used in creating tasks that automatically run the model as needed. Here is an example of using the **run** command at the prompt after loading the semantic model:

```
? (run 1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL G1 NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK P14
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED DIRECT-VERIFY
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   ------                 Stopped because no events left to process
0.15
24
NIL
```

After the trace there are multiple return values shown from the run command which indicate how long it ran, how many events occurred during the run, and whether it ended unexpectedly or not (a value of **nil** means a successful ending because it was not unexpected).

Inspecting model components

During the unit we inspected the contents of the buffers, the model's declarative memory, and checked why productions did not match. All of those can also be done using commands.

Buffers

You can use the command named **buffer-chunk** to find the names of the chunks in the buffers and inspect their contents. Calling it without any parameters will show all of the buffers and the chunks they contain returning a list of the buffer name and buffer chunk lists:

```
? (buffer-chunk)
GOAL: G1-0
TEMPORAL: NIL
AURAL: NIL
AURAL-LOCATION: NIL
```

```
MANUAL: NIL
VOCAL: NIL
RETRIEVAL: NIL
PRODUCTION: NIL
IMAGINAL: NIL
VISUAL: NIL
VISUAL-LOCATION: NIL
IMAGINAL-ACTION: NIL
((GOAL G1-0) (TEMPORAL) (AURAL) (AURAL-LOCATION) (MANUAL) (VOCAL) (RETRIEVAL)
(PRODUCTION) (IMAGINAL) (VISUAL) (VISUAL-LOCATION) (IMAGINAL-ACTION))
```

If you call it with the name of a buffer (or multiple buffers), then it will print out the chunks in the named buffers and return the list of the names of those chunks:

```
? (buffer-chunk goal)
GOAL: G1-0
G1-0
   OBJECT  CANARY
   CATEGORY  BIRD
   JUDGMENT  YES

(G1-0)
```

Declarative Memory

You can also inspect declarative memory from the command prompt. The command **dm** will print out all of the chunks in the model's declarative memory and return a list with the names of those chunks. You can also specify the names of chunks as parameters to the **dm** command and only those chunks will be printed. Here are some examples using the count model:

```
? (dm)
FIRST-GOAL
   START  TWO
   END  FOUR

FIVE
   NUMBER  FIVE

FOUR
   NUMBER  FOUR
   NEXT  FIVE

THREE
   NUMBER  THREE
   NEXT  FOUR

TWO
   NUMBER  TWO
   NEXT  THREE

ONE
   NUMBER  ONE
   NEXT  TWO

(FIRST-GOAL FIVE FOUR THREE TWO ONE)
```

```
? (dm five two one)
FIVE
   NUMBER  FIVE

TWO
   NUMBER  TWO
   NEXT  THREE

ONE
   NUMBER  ONE
   NEXT  TWO

(FIVE TWO ONE)
```

It is also possible to search declarative memory using the command **sdm**. Its parameters are a chunk specification just as one would specify in a retrieval request in a production, but without the +retrieval> indicator at the beginning. It prints out only those chunks from the model's declarative memory which match that specification and returns the list of their names. Here are some examples using the semantic model:

```
? (sdm object shark)
P1
   OBJECT  SHARK
   VALUE  TRUE
   ATTRIBUTE  DANGEROUS

P2
   OBJECT  SHARK
   VALUE  SWIMMING
   ATTRIBUTE  LOCOMOTION

P3
   OBJECT  SHARK
   VALUE  FISH
   ATTRIBUTE  CATEGORY

(P1 P2 P3)

? (sdm - attribute category - attribute nil)
P1
   OBJECT  SHARK
   VALUE  TRUE
   ATTRIBUTE  DANGEROUS

P2
   OBJECT  SHARK
   VALUE  SWIMMING
   ATTRIBUTE  LOCOMOTION

P4
   OBJECT  SALMON
   VALUE  TRUE
   ATTRIBUTE  EDIBLE
```

```
P5
   OBJECT  SALMON
   VALUE  SWIMMING
   ATTRIBUTE  LOCOMOTION

P7
   OBJECT  FISH
   VALUE  GILLS
   ATTRIBUTE  BREATHE

P8
   OBJECT  FISH
   VALUE  SWIMMING
   ATTRIBUTE  LOCOMOTION

P10
   OBJECT  ANIMAL
   VALUE  TRUE
   ATTRIBUTE  MOVES

P11
   OBJECT  ANIMAL
   VALUE  TRUE
   ATTRIBUTE  SKIN

P12
   OBJECT  CANARY
   VALUE  YELLOW
   ATTRIBUTE  COLOR

P13
   OBJECT  CANARY
   VALUE  TRUE
   ATTRIBUTE  SINGS

P15
   OBJECT  OSTRICH
   VALUE  FALSE
   ATTRIBUTE  FLIES

P16
   OBJECT  OSTRICH
   VALUE  TALL
   ATTRIBUTE  HEIGHT

P18
   OBJECT  BIRD
   VALUE  TRUE
   ATTRIBUTE  WINGS

P19
   OBJECT  BIRD
   VALUE  FLYING
   ATTRIBUTE  LOCOMOTION

(P1 P2 P4 P5 P7 P8 P10 P11 P12 P13 P15 P16 P18 P19)
```

One thing to note about the second example is that it specifies both that the attribute slot is not category and also not nil (which means the chunk must have some value for the slot). Here is what it returns if it were to only specify that the attribute slot is not category:

```
? (sdm - attribute category)
...
(G1 G2 G3 P1 P2 P4 P5 P7 P8 P10 P11 P12 P13 P15 P16 P18 P19 SHARK DANGEROUS
LOCOMOTION SWIMMING FISH SALMON EDIBLE BREATHE GILLS ANIMAL MOVES SKIN CANARY
COLOR SINGS BIRD OSTRICH FLIES HEIGHT TALL WINGS FLYING TRUE FALSE)
```

In this case it also finds all of the chunks which do not have an attribute slot because chunks without the slot fail the “attribute category” test and thus the negation of that is then true. That is something to be careful about when using the negation modifier in specifying retrieval requests in your productions as well.

Testing why not? for productions

The command to test whether a production matches the current state is called **whynot**. If you pass it no parameters it will print out each production in the model’s procedural memory along with either an indication that it matches or a reason why it does not match. You can also provide it with specific production names to test. It returns a list of the names of the productions which do match the current state. Here is an example using productions from the semantic model after it has been reset:

```
? (whynot direct-verify fail)

Production DIRECT-VERIFY does NOT match.
(P DIRECT-VERIFY
   =GOAL>
       OBJECT =OBJ
       CATEGORY =CAT
       JUDGMENT PENDING
   =RETRIEVAL>
       OBJECT =OBJ
       ATTRIBUTE CATEGORY
       VALUE =CAT
 ==>
   =GOAL>
       JUDGMENT YES
)
It fails because:
The GOAL buffer is empty.

Production FAIL does NOT match.
(P FAIL
   =GOAL>
       OBJECT =OBJ1
       CATEGORY =CAT
       JUDGMENT PENDING
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =GOAL>
       JUDGMENT NO
)
It fails because:
The GOAL buffer is empty.
NIL
```

Other model commands

Any of the commands that are specified in the model definition can also be called from the ACT-R command prompt. The **add-dm** and **p** commands are not usually called from the prompt since you want that knowledge to be included in the model when it starts running, but a command like **goal-focus** can occasionally be useful if you want to change the goal chunk for a model and run it again. That could have been convenient for the semantic model because instead of changing the file and reloading it to switch the goals one could have run the different goals sequentially by just calling goal-focus to change the chunk in the goal buffer and then run it again:

```
? (run 1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL G1 NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK P14
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED DIRECT-VERIFY
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   ------                 Stopped because no events left to process
0.15
24
NIL
? (goal-focus g2)
G2
? (run 1)
     0.150   GOAL                   SET-BUFFER-CHUNK GOAL G2 NIL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.200   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.200   DECLARATIVE            start-retrieval
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   DECLARATIVE            RETRIEVED-CHUNK P14
     0.250   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.300   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.300   DECLARATIVE            start-retrieval
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   DECLARATIVE            RETRIEVED-CHUNK P20
     0.350   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P20
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   PROCEDURAL             PRODUCTION-FIRED DIRECT-VERIFY
     0.400   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   ------                 Stopped because no events left to process
0.25
36
NIL
? (goal-focus g3)
G3
? (run 1)
     0.400   GOAL                   SET-BUFFER-CHUNK GOAL G3 NIL
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED INITIAL-RETRIEVE
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   DECLARATIVE            start-retrieval
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   DECLARATIVE            RETRIEVED-CHUNK P14
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P14
     0.500   PROCEDURAL             CONFLICT-RESOLUTION
     0.550   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.550   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.550   DECLARATIVE            start-retrieval
     0.550   PROCEDURAL             CONFLICT-RESOLUTION
     0.600   DECLARATIVE            RETRIEVED-CHUNK P20
     0.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P20
     0.600   PROCEDURAL             CONFLICT-RESOLUTION
     0.650   PROCEDURAL             PRODUCTION-FIRED CHAIN-CATEGORY
     0.650   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.650   DECLARATIVE            start-retrieval
     0.650   PROCEDURAL             CONFLICT-RESOLUTION
     0.700   DECLARATIVE            RETRIEVAL-FAILURE
     0.700   PROCEDURAL             CONFLICT-RESOLUTION
     0.750   PROCEDURAL             PRODUCTION-FIRED FAIL
     0.750   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.750   PROCEDURAL             CONFLICT-RESOLUTION
     0.750   ------                 Stopped because no events left to process
0.35
48
NIL
```

ACT-R Model Writing

This text and the corresponding texts in other units of the tutorial are included to help introduce cognitive modelers to the process of writing, testing, and debugging ACT-R models. Unlike the main tutorial units which cover the theory and use of ACT-R, these documents will cover issues related to using ACT-R from a software development perspective. They will focus mostly on how to use the tools provided to build and debug models, and will also describe some of the typical problems one may encounter in various situations and provide suggestions for how to deal with those issues.

Models are Programs

The important thing to note up front is that an ACT-R model is a program – it is a set of instructions which will be executed by the ACT-R software. There are many different methodologies which one can use when writing programs as well as different approaches to software testing which one can employ. These guides are not going to promote any specific approaches to either task. Instead, they will attempt to describe general techniques and the tools which one can use when working with ACT-R regardless of the programming and testing methods being used.

Learning to write ACT-R models is similar to learning a new programming language. However, ACT-R as a programming language differs significantly from most other languages and the objectives of writing a cognitive model are typically not the same things one tries to achieve in other programming tasks. Because of that, one of the difficulties that many beginning ACT-R modelers have is trying to treat writing an ACT-R model just like a programming task in any other programming language. Some of the important differences to keep in mind while modeling with ACT-R will be described in this section.

From a high level perspective, a significant difference between ACT-R and other programming languages is what will be running the program. The model is not being written as commands for a computer to execute, but as commands for a cognitive processor (essentially a simulated human mind) to perform. In addition to that, the operators available for use in writing the model are very low-level actions, much like assembly language in a computer programming language. Thus ACT-R is basically the opposite of most programming languages. It is a very low-level language written to run on a "processor" with many high-level capabilities built into it whereas most languages are a high-level set of operators targeting a very general low-level processor for execution.

Another important difference is how the sequence of actions is determined. In many programming languages the programmer specifies the commands to perform as a specific sequence of instructions with each one happening after the previous one, as written in the program. For ACT-R however the order of the productions in the model definition does not matter, nor does the order of the tests within an individual production matter. The next action to perform, i.e. which production to fire, is based on which one currently matches the current state of the buffers and modules, and that requires satisfying all of the conditions on the LHS of a production. Thus, the modeler is responsible for explicitly building the sequence of actions to take into the model because there is no automatic way to have the system iterate through them "in order".

Finally, perhaps the biggest difference between writing a cognitive model and most other programming tasks is that for cognitive modeling one is typically attempting to simulate or predict human behavior and performance, and human performance is often not optimal or efficient from a computer programming perspective. Thus, optimizations and efficient design metrics which are important in normal programming tasks, like efficient algorithms, code reuse, minimal number of steps, etc, are not always good design choices for creating an ACT-R model because such models will not perform "like a person". Instead, one has to consider the task from a human perspective and rely on psychological research and performance data to guide the design of the model.

ACT-R and Lisp

While ACT-R is its own modeling language, it is itself written in Lisp. ACT-R models are written using Lisp syntax and the ACT-R prompt is really just an interactive Lisp session. Because of that, some familiarity with Lisp programming can be helpful when working with ACT-R, but it is not absolutely required.

Errors and Warnings

When writing a model one is likely to encounter warnings from ACT-R and occasionally warnings and errors from the underlying Lisp. This section will provide some information on how to determine whether the problem was reported by ACT-R or the underlying Lisp and how to deal with those from the ACT-R command prompt in the standalone application version. This document is not going to describe how one would handle errors in other Lisp programs which may be used if one is running ACT-R from source code (presumably if you are comfortable enough to run ACT-R from sources you are already familiar with the software you are using to do so or can consult the appropriate documentation).

ACT-R Warnings

Warnings from ACT-R were seen when loading the semantic model as described in the primary text for this unit. They are an indication that there is a potentially problematic situation in the ACT-R model or code which is using ACT-R commands. An ACT-R warning may occur when the model is loaded and also while the model is being run. An ACT-R warning can be distinguished from a Lisp warning because the ACT-R warnings will always be printed inside of the Lisp "block comment" character sequence #| and |# and start with the word "Warning" followed by a colon. Here are some examples of ACT-R warnings:

```
#|Warning: Creating chunk STARTING with no slots |#

#|Warning: A retrieval event has been aborted by a new request |#
```

```
#|Warning: Production TEST already exists and it is being redefined. |#
```

When you get a warning from ACT-R, the recommendation is to make sure that you read the warning and determine whether it is an issue which needs to be corrected or is simply an indication of something that is not significant to the operation of the model. Ideally, the model should not generate any warnings, but occasionally it is convenient to just ignore an inconsequential warning, particularly early on in the development of a model or task when the warning is being generated by something that you haven't yet completed. However, you should be very careful when doing that because if you just start ignoring the warnings because you're "expecting them" you may miss a new warning that occurs which indicates a significant problem. Something else to be careful of is that many ACT-R warnings are only displayed when the ACT-R trace is enabled. Thus, until you are certain that a model is performing correctly the recommendation is to leave the trace enabled, and if you encounter any problems while the model is running with the trace turned off, turning the trace back on may show the warnings that indicate the issue.

Some of the most common ACT-R warnings will be described in more detail in this and later units of the model writing texts. If you do not understand what a particular ACT-R warning means, then one thing you can do to find out more information is search the ACT-R reference manual to find an example with the same or similar warning (things specific to the model like chunk or production names found in the warning would of course have to be omitted in the search). That should help to narrow down which ACT-R command generated the warning and provide more details about it.

Lisp Warnings

Lisp warnings are similar to ACT-R warnings in that they are an indication that something unexpected or unusual was encountered which you may need to correct. You will typically only encounter Lisp warnings if you are building experiments or tasks in Lisp for the model. Here are some typical things that will generate a Lisp warning: defining functions that use undefined variables, defining variables in functions and then not using them, loading a file which redefines a function that was defined elsewhere, and defining functions that reference other functions which do not yet exist. A Lisp warning will typically be displayed after the prompt as a Lisp comment which starts with a semicolon. Because they do not cause the system to halt they are often easy to ignore, but as with ACT-R warnings, the recommendation is to read and understand every warning that is displayed when you load a model or task file. Here is an example of a warning displayed after loading a task file that contains a function named test which creates a variable called x, but does not use it:

```
;Compiler warnings :
;   In TEST: Unused lexical variable X
TEST
?
```

Lisp Errors

An error is a serious condition that has occurred in Lisp and it will often cause things to stop until it is dealt with. Typical things that will cause a Lisp error are missing or unbalanced parenthesis that result in invalid Lisp syntax in the model file, trying to use commands which do not exist, or calling commands with invalid or an incorrect number of arguments. When an error occurs you will see some details about the error in the ACT-R window and you should resolve that problem before continuing. Here is an example showing an error when trying to call the command run without specifying a time:

```
? (run)
> Error: Too few arguments in call to #<Compiled-function RUN #x10131F69F>:
>        0 arguments provided, at least 1 required.
> While executing: RUN, in process listener(1).
> Type :POP to abort, :R for a list of available restarts.
> Type :? for other options.
1 >
```

It starts with a description of the error, which may be a little cryptic if you are not familiar with Lisp, but typically should give some indication of what caused the problem. After that it provides some information on how one can handle the error, and then below that you will see that the input prompt has changed from a "?" to a number followed by a ">" character. That prompt indicates that there is currently 1 error which has not been resolved. The recommendation is to always just abort the error by typing :pop, and that will return the prompt back to the "?".

Debugging Example

To show the tools one can use to debug an ACT-R model and describe some of the issues one may encounter when working with ACT-R models we will work through the process of debugging a model which is included with the unit materials. We will start with a model for the task that does not work and then though testing and debugging determine the problems and fix them, showing the ACT-R tools which one can use along the way. This task is going to start the testing and debugging with essentially the whole model written. When writing your own models you may find it easier to perform incremental testing as you go instead of waiting until you have written everything, and the same tools and processes would be applicable then too.

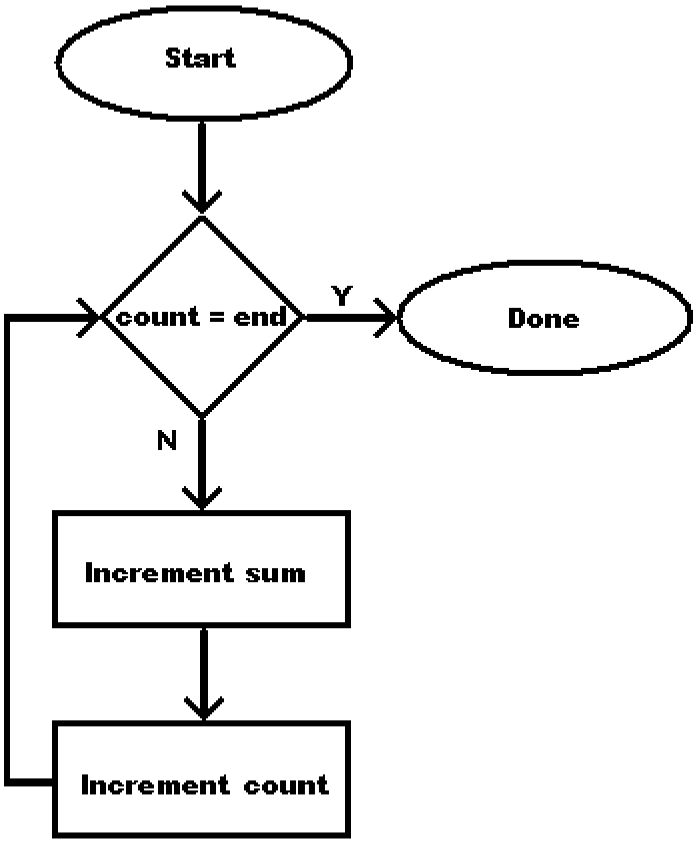
For this unit we will work through a broken version of the addition by counting task which is in the broken-addition.lisp file. Before starting the debugging process however, we will first look at the overall design of this model because without knowing what it should do we cannot appropriately determine if it is or is not doing the correct thing.

Addition Model design

Before starting to write a model it is useful to start with some design for how you intend the model to work. It does not have to be a complete specification of every step the model will take, but should at least provide a plan for where it starts, the general process it will follow, and what the end condition and results are. As you write the model you may also find it useful to update the design with more details as you go. In that way you will always have a record of how the model works and what it is supposed to do. Below is design information for the addition model provided at increasing levels of detail.

Here is the very general description of the model. This model will add two numbers together by counting up from the first number (incrementally adding one) a number of times indicated by the second number. It does this by retrieving chunks from declarative memory that indicate the ordering of numbers from zero to ten and maintaining running totals for the sum and current count in slots of the goal buffer.

Based on that description we can expand that a little and create a simple flow chart to indicate the basic process the model will follow:



Another important thing to specify is the way that the information will be encoded for the model, and generally that will involve specifying chunk-types for the task. Here are the chunk-types which we will use for this model:

The number chunk-type will be used to create chunks which encode the sequencing of numbers by indicating the order for a pair of numbers with number preceding next:

```
(chunk-type number number next)
```

The add chunk-type will be used to create chunks indicating the goal of adding two numbers and it contains slots for holding the two numbers, the final sum, and the running count as we progress through the additions:

```
(chunk-type add arg1 arg2 sum count)
```

The design of a model should also indicate detailed information about the starting conditions and the expected end state for the model. Here are some details for those aspects of this task:

Start: the model will have a chunk in the goal buffer. That chunk will have the starting number in the arg1 slot, the number to add to it will be in the arg2 slot, and it will have no other slots.

End: when the model finishes, the value of the sum slot of the chunk in the goal buffer will be the result of adding the number in that chunk's arg1 slot to the number in its arg2 slot.

For the model we've created for this task, we will also indicate specific details for what each of the productions we've written is supposed to do. Our model consist of four productions, each corresponding to a state in the flow chart above with the branching test encoded as conditions within the productions for the possible branch states of done and increment sum.

initialize-addition: (the start state) If the goal chunk has values in the arg1 and arg2 slots and does not have a sum slot then set the value of the sum slot to be the value of the arg1 slot, add a count slot with the value zero, and make a request to retrieve a chunk for the number in the arg1 slot.

terminate-addition: (the done state) If the goal has the value of the count slot equal to the value of the arg2 slot then stop the model, which will be done by removing the count slot from the goal chunk.

increment-sum: If the goal has a sum slot with a value and the value of the count slot is not equal to the value of the arg2 slot and we have retrieved a chunk for incrementing the current sum then update the sum slot to be the value from the next slot of that retrieved chunk and retrieve a chunk to increment the current count.

increment-count: If the goal has a sum and count and we have retrieved a chunk for incrementing the current count value then update the count slot to be the value from the next slot of that retrieved-chunk and retrieve a chunk to increment the current sum.

Now that we know what the model is supposed to do in detail, we can start testing what has been implemented thus far.

Loading the model

The first step is of course to load the model. However, when we do so we encounter a Lisp error. If the file is being loaded with the "Load ACT-R code" button on the Control Panel an "Error Loading" window will be displayed indicating an end of file (or "eof") occurred which will has some details that start like this:

```
Error #<SIMPLE-ERROR Error #<END-OF-FILE Unexpected end of file on #<BASIC-FILE-CHARACTER-INPUT-STREAM ...
```

If we load that using the load-act-r-model command we get an ACT-R warning which actually indicates that a Lisp error occurred, and what happened is that the load-act-r-model command actually prevents the Lisp error from occurring and automatically aborted it:

```
? (load-act-r-model "ACT-R:tutorial;unit1;broken-addition.lisp")
#|Warning: Error "Error #<SIMPLE-ERROR Error #<END-OF-FILE Unexpected end of file on
#<BASIC-FILE-CHARACTER-INPUT-STREAM (\"C:/Users/... \"/:closed #x21009AA3AD>> while
trying to load file \"ACT-R:tutorial;unit1;broken-addition.lisp\"> occurred while
trying to evaluate command \"load-act-r-model\" with parameters (\"ACT-
R:tutorial;unit1;broken-addition.lisp\" NIL)" while attempting to evaluate the form
("load-act-r-model" "ACT-R:tutorial;unit1;broken-addition.lisp" NIL) |#
NIL
```

Both of those are a little difficult to read, but whenever an error message contains END-OF-FILE it almost always means that there is a missing right parenthesis somewhere in the file, but some other possible causes could be an extra left parenthesis, a missing double-quote character, or an extra double-quote character. To fix this we will have to look at the file, find what is missing or doesn’t belong and correct it. If you are using an editor that has built in support for Lisp code, then it shouldn’t be too difficult to match parentheses or otherwise locate the issue, but if your editor does not have such capabilities then unfortunately it may be a difficult process to track down the problem. In this case, what we find is that the closing right parenthesis of the define-model call is missing at the very end of the file. After adding that into the file and saving it we should try to load it again. The load should be successful now, but there are several more ACT-R warnings which we should investigate before trying to run it.

Initial ACT-R Warnings

Here are the warnings displayed when the model is loaded:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ADD ARG1 =NUM1 ARG2
=NUM2 SUM NIL ==> =GOAL> ISA ADD SUM =NUM1 COUNT ZERO +RETRIEVAL> ISA NUMBER NUMBER
=NUM1). |#
#|Warning: Invalid syntax in (=GOAL> ADD ARG1 =NUM1 ARG2 =NUM2 SUM NIL) condition. |#
#|Warning: Cannot use nil as a slot name. |#
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
#|Warning: Production TERMINATE-ADDITION uses previously undefined slots (SUMM). |#
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
#|Warning: Productions test the SUMM slot in the GOAL buffer which is not requested or
modified in any productions. |#
#|Warning: Productions test the ARG2 slot in the GOAL buffer which is not requested or
modified in any productions. |#
```

Whenever a model generates ACT-R warnings when it is loaded the next step one takes should be to understand why the model generated those warnings because there is no point in trying to run it unless you know what problems it may have right from the start. Sometimes the warnings indicate a situation that is acceptable to ignore, like the default chunk creation warnings shown in the unit 1 text for the semantic model, but often they indicate something more serious which must be corrected in the model before it will run as expected.

To determine what the warnings mean one should start reading them from the top down because sometimes there may be multiple warnings generated for a single issue. Productions in particular often generate several warnings when there is a problem with creating one. For this model, all of the warnings are related to production issues and we will look at them in detail here to help explain what they mean.

This first warning indicates that the definition of the initialize-addition production, which it shows in the warning, is not valid and thus it could not create that production:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ADD ARG1 =NUM1 ARG2
=NUM2 SUM NIL ==> =GOAL> ISA ADD SUM =NUM1 COUNT ZERO +RETRIEVAL> ISA NUMBER NUMBER
=NUM1). |#
```

Whenever there is a "No production defined" warning there will be more warnings after that which will provide the details about what specifically was wrong with the production. In this case this is the next warning:

```
#|Warning: Invalid syntax in (=GOAL> ADD ARG1 =NUM1 ARG2 =NUM2 SUM NIL) condition. |#
```

It's telling us that there is something wrong with the =goal> test on the LHS, and the next warning provides additional details which may also help with that:

```
#|Warning: Cannot use nil as a slot name. |#
```

That is telling us that nil is in a slot name position of that =goal> condition and that nil is not a valid name for a slot.

The next warning displayed is just an indication that there are no more warnings about the problem in the initialize-addition production:

```
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
```

Before continuing to look at the rest of the warnings we will first understand what exactly lead to this sequence for the production. We know where the problem lies, the goal buffer test of the initialize-addition production, and the warning is telling us that it's trying to interpret nil as a slot name, but it doesn't tell us exactly why that is happening. We don't intend nil to be a slot name so that means the problem is likely elsewhere in the goal condition. If you look at the production it may be obvious what is wrong, but here we will look at the condition as displayed in the warning:

```
#|Warning: Invalid syntax in (=GOAL> ADD ARG1 =NUM1 ARG2 =NUM2 SUM NIL) condition. |#
```

The condition gets processed from left to right so we will look at it that way instead of focusing on the nil value which is indicated in the warning. After the buffer name, we see the first thing specified is add. That is not the name of a slot which we intend to use, but is the name of the chunk-type we are using to specify the goal chunk. However, to indicate the chunk-type for a condition we need to use the symbol isa, which is missing here. So the missing isa is likely the source of the problem. The warning doesn't tell us that because having an isa is not required in a buffer test – it's acceptable to only specify slots and values without indicating a chunk-type which is what happens here since there is no isa symbol. Thus it is parsing that condition as the add slot having a value of arg1, a slot named =num1 with a value arg2, a slot named =num2 with the value sum, and then a slot named nil which doesn't have a value. Nil being invalid as a slot name is the first thing which the production detects as being a problem with that and thus that's when it stops and produces the warning.

At this point one can either fix that problem and try loading it again or continue reading through the warnings. For the purposes of this text we are going to continue through all of the warnings first and then fix them afterwards, but some people prefer to fix problems one at a time and would instead stop here and fix the initialize-addition production before continuing.

The next warning is this one:

```
#|Warning: Production TERMINATE-ADDITION uses previously undefined slots (SUMM). |#
```

That indicates that the symbol summ occurs as a slot name in the production terminate-addition and that name wasn't specified in any of the chunk-types as a slot name.

The following warning is telling us that there are multiple productions with the name increment-sum, and thus the earlier one is being overwritten by a later one:

```
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
```

The last two warnings are what we would call style warnings in the productions:

```
#|Warning: Productions test the SUMM slot in the GOAL buffer which is not requested or
modified in any productions. |#
#|Warning: Productions test the ARG2 slot in the GOAL buffer which is not requested or
modified in any productions. |#
```

They indicate that in one or more productions there is a goal buffer testing for slots named summ and arg2 but those slots do not appear in any of the productions' actions. Style warnings describe a situation that exists among all the productions in the model and may point to an error in the logic of the productions or inconsistency in the usage of the chunk slots. However, since we have a production which was not defined and a previous warning about the slot summ, style warnings are not unexpected. Fixing those other issues may also eliminate the style warnings.

Now that we've looked over the warnings, some of them are things which need to be fixed before we can run the model and we will address them one at a time in the next section. One note before doing so however is to point out that occasionally the warnings may not be as easy to understand as these or reference ACT-R commands that don't occur explicitly in the model. In those cases you may need to consult the ACT-R reference manual to find out more information.

Fixing initialize-addition

As described above, the issue is that we are missing the isa symbol from the goal condition. Here is the initialize-addition production from the model:

```
(P initialize-addition
   =goal>
      add
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal
      ISA         add
      sum         =num1
      count       zero
   +retrieval>
      ISA         number
      number      =num1
)
```

If we add the missing isa to the goal condition like this:

```
(P initialize-addition
   =goal>
      isa         add
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal
      ISA         add
      sum         =num1
      count       zero
   +retrieval>
      ISA         number
      number      =num1
)
```

That should fix the problem. Alternatively, we could just remove add from the condition instead since the chunk-type is an optional declaration in a buffer test:

```
(P initialize-addition
   =goal>
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal
      ISA         add
      sum         =num1
      count       zero
   +retrieval>
      ISA         number
      number      =num1
)
```

Fixing terminate-addition

The last warning we have that's not a style warning is this one:

```
#|Warning: Production TERMINATE-ADDITION uses previously undefined slots (SUMM). |#
```

Here is the text of the terminate-addition production from the model:

```
(P terminate-addition
   =goal>
      ISA         add
      count       =num
      arg2        =num2
      summ         =answer
  ==>
   =goal>
      ISA         add
      count       nil
)
```

This warning seems fairly straight forward. There was a typo in the condition of the production and a slot name summ was used instead of the correct name sum.

```
(P terminate-addition
   =goal>
      ISA         add
      count       =num
      arg2        =num2
      sum         =answer
  ==>
   =goal>
      ISA         add
      count       nil
)
```

Fixing increment-sum

Here is the warning about increment-sum:

```
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
```

In this case the problem is two productions with the same name. A simple approach would be to just change the name of one of them to clear the warning, but it is better to understand why we have two productions with the same name and then if both are indeed valid productions to name them correctly.

Comparing the productions in the model to the design of the task that we created it appears that the second instance of increment-sum is the correct version and that the first one should be increment-count. Something like that may have come about simply as a typo or perhaps by copying-and-pasting increment-sum, since the two productions are very similar, and then failing to change the name on that new one after making other changes to it. Whatever the cause, we will now change the name of the first one to increment-count.

Now that we've addressed those warnings we need to save the file and reload it.

More Warnings

When we load the model now we see another set of warnings:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ISA ADD ARG1 =NUM1
ARG2 =NUM2 SUM NIL ==> =GOAL ISA ADD SUM =NUM1 COUNT ZERO +RETRIEVAL> ISA NUMBER
NUMBER =NUM1). |#
#|Warning: First item on RHS is not a valid command |#
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
#|Warning: Productions test the ARG2 slot in the GOAL buffer which is not requested or
modified in any productions. |#
```

Again it is indicating problems with the initialize-addition production and we still have one of the style warnings. This is an important thing to note about production warnings. No matter how many problems may exist in a production, only one of them will generate warnings at a time because once a problem is detected no further processing of that production will occur. Thus it may take several iterations of addressing the warnings, fixing the production issues, and then reloading before all of the productions are syntactically correct.

This time the warning for initialize-addition indicates that there is a problem with the first action on the RHS of the production, and here again is our updated version of the production:

```
(P initialize-addition
   =goal>
      isa add
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal
      ISA         add
      sum         =num1
      count       zero
   +retrieval>
      ISA         number
      number      =num1
)
```

Looking at that closely, we can see that there is a missing “>” symbol at the end of the goal modification action above. We will add that, save the model, and load it yet again.

Still Have Style Warnings

There are still some style warnings displayed when we load the model:

```
#|Warning: Productions test the ARG1 slot in the GOAL buffer which is not requested or
modified in any productions. |#
#|Warning: Productions test the ARG2 slot in the GOAL buffer which is not requested or
modified in any productions. |#
```

We could try to address those now, but since all of the productions are at least defined we will temporarily ignore them and see what happens when we try to run the model. It may be that these are safe to ignore, and since understanding them may require investigating all the productions to determine what is wrong we will just try a quick test of the model at this point and then come back to understanding and fixing them later, if necessary.

Testing

When testing a model one of the important issues is generating meaningful tests, and the design of the model is useful in determining what sorts of things to test. The tests should cover a variety of possible input values to make sure the model is capable of handling all the types of input it is expected to be able to handle. Similarly, tests should be done to make sure that all of the components of the model operate as intended. Thus, if the model has different strategies or choices it can make there should be enough tests to make sure that all of those strategies operate successfully. Similarly, if the model is designed to be capable of detecting and/or correcting for invalid values or unexpected situations one will also want to test a variety of those as well. While it is typically not feasible to test all possible situations, one should test enough of them to feel confident that the model is capable of performing correctly.

Because this model does not have any different strategies or choices nor is it designed to be able to deal with unexpected situations we only really need to generate tests for valid inputs, which are addition problems of non-negative numbers with sums between zero and ten. Because that is not an extremely large set of options (only 66 possible problems) one could conceivably test all of them, particularly if some task code was written to generate and verify them automatically, but being able to enumerate all the possible cases is not usually feasible. Thus, we will treat this as one would a more general task and generate some meaningful test cases to run explicitly instead of trying to automate it.

One way to generate tests would be to just randomly pick a bunch of different addition problems, but a more systematic approach is usually much more useful. When dealing with a known range of possible values, a good place to start is to test values at the beginning and end of the possible range, and starting with what seems to be the easiest case is usually a good start. Thus, our first test will be to see if the model can correctly add 0+0.

The First Run

To do that, we need to create a chunk to place into the goal buffer with those values in it. The model as given already has such a chunk created called test-goal found along with the other chunks created for declarative memory. So, at this point it might seem like a good time to try to run the model, and here is what we get when we do:

```
? (run 10)
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.000   ------                 Stopped because no events left to process
```

Nothing happened. While it may be obvious to you why this model did not do anything at this point, we are still going to walk through the steps that one can take to figure that out. The first step in figuring that out is to determine what you expect should have happened, and having a thorough design can be helpful with that. In this case what should have happened is that the initialize-addition production should fire to start the model along the task.

When one expects a production to fire and it does not, the ACT-R tool that can be used to determine the reason is the whynot command because that will explain why a production did not match the current context. That tool is accessible either by calling the command at the prompt, or through the procedural viewer in the ACT-R Environment. When using the whynot command one can provide any number of production names along with it (including none). For each of the productions provided it will print out a line indicating whether the production matches or not, and then either the current instantiation of the production if it does match the current context or the production itself along with a reason why it does not match. If no production names are provided then the whynot information will be reported for all productions. To use the tool in the procedural viewer one must highlight a production in the list of productions on the left of the window and then press the button labeled “Why not?” on the top left. That will open another window which will contain the same information as is displayed by the whynot command.

Because the model stopped at the time when we expected that production to be selected we can use the whynot tool now and find out why it did not fire in the current model state. Here are the results of calling the whynot command for initialize-addition:

```
? (whynot initialize-addition)
```

```
Production INITIALIZE-ADDITION does NOT match.
(P INITIALIZE-ADDITION
   =GOAL>
       ARG1 =NUM1
       ARG2 =NUM2
       SUM NIL
 ==>
   =GOAL>
       SUM =NUM1
       COUNT ZERO
   +RETRIEVAL>
       NUMBER =NUM1
)
It fails because:
The GOAL buffer is empty.
```

It did not fire because the goal buffer is empty. Looking at our model we can see that the goal buffer is empty because we do not call the goal-focus command to put the test-goal chunk into the buffer. We need to add this:

```
 (goal-focus test-goal)
```

to the model definition or call that from the command prompt before running the model. Since we will probably need to make more changes to the model over time it's probably best to just add that to the file, save it, and then reload.

When we load it we see that the style warnings no longer appear. Setting an initial goal removed the problem that was being indicated – the model was testing for slots in a goal buffer chunk which weren't being set by the productions. By putting a chunk with those slots into the buffer when the model is defined the warnings go away. If we had not skipped over the style warnings we may have been able to determine that setting a goal chunk was necessary prior to running it.

The Second Run

Here is what happens when we run it now:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK ZERO
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.100   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   ------                 Stopped because no events left to process
```

Looking at that trace, it has fired the productions we would expect from our design. First it initializes the addition process, and then it terminates because we have counted all the numbers that it needed (which is none). The next thing to check is to make sure that it performed the changes to the goal buffer chunk as intended to create the appropriate result.

To check the chunk in the goal buffer we can use either the buffer-chunk command from the prompt or the buffers tool in the ACT-R Environment. For the command, any number of buffer names can be provided (including none). For each buffer provided it will print out the buffer name, the name of the chunk in the buffer, and then print the details for that chunk. If no buffer names are provided then for every buffer in ACT-R it will print the name of the buffer along with the name of the chunk currently in that buffer. To use the buffers tool one can select a buffer from the list on the left of the window and then the details as would be printed by the buffer-chunk command for that buffer will be shown on the right. One may open multiple buffers windows if desired, which can be useful when comparing the contents of different buffers.

Here is the output from the buffer-chunk command for the goal buffer:

```
? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ZERO
   ARG2  ZERO
   SUM  ZERO
```

There we see that the sum slot has the value 0 which is what we expect for 0+0. The model has worked successfully for this test. However, that was a very simple case and we do not yet know if it will actually work when there is counting required, or in fact if it can add zero to other numbers correctly. Thus, we need to perform more tests before we can consider the model to be finished.

Next Test

For the next test it seems reasonable to verify that it can also add 0 to some other number since that does not involve any more productions than the last test and would be good to know before trying any more involved tasks. To do that we will try the problem 1+0 and to do so we need to change the arg1 value of test-goal from 0 to 1 like this in the model:

```
(test-goal ISA add arg1 one arg2 zero)
```

We need to then save that change and reload the model. Now we will run it again and here is the result of the run and the chunk from the goal buffer:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK ONE
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   ------                 Stopped because no events left to process


? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ONE
   ARG2  ZERO
   SUM  ONE
```

Everything looks as we would expect so now it seems reasonable to move on to a test which requires actually adding numbers.

Test with Addition

Since this is the first test of performing an addition we should again create a simple test, and adding 1+1 seems like a good first step since we know the model can add 0+0 and 1+0 correctly. To do that we again change the chunk test-goal, save and load the model.

```
(test-goal ISA add arg1 one arg2 one)
```

Before running it, it would be a good idea to make sure we know what to expect. Given the model design above, we expect to see four productions fire in this order: initialize-addition to get things started, increment-sum to add the first number, increment-count to update the count value, and then terminate-addition since our count will then be equal to 1.

Here is what we get when we run it:

```
? (run 10)
     0.000   GOAL                    SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL              CONFLICT-RESOLUTION
     0.050   PROCEDURAL              PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE             start-retrieval
     0.050   PROCEDURAL              CONFLICT-RESOLUTION
     0.100   DECLARATIVE             RETRIEVED-CHUNK ONE
     0.100   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL              PRODUCTION-FIRED TERMINATE-ADDITION
     0.100   PROCEDURAL              CONFLICT-RESOLUTION
     0.100   ------                  Stopped because no events left to process
```

It does not do what we expected it to do. The first production fired as we would expect, but then instead of the increment-sum production firing the terminate-addition production fired and stopped the process just as it did when the model was adding 0. Now we have to determine what caused that problem, and the first step towards doing that is determining when in the model run the first problem occurred.

Typically, the first thing to do is to look at the trace and compare it to the actions we would expect to happen. When doing that it is often helpful to have more detail in the trace so that we see all of the actions that occur in the model. Thus, we would want to set the :trace-detail parameter to high in the model, save it, load it, and then run it again.

```
(sgp :trace-detail high :esc t :lf .05)
```

Here is the trace with the detail level set to high:

```
? (run 10)
     0.000   GOAL                    SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL              CONFLICT-RESOLUTION
     0.000   PROCEDURAL              PRODUCTION-SELECTED INITIALIZE-ADDITION
     0.000   PROCEDURAL              BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL              PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL              MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL              MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-SELECTED TERMINATE-ADDITION
     0.050   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   DECLARATIVE            RETRIEVED-CHUNK ONE
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     0.100   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   ------                 Stopped because no events left to process
```

Reading through that trace the first thing that seems wrong is the selection of terminate-addition at time 0.050 (which doesn't show up in the trace with the default trace-detail level). So, that is where we will investigate further to determine why the problem occurred. With more complicated models, reading through the trace may not provide quite as definitive an answer, because there could be situations where everything appears to go as expected but the model still generates a wrong result. In those cases, it may be necessary to add even more detail to the trace by putting !output! actions into the productions to display additional information or to walk through the model one event at a time using the stepper tool of the ACT-R Environment (as we will discuss later) and inspect the buffer contents and module states along the way.

Now that we know the problem seems to be at time 0.050 with the selection of the terminate-addition production the next step is figuring out why it is selected at that time. One could start by just looking at the model code and trying to determine why that may have happened, but for the purposes of this exercise we will do a more thorough investigation using the stepper tool because often one will need to see more information about the current state of the system at that time to determine the problem. [If one does not want to use the stepper tool in the ACT-R Environment there is also a run-step command which can be called instead of run to step through things at the prompt, but we will not describe the use of that here and you should consult the ACT-R reference manual for details on using that command instead.] To use the stepper tool it should be opened before running the model (it can be opened while a model is running but it is best to open it in advance so that one does not miss the early events that occur), and then when the model is run the stepper will stop the system before every event that will be displayed in the trace (thus the trace-detail setting also controls how detailed the stepping is with the stepper tool). While the stepper has the model paused, it will show the action that will happen next near the top of the stepper display and for some actions it will also show additional details in the windows below that after they occur. When the stepper has the system paused, all of the other Environment tools can still be used to inspect the components of the system. Now that we have an idea where the problem occurs we want to get the model to that point and investigate further. So, we should reset the model, open the stepper tool, and then run the model.

To get to the event we are interested in, the production selection at time 0.050 seconds, one could just continually hit the step button until that action is the next one. For this model, since there are not that many actions, that would not be difficult. However, if the problem occurs much later into a run, that may not be a feasible solution. In those situations one will want to take advantage of the "Run Until:" button in the stepper. That can be used to run the model up to a specific time, until a specific production is selected or fired, or there is an event generated by a specified module. To select which type of action to run until one must select it using the menu button to the right of the "Run Until:" button, and then one must provide the details of when to stop (a time, production name, or module name) in the entry to the right of that button. For this task, since we are interested in the selection of a production we can use the run until button to make that easier. Thus, we should select production from the menu button, type terminate-addition in the entry box, and then press the “Run Until:” button. Doing that we see the trace printed out up to that point and the stepper now shows that selection is the next event. Our design for this production is that it should stop the model when there is a sum and the count is equal to the second argument, or specifically when the count slot of the chunk in the goal buffer is the same as the arg2 slot of the chunk in the goal buffer. If we look at the chunk in the goal buffer at this time we see that those values are not the same:

```
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ONE
   ARG2  ONE
   SUM  ONE
   COUNT  ZERO
```

Thus there is likely something wrong with the terminate-addition production. We can look at that production in the model file, open a procedural inspector to look at it, or take one step in the stepper to perform that selection and see the details in the stepper. However you choose to look at it, what you should find is that it is binding three different variables to the slots being tested and it is not actually comparing any of them:

```
(P terminate-addition
   =goal>
      ISA         add
      count       =num
      arg2        =num2
      sum         =answer
==>
   =goal>
      isa         add
      count       nil
)
```

So we need to change that so it does the comparison correctly, which means using the same variable for both the count and arg2 tests. If we change the arg2 test to also use =num that should fix the problem. So we should close the stepper, make that change to the model file, save it, reload it, and they try running it again. Of course, we did not necessarily need to go through all of those steps to locate and determine what was wrong because we may have been able to figure that out just from reading the model file, but that is not always the case, particularly for larger and more complex models, so knowing how to work through that process is an important skill to learn.

Before reloading the model, we might also want to change the trace detail level down from high so that it is easier to check if the model does what we expect. Setting it to low will give us a minimal trace, but that should still be sufficient since it will show all the productions that fire. After making that change as well and then reloading here is what the model does when we run it:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
...
     9.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     9.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     9.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     9.950   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
    10.000   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
    10.000   ------                 Stopped because time limit reached
```

Now we have a problem where the increment-sum production fires repeatedly. Again, one could go straight to looking at the model code to try to determine what is wrong, but here we will work through a more rigorous process of stepping through the task and using the diagnostic tools that are available.

As before, the first step should be to turn the trace detail back to high so that we can see all of the details. We can run it now and look at the trace, but we don't need all 10 seconds worth since the problem occurs well before the first second is over. So, we will only run the model up to time 0.300 since that is after the first repeat of increment-sum which we know to be a problem:

```
? (run .3)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.000   PROCEDURAL             PRODUCTION-SELECTED INITIALIZE-ADDITION
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK ONE
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   PROCEDURAL             PRODUCTION-SELECTED INCREMENT-SUM
     0.100   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK ZERO
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   PROCEDURAL             PRODUCTION-SELECTED INCREMENT-SUM
     0.200   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.200   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.250   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.250   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK ZERO
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   PROCEDURAL             PRODUCTION-SELECTED INCREMENT-SUM
     0.300   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.300   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.300   ------                 Stopped because time limit reached
```

As with the last problem, here again the issue looks to be an incorrect production selection since we expect increment-count to follow increment-sum. Thus, it is the selection at time 0.200 which seems to be in error. That is where we will investigate further using the stepper.

To do so we need to reset the model, open the stepper, and then run it again for at least 0.200 seconds. Again, we could use step to advance to where the problem is, but here again the run until button provides us with shortcuts because we can either advance to the selection of increment-sum or directly to the time we are interested in. This time, we will use the time option to skip ahead to the time at which we notice the problem.

Select time as the run until option using the menu button and enter 0.2 in the entry box. Then press "Run Until:" to skip to the first event which occurs at that time. The event we are interested in is not that first event at that time, so we now need to hit the step button a few times to get to the production-selected event at time 0.200. Looking at the details of the production-selected event in the stepper there are actually two things worth noting. The first is of course that increment-sum is selected which we do not want, and the other is that increment-count is not listed under the "Possible Productions" section which lists all of the productions which matched the state and could possibly have been be selected. Thus, while we would expect it to be selected now it did not actually match the current state. Both of those issues will need to be fixed, but first we will correct the issue with increment-sum since that seems more important – there is no point in trying to fix increment-count if increment-sum is still going to fire continuously.

Again, here is where having a thorough design for the model will help us figure out what the problem is since we can compare the production as written to what we intend it to do, but sometimes, particularly while learning how to model with ACT-R, you may not have considered all the possible details in the initial design. Thus, you may have to figure out why the production does not work and adjust your design as well when encountering a problem. Here we will look more at the production itself along with the high-level design instead of just looking at our detailed design specification. The first thing to realize is that since the production is firing again after itself, that means that either its action is not changing the state of the buffers and modules thus it will continue to match or that its condition is not sensitive to any changes which it makes thus allowing it to continuously match (and of course it is also possible that both of those are true). Here is the production from the model for reference:

```
(P increment-sum
   =goal>
      ISA         add
      sum         =sum
      count       =count
    - arg2        =count
   =retrieval>
      ISA         number
      next        =newsum
==>
   =goal>
      ISA         add
      sum         =newsum
   +retrieval>
      ISA         number
      number      =count
)
```

We will start by looking at the action of the production. It modifies the sum slot of the goal to be the next value based on the retrieved chunk, and it requests a retrieval for the chunk corresponding to the current count so that it can be incremented. Those seem to be the correct actions to take for this production and do result in a change to the state of the buffers. Those actions show up in the high detail trace when the model runs, and if we are really concerned we could also step through those actions with the stepper and inspect the buffer contents, but that does not seem necessary at this point. Now we should look at the condition of the production, keeping in mind the changes that its action makes because testing those appropriately is what the production is apparently missing. Looking at the condition of this production we see that it tests the sum slot, which is what gets changed in the action, but it is not actually using that value for anything. Thus, as long as there is any value in that slot this production will fire. Similarly, in the retrieval buffer test of this production there are no constraints on what the chunk in the buffer should look like, only that it have a value in the next slot. The only real constraint specified in the condition of this production is that the count slot's value does not match the arg2 slot's value. Thus, we will have to change something in the condition of this production so that it does not fire again after itself.

Considering our high-level design, it is supposed to fire to increment the sum. Thus, it should only fire when we have retrieved a fact which relates to the sum, but it does not have such a constraint currently. So, we need to add something to it so that it only fires when the retrieved chunk is relevant to the current sum. Given the way the number chunks are set up, what we need to test is that the value in the number slot of the chunk in the retrieval buffer matches the value in the sum slot of the chunk in the goal buffer. Adding that constraint to the production like this:

```
(P increment-sum
   =goal>
      ISA         add
      sum         =sum
      count       =count
    - arg2        =count
   =retrieval>
      ISA         number
      number      =sum
      next        =newsum
==>
   =goal>
      ISA         add
      sum         =newsum
   +retrieval>
      ISA         number
      number      =count
)
```

seems like the right thing to do, and we can now save, load, and retest the model.

Here is the trace we get now:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.000   PROCEDURAL             PRODUCTION-SELECTED INITIALIZE-ADDITION
     0.000   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.050   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK ONE
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   PROCEDURAL             PRODUCTION-SELECTED INCREMENT-SUM
     0.100   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.100   PROCEDURAL             BUFFER-READ-ACTION RETRIEVAL
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK ZERO
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   ------                 Stopped because no events left to process
```

It does not have increment-sum selected and firing again after the first time. So, now we need to determine why increment-count, which we expect to be selected now, is not. Since the model has already stopped where we expect increment-count to be selected we do not need the stepper to get us to that point. All we need to do now is determine why it is not being selected, and to do that we will use the whynot tool again, either from the command prompt or in the procedural viewer. Here is what we get from calling the whynot command for increment-count:

```
> (whynot increment-count)

Production INCREMENT-COUNT does NOT match.
(P INCREMENT-COUNT
   =GOAL>
       SUM =SUM
       COUNT =COUNT
   =RETRIEVAL>
       NUMBER =SUM
       NEXT =NEWCOUNT
 ==>
   =GOAL>
       COUNT =NEWCOUNT
   +RETRIEVAL>
       NUMBER =SUM
)
It fails because:
The value in the NUMBER slot of the chunk in the RETRIEVAL buffer does not satisfy the
constraints.
```

It tells us that it does not match and that one reason for that is because of a mismatch on the number slot of the chunk in the retrieval buffer. We can verify that by looking at the production and the contents of the goal and retrieval buffers. The production's constraint on the number slot is that its value must match the value of the sum slot of the chunk in the goal buffer:

```
 (P increment-count
   =goal>
      ISA         add
      sum         =sum
      count       =count
   =retrieval>
      ISA         number
      number      =sum
      next        =newcount
==>
   =goal>
      ISA         add
      count       =newcount
   +retrieval>
      isa        number
      number     =sum
)
```

here are the chunks in the goal and retrieval buffers:

```
? (buffer-chunk goal retrieval)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ONE
   ARG2  ONE
   SUM  TWO
   COUNT  ZERO

RETRIEVAL: ZERO-0 [ZERO]
ZERO-0
   NUMBER  ZERO
   NEXT  ONE
```

Looking at that it is indeed true that they do not match. Notice however that the number slot's value from the retrieval buffer does match the count slot's value in the goal buffer. Given that this production is trying to increment the count, that is probably what we should be checking instead in this production i.e. that we have retrieved a chunk relevant to the current count. Thus, if we change the production to test the count slot's value instead it might fix the problem:

```
(P increment-count
   =goal>
      ISA         add
      sum         =sum
      count       =count
   =retrieval>
      ISA         number
      number      =count
      next        =newcount
==>
   =goal>
      ISA         add
      count       =newcount
   +retrieval>
      isa        number
      number     =sum
)
```

Along with that change we should probably also change the trace-detail back down to low before saving, loading, and running the next test to make it easier to follow the production sequence. Here is what we see when running the model again along with the chunk in the goal buffer at the end:

```
? (run 1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.300   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     0.300   ------                 Stopped because no events left to process

? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ONE
   ARG2  ONE
   SUM  TWO
```

The goal shows the correct sum for 1+1 and the model performed the sequence of productions that we would expect.

Verification

Before going on and performing more new tests, we should consider whether or not the changes that we have recently made will affect any of the other tests which we have already run i.e. 0+0 and 1+0. In both of those cases the terminate-addition production was fired, and we have had to change that to work correctly to perform the addition of 1+1, so it is a little curious that the “broken” production did those tasks correctly. Thus, to be safe we should probably retest at least one of those to make sure that adding zero still works correctly and was not just a fluke. Here is the result of testing 1+0 again:

```
? (run 1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.100   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     0.100   ------                 Stopped because no events left to process

? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ONE
   ARG2  ZERO
   SUM  ONE
```

Everything looks correct there and given that terminate-addition now works as it was intended we may feel confident enough in the tests so far that we can move on, but if one wants to be cautious, then running the 0+0 test could also be done.

Now that the model has successfully performed three different addition problems we might be tempted to call it complete, but those were all very simple problems and it is supposed to be able to add any numbers from zero to ten which sum to ten or less. So, we should perform some more tests before considering it done.

Test of a large sum

Since our early tests were for small sums it would be useful to also test the other end of the range. There are multiple options for numbers which sum to 10, but if we pick 0+10 that will test both the maximum possible sum as well as also testing the largest number of additions it is expected to be able to do. To run the test we again need to change the goal to represent that problem, save it, load it, and then run it. Here are the trace and resulting goal chunk:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.550   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.650   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.700   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.750   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FOUR
     0.950   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.000   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FOUR
     1.050   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FIVE
     1.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FIVE
     1.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SIX
     1.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SIX
     1.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL EIGHT
     1.550   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL EIGHT
     1.650   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.700   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL NINE
     1.750   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL NINE
     1.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TEN
     1.900   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     1.900   ------                 Stopped because no events left to process

? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ZERO
   ARG2  TEN
   SUM  TEN
```

The goal chunk is correct with a sum of ten, and thus one might think that it was a successful test. However, if we look at the trace more carefully we will see that something is not quite right. Since the count was ten we would expect to see ten firings of each of increment-sum and increment-count, but the model only fires each nine times. So, there is something else wrong in the model, because even though it got the right answer it did not get there the right way. As with all of the other problems, one could just immediately start looking at the model code to try to find the issue, but here again we will walk through a more rigorous approach.

To determine what went wrong along the way we will walk through the model with the stepper and watch the chunks in the goal and retrieval buffers as the model progresses. For this test we can leave the trace-detail at low for a first pass because that will require fewer steps through the task, and only if we do not find a problem at that level will we move it up to a higher level.

Reset the model and open the stepper along with two buffers windows, one for the goal and one for retrieval. Now run the model and start stepping through the actions watching the changes which occur in the two buffers as it goes. Everything starts off well with the sum and count both incrementing by one each time as the model goes along. However, after executing the event at time 1.300 we see something wrong in the retrieval buffer. The chunk that is retrieved has six in the number slot and eight in the next slot. If we continue to step through the model's actions we see that increment-sum uses that chunk to incorrectly increment the sum from six to eight, and then that chunk is retrieved again and increment-count also skips over the number seven as it goes. So, we need to correct the chunk named six in the model's declarative memory so that it goes from six to seven instead of six to eight. Had we only looked at the result in the goal chunk we would not have noticed this problem. We may have caught it with other tests, but when running a test it is best to make sure that it is completely successful before moving on to test other values.

To correct the problem we need to change the chunk six. Looking at the other declarative memory chunks there is already a chunk for seven, but the chunk for six incorrectly indicates that eight is the next value. We need to change that to seven instead:

```
(add-dm
 (zero isa number number zero next one)
 (one isa number number one next two)
 (two isa number number two next three)
 (three isa number number three next four)
 (four isa number number four next five)
 (five isa number number five next six)
 (six isa number number six next seven)
 (seven isa number number seven next eight)
 (eight isa number number eight next nine)
 (nine isa number number nine next ten)
 (ten isa number number ten)
 (test-goal ISA add arg1 zero arg2 ten))
```

If we save that and run it again we get this trace and resulting goal chunk which shows the correct sum and which takes the correct number of steps to get there:

```
? (run 10)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TEST-GOAL NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ONE
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.550   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.650   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.700   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.750   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FOUR
     0.950   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.000   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FOUR
     1.050   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FIVE
     1.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FIVE
     1.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SIX
     1.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SIX
     1.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SEVEN
     1.550   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL SEVEN
     1.650   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.700   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL EIGHT
     1.750   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL EIGHT
     1.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     1.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL NINE
     1.950   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     2.000   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL NINE
     2.050   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     2.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TEN
     2.100   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
     2.100   ------                 Stopped because no events left to process

? (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
   ARG1  ZERO
   ARG2  TEN
   SUM  TEN
```

Now that we have successfully tested the other extreme we may feel more confident that the model works correctly, but we should probably test a few sums in the middle of the range just to be certain before calling it complete. Some values that seem worthwhile for testing would be things like 3+4 since we have recently added a chunk for seven to make sure that it is correct, and similarly 7+1 and 1+7 might be good tests to perform to make sure our new chunk gets used correctly. Another test that may be useful would be 5+5 because it both counts to the maximum sum and checks whether the model works correctly for matching sum and count values.

We will not work through those tests here, but you should perform some of those, as well as others that you choose for additional practice in testing and verifying results. In testing the model further you should find a curious situation for some types of addition problems. In those problems the model will produce the correct answer in the intended way, but a thorough inspection will show that it had the possibility to do things wrong along the way. Why it always does the correct thing is beyond the scope of this unit, but issues like that will be addressed in later units.

Unit 2: Perception and Motor Actions in ACT-R

2.1 ACT-R Interacting with the World

This unit will introduce some of the mechanisms which allow ACT-R to interact with the world, which for the purposes of the tutorial will be experiments presented via the computer. This is made possible with the addition of perceptual and motor modules which were originally developed by Mike Byrne as a separate system called ACT-R/PM but which are now an integrated part of the ACT-R architecture. These additional modules provide a model with visual, motor, auditory, and vocal capabilities based on human performance, and also include mechanisms for interfacing those modules to the world. The auditory, motor, and vocal modules are based upon the corresponding components of the EPIC architecture developed by David Kieras and David Meyer, but the vision module is based on visual attention work which was done in ACT-R. The interface to the world which we will use in the tutorial allows the model to interact with the computer i.e. process visual items presented, press keys, and move and click the mouse, for tasks which are created using tools built into ACT-R for generating user interfaces. It is also possible for one to extend that interface or implement new interfaces to allow models to interact with other environments, but that is beyond the scope of the tutorial.

2.1.1 Experiments

From this point on in the tutorial most of the models will be performing an experiment or task which requires interacting with the world in some way. That means that one will also have to run the corresponding task for the model to interact with in addition to running the model. All of the tasks for the tutorial models have been written in both ANSI Common Lisp and Python 3. The tutorial will show how to run both versions of the task, and the experiment description document in each unit will describe the code which implements both versions of all of the tasks. From the model's perspective, it does not matter which implementation of the task is used, and the single model file included with the tutorial can be run with either version producing the same results.

2.2 The First Experiment

The first experiment is very simple and consists of a display in which a single letter is presented. The participant's task is to press the key corresponding to that letter. When any key is pressed, the display is cleared and the experiment ends. This experiment can be run for either a human participant or for an ACT-R model to perform, and the model in the demo2-model.lisp file in the tutorial is able to perform this task. If you wish to run the task for a human participant then you must have an ACT-R experiment window viewer running to see and interact with the task, and the ACT-R Environment includes such a viewer which will be available as long as you do not close the Control Panel window.[1]

[1] There is also an additional application included with the ACT-R software which will allow the experiment windows to be used through a browser. The readme file with the ACT-R standalone software contains instructions on how to

2.2.1 Running the Lisp version

The first thing you will need to do to run the Lisp version of the experiment is load the experiment code. That can be done using the "Load ACT-R code" button in the Environment just as you loaded the model files in unit 1. The file is called demo2.lisp and is found in the lisp directory of the ACT-R tutorial. When you load that file it will also load a model which can perform the task from the demo2-model.lisp file in the tutorial/unit2 directory, and if you look at the top of the Control Panel you will see that it says DEMO2 under "Current Model". To run the experiment you will call the demo2-experiment function, and it has one optional parameter which indicates whether a human participant will be performing the task. As a first run you should perform the task yourself, and to do that you will evaluate the **demo2-experiment** function at the prompt in the ACT-R window and pass it a value of t (the Lisp symbol representing true):

```
? (demo2-experiment t)
```

When you enter that a window titled "Letter recognition" will appear with a letter in it (the window may be obscured by other open windows so you may have to arrange things to ensure you can see everything you want). When you press a key while that experiment window is the active window the experiment window will clear and that is the end of the experiment. The letter you typed will be returned by the **demo2-experiment** function.

2.2.2 Running the Python version

To run the Python version you should first run an interactive Python session on your machine (instructions on how to do that are not part of this tutorial and you will need to consult the Python documentation for details). Then you can import the demo2 module which can be found in the python directory of the ACT-R tutorial. For the examples in this tutorial we will assume that that directory is the current directory for the Python session or that directory has been added to the Python search path, and we will also assume that the Python prompt is the three character sequence ">>>". When you import that module it will first output a line indicating that it has connected your Python session to the ACT-R software running on your machine:

```
>>> import demo2
ACT-R connection has been started.
```

The demo2 module will also have ACT-R load the model for this task, which is found in the demo2-model.lisp file in the tutorial/unit2 directory, and if you look at the top of the Control Panel you will see that it now says DEMO2 under "Current Model". To run the experiment you will call the **experiment** function in that module, and it has one optional parameter which indicates whether a human participant will be performing the task. As a first run you should

run that if you do not want to use the ACT-R Environment application.

perform the task yourself, and to do that you will call the **experiment** function at the Python prompt and pass it the value True:

```
>>> demo2.experiment(True)
```

When you enter that a window titled "Letter recognition" will appear with a letter in it (the window may be obscured by other open windows so you may have to arrange things to ensure you can see everything you want). When you press a key while that experiment window is the active window the experiment window will clear and that is the end of the experiment. The letter you typed will be returned by the **experiment** function.

2.2.3 Running the model in the experiment

You can run the model through the experiment by calling the function without including the true value which indicates a human participant. That would look like this for the two different versions:

```
? (demo2-experiment)
```

or

```
>>> demo2.experiment()
```

Regardless of which version you run, you will see the following trace of the model performing the task:

```
0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL NIL
0.000   VISION         PROC-DISPLAY
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.000   PROCEDURAL     PRODUCTION-SELECTED FIND-UNATTENDED-LETTER
0.000   PROCEDURAL     BUFFER-READ-ACTION GOAL
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-UNATTENDED-LETTER
0.050   PROCEDURAL     MOD-BUFFER-CHUNK GOAL
0.050   PROCEDURAL     MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION         Find-location
0.050   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-SELECTED ATTEND-LETTER
0.050   PROCEDURAL     BUFFER-READ-ACTION GOAL
0.050   PROCEDURAL     BUFFER-READ-ACTION VISUAL-LOCATION
0.050   PROCEDURAL     QUERY-BUFFER-ACTION VISUAL
0.100   PROCEDURAL     PRODUCTION-FIRED ATTEND-LETTER
0.100   PROCEDURAL     MOD-BUFFER-CHUNK GOAL
0.100   PROCEDURAL     MODULE-REQUEST VISUAL
0.100   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.100   PROCEDURAL     CLEAR-BUFFER VISUAL
0.100   VISION         Move-attention VISUAL-LOCATION0-1 NIL
```

```
0.100   PROCEDURAL     CONFLICT-RESOLUTION
0.185   VISION         Encoding-complete VISUAL-LOCATION0-1 NIL
0.185   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     CONFLICT-RESOLUTION
0.185   PROCEDURAL     PRODUCTION-SELECTED ENCODE-LETTER
0.185   PROCEDURAL     BUFFER-READ-ACTION GOAL
0.185   PROCEDURAL     BUFFER-READ-ACTION VISUAL
0.185   PROCEDURAL     QUERY-BUFFER-ACTION IMAGINAL
0.235   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.235   PROCEDURAL     MOD-BUFFER-CHUNK GOAL
0.235   PROCEDURAL     MODULE-REQUEST IMAGINAL
0.235   PROCEDURAL     CLEAR-BUFFER VISUAL
0.235   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.235   PROCEDURAL     CONFLICT-RESOLUTION
0.435   IMAGINAL       CREATE-NEW-BUFFER-CHUNK IMAGINAL
0.435   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.435   PROCEDURAL     CONFLICT-RESOLUTION
0.435   PROCEDURAL     PRODUCTION-SELECTED RESPOND
0.435   PROCEDURAL     BUFFER-READ-ACTION GOAL
0.435   PROCEDURAL     BUFFER-READ-ACTION IMAGINAL
0.435   PROCEDURAL     QUERY-BUFFER-ACTION MANUAL
0.485   PROCEDURAL     PRODUCTION-FIRED RESPOND
0.485   PROCEDURAL     MOD-BUFFER-CHUNK GOAL
0.485   PROCEDURAL     MODULE-REQUEST MANUAL
0.485   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.485   PROCEDURAL     CLEAR-BUFFER MANUAL
0.485   MOTOR          PRESS-KEY KEY v
0.485   PROCEDURAL     CONFLICT-RESOLUTION
0.735   MOTOR          PREPARATION-COMPLETE 0.485
0.735   PROCEDURAL     CONFLICT-RESOLUTION
0.785   MOTOR          INITIATION-COMPLETE 0.485
0.785   PROCEDURAL     CONFLICT-RESOLUTION
0.885   KEYBOARD       output-key DEMO2 v
0.885   VISION         PROC-DISPLAY
0.885   VISION         visicon-update
0.885   PROCEDURAL     CONFLICT-RESOLUTION
0.970   VISION         Encoding-complete VISUAL-LOCATION0-1 NIL
0.970   VISION         No visual-object found
0.970   PROCEDURAL     CONFLICT-RESOLUTION
1.035   MOTOR          FINISH-MOVEMENT 0.485
1.035   PROCEDURAL     CONFLICT-RESOLUTION
1.035   ------         Stopped because no events left to process
```

Unlike the previous unit where we had to run the model explicitly using the Run button on the Control Panel, the code which implements this experiment automatically runs the model to do the task. It is not necessary that it operate that way, but it is often convenient to build the experiments for the models to do so, particularly in tasks like those used in later units where we will be running the models through the experiments many times to collect performance measures.

Looking at that trace we see production firing being intermixed with actions of the vision, imaginal, and motor modules as the model encodes the stimulus and issues a response. If you watch the window while the model is performing the task you will also see a red circle drawn. That is a debugging aid which indicates the model's current point of visual attention, and can be turned off if you do not want to see it. You may also notice that the task always presents the letter "V". That is done so that it always generates the same trace. In the following sections we will look at how the model perceives the letter being presented, how it issues a response, and then briefly discuss some parameters in ACT-R that control things like the attention marker and the pseudo-random number generator.

2.3 Control and Representation

Before looking at the details of the new modules used in this unit we will first look at a difference in how the information for the task is represented compared to the unit 1 models. If you open the demo2-model.lisp file and look at the model definition you will find two chunk-types created for this model:

```
(chunk-type read-letters state)
(chunk-type array letter)
```

The chunk-type read-letters specifies one slot which is called state and will be used to track the current task state for the model. The other chunk-type, array, also has only one slot, which is called letter, and will hold a representation of the letter which is seen by the model.

In unit 1, the chunk that was placed into the **goal** buffer had slots which held all of the information relevant to performing the task. That approach is how ACT-R models were typically built in older versions of the architecture, but now a more distributed representation of the model's task information across two buffers is the recommended approach to modeling with ACT-R. The **goal** buffer should be used to hold control state information – the internal representation of what the model is doing and where it is in the task. A different buffer, the **imaginal** buffer, should be used to hold the chunk which contains the current problem state information – the information needed to perform the task. In the demo2 model, the **goal** buffer will hold a chunk based on the read-letters chunk-type, and the **imaginal** buffer will hold a chunk based on the array chunk-type.

2.3.1 The State Slot

In this model, the state slot of the chunk in the **goal** buffer will maintain information about what the model is doing, and it is used to explicitly indicate which productions are appropriate at any time. This is often done when writing ACT-R models because it provides an easy means of specifying an ordering of the productions and it can make it easier to understand the way the model operates by looking at the productions. It is however not always necessary to do so, and there are other means by which the same control flow can be accomplished. In fact, as we will see in a later unit there can be consequences to keeping extra information in a buffer. However, because it does make the production sequencing in a model clearer you will see a slot named state (or something similar) in many of the models in the tutorial. As an additional challenge for this unit, you should try to modify the demo2 model so that it works without needing to maintain an explicit state marker and thus not need to use the **goal** buffer at all.

2.4 The Imaginal Module

The first new module we will describe in this unit is the imaginal module. This module has a buffer called **imaginal** which is used to create new chunks. These chunks will be the model's internal representation of information – its internal image (hence the name). Like any buffer, the chunk in the **imaginal** buffer can be modified by the productions to build that representation using RHS modification actions as shown in unit 1.

An important issue with the **imaginal** buffer is how a chunk first gets into the buffer. Unlike the **goal** buffer's chunk which we have been creating and placing there in advance of the model starting, the imaginal module will create the chunk for the **imaginal** buffer in response to a request from a production.

All requests to the imaginal module through the **imaginal** buffer are requests to create a new chunk. The imaginal module will create a new chunk using the slots and values provided in the request and place that chunk into the **imaginal** buffer. An example of this is shown in the action of the encode-letter production of the demo2 model:

```
(P encode-letter
   ...
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
)
```

We will come back to the condition of that production later. For now, we are interested in this request on the RHS:

```
  +imaginal>
      isa         array
      letter      =letter
```

This request of the **imaginal** buffer is asking the imaginal module to create a chunk which has a slot named letter that has the value of the variable =letter. We see the request and its results in these lines of the trace:

```
0.235   PROCEDURAL    PRODUCTION-FIRED ENCODE-LETTER
...
0.235   PROCEDURAL    MODULE-REQUEST IMAGINAL
...
0.235   PROCEDURAL    CLEAR-BUFFER IMAGINAL
...
```

```
0.435   IMAGINAL       CREATE-NEW-BUFFER-CHUNK IMAGINAL
0.435   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
```

When the encode-letter production fires it makes that request and automatically clears the buffer at that time as happens for all buffer requests. Then, we see that the imaginal module reports that it is creating a new chunk and that chunk is then placed into the buffer.

Something to notice in the trace is that the chunk was not immediately placed into the buffer as a result of the request. It took .2 seconds before the chunk was made available. This is an important aspect of the imaginal module – it takes time to build a representation. The amount of time that it takes the imaginal module to create a chunk is a fixed cost, and the default time is .2 seconds (that can be changed with a parameter). In addition to the time cost, the imaginal module is only able to create one new chunk at a time. That does not affect this model because it is only creating the one new chunk in the **imaginal** buffer, but in models which require a richer representation that bottleneck may be a constraint on how fast the model can perform the task. In such situations one should first verify that the module is available to create a new chunk before making a request. That is done with a query of the buffer on the LHS, as is done in this production to check that the state of the module is free:

```
(P encode-letter
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
)
```

Additional information about querying modules will be described later in the unit.

In this model, the **imaginal** buffer will hold a chunk which contains a representation of the letter which the model reads from the screen. For this simple task, that representation is not necessary because the model could use the information directly from the **visual** buffer to do the task, but for most tasks there will be more than one piece of information which must be acquired incrementally which requires storing the intermediate values as it progresses.

2.5 The Vision Module

Many tasks involve interacting with visible stimuli and the vision module provides the model with a means for acquiring visual information. It is designed as a system for modeling visual attention that assumes there are lower-level perceptual processes that generate the representations with which it operates, but it does not model those perceptual processes in detail. The experiment generation tools in ACT-R create tasks which can provide representations of text, lines, and button features from the displays it creates to the vision module. The ability to provide visual feature information to the vision module is also available to modelers for creating new visual features, but that is beyond the scope of the tutorial.

The vision module has two buffers. There is a **visual** buffer that holds a chunk which represents an object in the visual scene and a **visual-location** buffer that holds a chunk which represents the location of an object in the visual scene. Visual interaction is shown in the demo2 model in the two productions **find-unattended-letter** and **attend-letter**.

2.5.1 Visual-Location buffer

The **find-unattended-letter** production applies whenever the **goal** buffer's chunk has the value start in the state slot (which is how the chunk is initially created in the model file):

```
(P find-unattended-letter
   =goal>
      ISA         read-letters
      state       start
 ==>
   +visual-location>
      :attended    nil
   =goal>
      state       find-location
)
```

It makes a request of the **visual-location** buffer and it changes the **goal** buffer chunk's state slot to the value find-location. It is important to note that those values for the state slot of that chunk are arbitrary. The values used in this model were chosen to help make clear what the model is doing, but the model would continue to operate the same if all of the corresponding references were consistently changed to other values instead.

A **visual-location** request asks the vision module to find the location of an object in its visual scene (which for this model is the current experiment's window) that meets the specified requirements, build a chunk to represent the location of that object if one exists, and place that chunk in the **visual-location** buffer.

The following portion of the trace reflects the actions performed by this production:

```
0.050   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.050   PROCEDURAL    MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION        Find-location
0.050   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0
```

We see the notice of the **visual-location** request and the automatic clearing of the **visual-location** buffer due to the request being made by the production. Then the vision module reports that it is finding a location, and after that it places a chunk into the buffer. Notice that there was no time involved in handling the request – all those actions took place at time 0.050 seconds. The **visual-location** requests always finish immediately which reflects the assumption that there is a perceptual system within the vision module operating in parallel with the procedural module that can make these visual features immediately available.

If you run the model through the task again and step through the model's actions using the Stepper you can use the "Buffers" tool to see that the chunk **visual-location0-1** will be in the **visual-location** buffer after that last event:

```
VISUAL-LOCATION0-1
   KIND  TEXT
   VALUE  TEXT
   COLOR  BLACK
   HEIGHT  10
   WIDTH  7
   SCREEN-X  430
   SCREEN-Y  456
   DISTANCE  1080
   SIZE  0.19999999
```

There are a lot of slots in the chunk placed into the **visual-location** buffer, and when making the request to find a location all of those features can be used to constrain the results. None of them are important for this unit, but we will describe the **screen-x** and **screen-y** slots here. They encode the position of the object in the visual scene. For the experiment windows created by the ACT-R interface tools the upper-left corner of the screen is **screen-x** 0 and **screen-y** 0. The x coordinates increase from left to right, and the y coordinates increase from top to bottom. The value for a feature in the experiment window is the position of the experiment window's upper-left corner on the screen plus the position of the item in the window – it is the global position on the whole screen not the relative position within the window. Typically, the specific values are not that important for the model and do not need to be specified when making a request for a location. There is a set of descriptive specifiers that can be used for requests on those slots, like lowest or highest, but those details will not be discussed until unit 3.

2.5.1.1 The attended request parameter

If we look at the request which was made of the **visual-location** buffer in the **find-unattended-letter** production:

```
+visual-location>
     :attended   nil
```

we see that all it consists of is ":attended nil" in the request. This **:attended** specification is called a request parameter. It acts like a slot in the request, but does not correspond to a slot in the chunk which was created. A request parameter is valid for any request to a buffer regardless of any chunk-type that is specified (or when no chunk-type is specified as is the case here). Request parameters are used to supply general information to the module about a request which may not correspond to any information that would be included in a chunk that is placed into a buffer. A request parameter is specific to a particular buffer and will always start with a ":" which distinguishes it from an actual slot of some chunk-type.

For a visual-location request one can use the :**attended** request parameter to specify whether the vision module should try to find the location of an object which the model has previously looked at (attended to) or not. If it is specified as **nil**, then the request is for a location which the model has not attended, and if it is specified as **t**, then the request is for a location which has been attended previously. There is also a third option, **new**, which means that the model has not attended to the location and that the object has also recently appeared in the visual scene.

2.5.2 The attend-letter production

The **attend-letter** production applies when the goal state is find-location, there is a chunk in the **visual-location** buffer, and the vision module is not currently active with respect to the **visual** buffer:

```
(P attend-letter
   =goal>
      ISA         read-letters
      state       find-location
   =visual-location>
   ?visual>
      state       free
==>
   +visual>
      cmd         move-attention
      screen-pos  =visual-location
   =goal>
      state       attend
)
```

On the LHS of this production are two tests that have not been used in previous models. The first of those is a test of the **visual-location** buffer which has no constraints specified for the slots of the chunk in that buffer. All that is necessary for this production is that there is a chunk in the buffer and the details of its slot values do not matter. The other is a query of the **visual** buffer.

2.5.3 Checking a module's state

On the LHS of **attend-letter** a query is made of the **visual** buffer to test that the **state** of the vision module is **free**. All buffers will respond to a query for their module's **state** and the possible values for that query are **busy, free,** or **error** as was shown in unit 1. The test of **state free** is a check to make sure the buffer being queried is available for a new request through the indicated buffer. If the **state** is **free**, then it is safe to issue a new request through that buffer, but if it is **busy** then it is usually not safe to do so.

Typically, a module is only able to handle one request to a buffer at a time, and that is the case for both the **imaginal** and **visual** buffers which require some time to produce a result. Since all of the model's modules operate in parallel it might be possible for the procedural module to select a production which makes a request to a module that is still working on a previous request. If a production were to fire at such a point and issue a request to a module which is currently busy and only able to handle one request at a time, that is referred to as "jamming" the module. When a module is jammed, it will output a warning message in the trace to let you know what has happened. What a module does when jammed varies from module to module. Some modules ignore the new request, whereas others abandon the previous request and start the new one. As a general practice it is best to avoid jamming modules. Thus, when there is the possibility of jamming a module one should query its state before making a request.

Note that we did not query the state of the **visual-location** buffer in the **find-unattended-letter** production before issuing the **visual-location** request. That is because we know that those requests always complete immediately and thus the **state** of the vision module for the **visual-location** buffer is always **free**. We did however test the state of the imaginal module before making the request to the **imaginal** buffer in the **encode-letter** production. That query is not necessary in this model and removing it will not change the way the model performs. That is because that is the only request to the imaginal module in the model and that production will not fire again because of the change to the **goal** buffer chunk's state slot. Thus there is no risk of jamming the imaginal module in this model, but omitting queries which appear to be unnecessary is a risky practice. It is always a good idea to query the state in every production that makes a request that could potentially jam a module even if you think that it will not happen because of the structure of the other productions in the current model. Doing so makes it clear to anyone else who may read the model, and it also protects you from problems if you decide later to apply that model to a different task where the assumption which avoids the jamming no longer holds.

In addition to the state, there are also other queries that one can make of a buffer. Unit 1 presented the general queries that are available to all buffers. Some buffers also provide queries that are specific to the details of the module and those will be described as needed in the tutorial. To see the status of all the information which can be queried for a buffer you can use the "Buffers" tool in the Control Panel which was used in the last unit to display information about the chunk in a buffer. The see the status of a buffer press the button labeled "Status" at the top right on the buffer viewer. That will change the information shown from the contents of the buffer to its status details. Those details show the standard queries for each buffer along with the current value (either **t** or **nil**) for such a query at this time along with any additional queries which may be specific to that buffer and which may take a slightly different form (details on all of the queries for each buffer can be found in the reference manual).

2.5.4 Chunk-type for the visual-location buffer

You may have noticed that we did not specify a chunk-type with either the request to the **visual-location** buffer in the **find-unattended-letter** production or its testing in the condition of the **attend-letter** production. That was because we didn't specify any slots in either of those places (recall that :attended is a request parameter and not a slot) thus there is no need to specify a chunk-type for verification that the slots used are correct. If we did need to request or test specific features with the **visual-location** buffer there is a chunk-type named visual-location which one can use to do so which has the slots screen-x, screen-y, distance, kind, color, value, height, width, and size.

2.5.5 Visual buffer

On the RHS of **attend-letter** it makes a request of the **visual** buffer which specifies two slots: cmd and screen-pos:

```
   +visual>
      cmd         move-attention
      screen-pos  =visual-location
```

Unlike the other buffers which we have seen so far, the **visual** buffer is capable of performing different actions in response to a request. The cmd slot in a request to the **visual** buffer indicates which specific action is being requested. In this request that is the value move-attention which indicates that the production is asking the vision module to move its attention to some location, create a chunk which encodes the object that is there, and place that chunk into the **visual** buffer. The location to which the module should move its attention is specified in the screen-pos (short for screen-position) slot of the request. In this production that location is the chunk that is in the **visual-location** buffer. The following portion of the trace shows this request and the results:

```
0.100   PROCEDURAL    PRODUCTION-FIRED ATTEND-LETTER
...
0.100   PROCEDURAL    MODULE-REQUEST VISUAL
...
0.100   PROCEDURAL    CLEAR-BUFFER VISUAL
0.100   VISION        Move-attention VISUAL-LOCATION0-1 NIL
...
0.185   VISION        Encoding-complete VISUAL-LOCATION0-1 NIL
0.185   VISION        SET-BUFFER-CHUNK VISUAL TEXT0
```

The request to move-attention is made at time 0.100 seconds and the vision module reports receiving that request at that time as well. Then 0.085 seconds pass before the vision module reports that it has completed encoding the object at that location, and it places a chunk into the **visual** buffer at time 0.185 seconds. Those 85 ms represent the time to shift attention and create the visual object. Altogether, counting the two production firings (one to request the location and one to request the attention shift) and the 85 ms to execute the attention shift and object encoding, it takes 185 ms to create the chunk that encodes the letter on the screen.

If you step through the model you will find this chunk in the **visual** buffer after those actions have occurred:

```
TEXT0-0
   SCREEN-POS  VISUAL-LOCATION0-0
   VALUE  "v"
   COLOR  BLACK
   HEIGHT  10
   WIDTH  7
   TEXT  T
```

Most of the chunks created for the **visual** buffer by the vision module are going to have a common set of slots. Those will include the **screen-pos** slot which holds the location chunk which represents where the object is located (which will typically have the same information as the location to which attention was moved) and then **color**, **height**, and **width** slots which hold information about the visual features of the object that was attended. In addition, depending on the details of the object which was attended, other slots may provide more details. When the object is text from an experiment window, the **value** slot will hold a string that contains the text encoded from the screen. As seen above for the text item there is also a slot named **text** which has the value t (the Lisp truth symbol) to indicate that the item is classified as text. Information about the chunk-types available for visual items will be described below.

After a chunk has been placed in the **visual** buffer this model harvests that chunk with the **encode-letter** production:

```
(P encode-letter
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
)
```

which makes a request to the **imaginal** buffer to create a new chunk which will hold a representation of the letter as was described in the section above on the imaginal module.

2.5.6 Chunk-types for the visual buffer

As with the **visual-location** buffer you may have noticed that we also didn’t specify a chunk-type with the request of the **visual** buffer in the **attend-letter** production or the harvesting of the chunk in the **encode-letter** production. Unlike the **visual-location** buffer, there actually are slots specified in both of those cases for the **visual** buffer. Thus, for added safety we should have specified a chunk-type in both of those places to validate the slots being used.

For the chunks placed into the **visual** buffer by the vision module when attending to an item there is a chunk-type called visual-object which provides these slots: screen-pos, value, status, color, height, width, and distance. For the objects which the vision module receives from the experiment window interface the visual-object chunk-type is always an acceptable choice. Therefore a safer specification of the **encode-letter** production would include the chunk-type declaration shown here in red:

```
(P encode-letter
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      isa         visual-object
      value       =letter
   ?imaginal>
      state       free
==> ...)
```

For the request to the **visual** buffer there are a couple of options available for how to include a chunk-type declaration to verify the slots. The first option is to use the chunk-type named vision-command which includes all of the slots for all of the requests which can be made to the **visual** buffer:

```
   +visual>
      isa         vision-command
      cmd         move-attention
      screen-pos  =visual-location
```

Because that contains slots for all of the different requests which the **visual** buffer can handle it is not as safe as a more specific chunk-type which only contains the slots for the specific command being used. For the move-attention command there is another chunk-type called move-attention which only contains the slots that are valid for the move-attention request. Therefore, a more specific declaration for the request to the **visual** buffer for the vision module to move attention to an object would be:

```
   +visual>
```

```
    isa          move-attention
    cmd          move-attention
    screen-pos   =visual-location
```

Specifying move-attention twice in that request looks a little awkward. There is a way to avoid that redundancy and still maintain the safety of the chunk-type declaration, but we will not be describing that until later in the tutorial.

2.5.7 Other Vision module actions

If you look closely at the trace you will find that there are seven other events which are attributed to the vision module that we have not yet described:

```
0.000   VISION          PROC-DISPLAY
0.000   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION          visicon-update
...
0.885   VISION          PROC-DISPLAY
0.885   VISION          visicon-update
...
0.970   VISION          Encoding-complete VISUAL-LOCATION0-1 NIL
0.970   VISION          No visual-object found
```

These actions represent activity which the vision module has performed without being requested to do so. The ones which indicate actions of proc-display and visicon-update are notices of internal updates which may be useful for the modeler to know, but are not directly relevant to the model itself. The proc-display actions indicate that something has happened in the world which has caused the vision module to reprocess the information which is available to it. The one at time 0 happens because that is when the model first begins to interact with the display, and the one at time .885 occurs because when the model pressed the key the screen was erased. Those are followed at the same times (though not necessarily immediately) by actions which say visicon-update. The visicon-update action is an indication that the reprocessing of the visual information resulted in an actual change to the information available in the vision module.

The other event at time 0 is the result of a mechanism in the vision module which we will not discuss until the next tutorial unit, so for now you should just ignore it. The actions at time .970 will be described in the next section.

2.5.8 Visual Re-encoding

The two lines in the trace of the model at time .970 performed by the vision module were not the result of a request in a production:

```
0.970   VISION          Encoding-complete VISUAL-LOCATION0-1 NIL
0.970   VISION          No visual-object found
```

The first of those is an encoding-complete action as we saw above when the module was requested to move-attention to a location. This encoding-complete is triggered by the proc-display which occurred at time 0.885 in response to the screen being cleared after the key press. If the vision module is attending to a location when it reprocesses the visual scene it will automatically re-encode the information at the attended location to encode any changes that may have occurred there. This re-encoding takes 85 ms just as an explicit request to attend an item does. If the visual-object chunk representing that item is still in the **visual** buffer it will be updated to reflect any changes. If there is no longer a visual item on the display at the location where the model is attending (as is the case here) then the trace will show a line indicating that no object was found. That will result in the vision module noting a failure in the **visual** buffer and a **state** of **error** through the **visual** buffer until there is another successful encoding (very much like a memory retrieval failure in the **retrieval** buffer).

This automatic re-encoding process of the vision module can require that you be careful when writing models that process changing displays for two reasons. The first is that you cannot be guaranteed that the chunk in the **visual** buffer will not change in response to a change in the visual display. The other is because while the re-encoding is occurring, the vision module is **busy** and cannot handle a new attention shift. This is one reason it is important to query the visual **state** before all visual requests to avoid jamming the vision module – there may be activity other than that which is requested explicitly by the productions.

2.5.9 Stop Visually Attending

If you do not want the model to re-encode the information at the location it is currently attending you need to have it stop attending to the visual scene. That is done by issuing a clear command to the vision module as an action:

```
+visual>
   cmd clear
```

This request will cause the model to stop attending to any visual items until a new request to move-attention is made and thus it will not re-encode items if the visual scene changes.

If one wants the safety of a chunk-type declaration with that, then as indicated above the vision-command chunk-type could be used:

```
+visual>
   isa vision-command
   cmd clear
```

2.6 Learning New Chunks

This process of seeking the location of an object in one production, switching attention to the object in a second production, and harvesting the object in a third production is a common process in ACT-R models for handling perceptual information. Something to keep in mind about that processing is that this is one way in which ACT-R can acquire new declarative chunks. As was noted in the previous unit, the declarative memory module stores the chunks which are cleared from buffers, and that includes the perceptual buffers. Thus, as those perceptual chunks are cleared from their buffers, they will be recorded in the model's declarative memory.

2.7 The Motor Module

When we speak of motor actions in ACT-R we are only concerned with hand movements. It is possible to extend the motor module to other modes of action, but the provided motor module is built around controlling a pair of hands. In this unit the model will only be performing finger presses on a virtual keyboard, but there are also actions available for the model's hands to use a virtual mouse or joystick. It is also possible for the modeler to create additional motor capabilities and new devices to interact with, but that is beyond the scope of the tutorial.

The buffer for interacting with the motor module is called the **manual** buffer. Unlike other buffers however, the **manual** buffer will not have any chunks placed into it by its module. It is only used to issue commands and to query the state of the motor module. As with the vision module, you should always check to make sure that the motor module is **free** before making any requests to avoid jamming it. The **manual** buffer query to test the state of the module works the same as the one described for the vision module:

```
?manual>
    state free
```

That query will be true when the module is available.

The motor module actually has a more complex set of internal states than just **free** or **busy** because there are multiple stages in performing the motor actions. By testing those internal states it is possible to make a new request to the motor module before the previous one has fully completed if it does not conflict with the ongoing action. However we will not be discussing the details of those internal states in the tutorial, and testing the overall state of the module will be sufficient for performing all of the tasks used in the tutorial.

The **respond** production from the demo2 model shows the **manual** buffer in use:

```
(P respond
   =goal>
      ISA         read-letters
      state       respond
   =imaginal>
      isa         array
      letter      =letter
   ?manual>
```

```
      state       free
==>
   =goal>
      state       done
   +manual>
      cmd         press-key
      key         =letter
)
```

This production fires when a letter has been encoded in the **imaginal** buffer, the goal state slot has the value respond, and the **manual** buffer indicates that the motor module is free. A request is made to press the key corresponding to the letter from the letter slot of the chunk in the **imaginal** buffer and the state slot of the chunk in the **goal** buffer is changed to done.

Because there are many different actions which the motor module is able to perform, when making a request to the **manual** buffer a slot named cmd is used to indicate which action to perform. The **press-key** action used here assumes that the model's hands are located over the home row on the keyboard (which they are by default when using the provided experiment interface). From that position a press-key request will move the appropriate finger to touch type the character specified in the key slot of the request and then return that finger to the home row position.

Here are the events related to the **manual** buffer request from that production firing:

```
0.485   PROCEDURAL    PRODUCTION-FIRED RESPOND
...
0.485   PROCEDURAL    MODULE-REQUEST MANUAL
...
0.485   PROCEDURAL    CLEAR-BUFFER MANUAL
0.485   MOTOR         PRESS-KEY KEY v
...
0.735   MOTOR         PREPARATION-COMPLETE 0.485
...
0.785   MOTOR         INITIATION-COMPLETE 0.485
...
0.885   KEYBOARD      output-key DEMO2 v
...
1.035   MOTOR         FINISH-MOVEMENT 0.485
```

When the production is fired at time 0.485 seconds a request is made to press the key, the buffer is automatically cleared (even though the motor module does not put chunks into its buffer the procedural module still performs the clear action), and the motor module indicates that it has received a request to press the "v" key. However, it takes 250ms to prepare the features of the movement (preparation-complete), 50ms to initiate the action (initiation-complete), another 100ms until the key is actual struck and detected by the keyboard (output-key), and finally it takes another 150ms for the finger to return to the home row and be ready to move again (finish-movement). Thus the time of the key press is at 0.885 seconds, however the motor module is still busy until time 1.035 seconds. The numbers shown after the motor actions of preparation-complete, initiation-complete, and finish-movement are the time of the request to which they correspond for reference. The **press-key** request does not model the typing skills of an expert typist, but it does represent one who is able to touch type individual letters competently without looking at about 40 words per minute, which is usually a sufficient mechanism for modeling average performance in simple keyboard response tasks.

2.7.1 Motor module chunk-types

Like the **visual-location** and **visual** buffer requests the production which makes the request to the **manual** buffer did not specify a chunk-type. The **manual** buffer request is very similar to the request to the **visual** buffer. It has a slot named cmd which contains the action to perform and then additional slots as necessary to specify details for performing that action. The options for declaring a chunk-type in the request are also very similar to those for the **visual** buffer.

One option is to use the chunk-type named motor-command which includes all of the slots for all of the requests which can be made to the **manual** buffer:

```
+manual>
   isa         motor-command
   cmd         press-key
   key         =letter
```

Another is to use a more specific chunk-type named press-key that only has the valid slots for the press-key action (cmd and key):

```
+manual>
   isa         press-key
   cmd         press-key
   key         =letter
```

Again, that repetition is awkward and we will come back to that later in the tutorial.

2.8 Strict Harvesting

Another mechanism of ACT-R is displayed in the trace of this model. It is a process referred to as "strict harvesting". It states that if the chunk in a buffer is tested on the LHS of a production (often referred to as harvesting the chunk) and that buffer is not modified on the RHS of the production, then that buffer is automatically cleared. This mechanism is displayed in the events of the **attend-letter**, **encode-letter**, and **respond** productions which harvest, but do not modify the **visual-location**, **visual**, and **imaginal** buffers respectively:

```
0.100   PROCEDURAL    PRODUCTION-FIRED ATTEND-LETTER
...
0.100   PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
...
```

```
0.235   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
...
0.235   PROCEDURAL     CLEAR-BUFFER VISUAL
...
0.485   PROCEDURAL     PRODUCTION-FIRED RESPOND
...
0.485   PROCEDURAL     CLEAR-BUFFER IMAGINAL
```

By default, this happens for all buffers except the **goal** and **temporal** buffers (the **temporal** buffer and its module will not be covered in the tutorial), but it is controlled by a parameter called :do-not-harvest which can be used to configure which (if any) of the buffers are excluded from strict harvesting.

If one wants to keep a chunk in a buffer after a production fires without modifying the chunk then it is valid to specify an empty modification to do so. For example, if one wanted to keep the chunk in the **visual** buffer after **encode-letter** fired we would only need to add an =visual> action to the RHS:

```
(P encode-letter-and-keep-chunk-in-visual-buffer
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
   =visual>
)
```

Strict harvesting also applies to the buffer failure query. A production which makes a query for buffer failure will also trigger the strict harvesting mechanism for that buffer and clear it as an action unless that buffer is modified in the production. Typically, there will not be a chunk in the buffer when a buffer failure occurs which needs to be cleared, but clearing a buffer also clears the failure notice from the buffer as well which is useful to avoid detecting the same failure repeatedly.

2.9 More ACT-R Parameters

The model code description document for unit 1 introduced the **sgp** command for setting ACT-R parameters. In the demo2 model the parameters are set like this:

```
(sgp :seed (123456 0))
(sgp :v t :show-focus t :trace-detail high)
```

All of these parameters are used to control how the software operates and do not affect the model's performance of the task. These settings are used to make working with this model easier, and are things that you may want to use when working with other models.

The first **sgp** command is used to set the :seed parameter. This parameter controls the starting point for the pseudo-random number generator used by ACT-R. Typically you do not need to use this parameter; however by setting it to a fixed value the model will always produce the same behavior (assuming that all the variation is attributable to randomness generated using the ACT-R mechanisms). In this model, that is why the randomly chosen letter is always "V". If you remove this parameter setting from the model, save it, and then reload, you will see different letters chosen when the experiment is run. For the tutorial models, we will often set the :seed parameter in the demonstration model of a unit so that the model always produces exactly the same trace as presented in the unit text, but you should feel free to remove that to further investigate the models.

The second **sgp** call sets three parameters. The :v (verbose) parameter controls whether the trace of the model is output. If :v is **t** (which is the default value) then the trace is displayed and if :v is set to **nil** the trace is not printed. It is also possible to direct the trace to an external file instead of the interactive session, and you should consult the reference manual for information on how to do that if you would like to do so. Without printing out the trace the model runs significantly faster, and that will be important in later units when we are running the models through the experiments multiple times to collect data. The :show-focus parameter controls whether or not the visual attention ring is displayed in the experiment window when the model is performing the task. It is a useful debugging tool, but for some displays you may not want it because it could obscure other things you want to see. If it is set to the value **t** then the red ring will be displayed. If it is set to **nil** then it will not be shown. It can also be set to the name of a color e.g. green, blue, yellow, etc. to change how it is displayed which can be helpful when there are multiple models simultaneously interacting with the same task to be able to distinguish where each model is attending. The :trace-detail parameter, which was described in the unit 1 code description document, is set to high so that all the actions of the modules show in the trace for this task.

2.10 Unit 2 Assignment

Your assignment is to extend the abilities of the demo2 model to perform a more complex experiment. The new experiment presents three letters. Two of those letters will be the same. The participant's task is to press the key that corresponds to the letter that is different from the other two. The code to perform the experiment is found in the unit2 file in the Lisp and Python directories. By default it will load the model found in the tutorial/unit2/unit2-assignment-model.lisp file. That initial model file only contains two chunk-type definitions and creates a chunk to indicate the initial goal which is placed into the **goal** buffer.

The experiment is run very much like the demonstration experiment described earlier, and we will repeat the detailed instructions for how to load and run an experiment again here. In future units however we will assume that you are familiar with the process and only indicate the functions necessary to run the experiments.

2.10.1 Running the Lisp version

The first thing you will need to do to run the Lisp version of the experiment is load the experiment code. That can be done using the "Load ACT-R code" button in the Environment. The file is called unit2.lisp and is found in the tutorial/lisp directory of the ACT-R software. When you load that file it will also load the model from the unit2-assignment-model.lisp file in the tutorial/unit2 directory, and if you look at the top of the Control Panel after loading the file you will see that it says UNIT2 under "Current Model". To run the experiment you will call the unit2-experiment function, and it has one optional parameter which indicates whether a human participant will be performing the task. As a first run you should perform the task yourself, and to do that you will evaluate the **unit2-experiment** function at the prompt in the ACT-R window and pass it a value of t (the Lisp symbol representing true):

```
? (unit2-experiment t)
```

When you enter that a window titled "Letter difference" will appear with three letters in it. When you press a key while that experiment window is the active window the experiment will record your key press and determine if it is the correct response. If the response is correct unit2-experiment will return t:

```
? (unit2-experiment t)
T
?
```

and if it is incorrect it will return nil:

```
? (unit2-experiment t)
NIL
?
```

To run the model through the task you call the unit2-experiment function without providing a parameter. With the initial model that will result in this which returns nil indicating an incorrect response (because the model did not make a response):

```
? (unit2-experiment)
 0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL NIL
 0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
 0.000   VISION         visicon-update
 0.000   PROCEDURAL     CONFLICT-RESOLUTION
 0.500   PROCEDURAL     CONFLICT-RESOLUTION
 0.500   ------         Stopped because no events left to process
```

```
NIL
?
```

2.10.2 Running the Python version

To run the Python version you should first run an interactive Python session on your machine (you can use the same session used for the demonstration experiment if it is still open). Then you can import the unit2 module which is in the tutorial/python directory of the ACT-R software. If this is the same session you used before there will not be a notice that it has connected to ACT-R because it is already connected:

```
>>> import unit2
>>>
```

If this is a new session then it will print the confirmation that it has connected to ACT-R:

```
>>> import unit2
ACT-R connection has been started.
```

The unit2 module will also have ACT-R load the unit2-assignment-model.lisp file from the tutorial/unit2 directory, and if you look at the top of the Control Panel after you import the unit2 module you will see that it now says UNIT2 under "Current Model". To run the experiment you will call the **experiment** function from the unit2 module, and it has one optional parameter which indicates whether a human participant will be performing the task. As a first run you should perform the task yourself, and to do that you will call the **experiment** function at the Python prompt and pass it the value True:

```
>>> unit2.experiment(True)
```

When you enter that a window titled "Letter difference" will appear with three letters in it. When you press a key while that experiment window is the active window the experiment will record your key press and determine whether it was the correct response or not. If it was correct it will return the value True:

```
>>> unit2.experiment(True)
True
```

If it was not correct it will return False:

```
>>> unit2.experiment(True)
False
```

To run the model through the task you call the experiment function without providing a parameter. With the initial model that will result in this which returns False indicating an incorrect response (which was because the model did not make a response):

```
>>> unit2.experiment()
```

```
 0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL NIL
 0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
 0.000   VISION         visicon-update
 0.000   PROCEDURAL     CONFLICT-RESOLUTION
 0.500   PROCEDURAL     CONFLICT-RESOLUTION
 0.500   ------         Stopped because no events left to process
False
```

2.10.3 Modeling task

Your task is to write a model that always responds correctly when performing the task. In doing this you should use the demo2 model as a guide. It reflects the way to interact with the imaginal, vision, and motor modules and the productions it contains are similar to the productions you will need to write. You will also need to write additional productions to read the other letters and decide which key to press.

You are provided with a chunk-type you may use for specifying the goal chunk, and the starting model already creates one and places it into the **goal** buffer. This chunk-type is the same as the one used in the demo2 model and only contains a slot named state:

```
(chunk-type read-letters state)
```

The initial goal provided looks just like the one used in demo2**:**

```
(goal isa read-letters state start)
```

There is an additional chunk-type specified which has slots for holding the three letters which can be used for the chunk in the **imaginal** buffer:

```
(chunk-type array letter1 letter2 letter3)
```

You do not have to use these chunk-types to solve the problem. If you have a different representation you would like to use feel free to do so. There is no one “right” model for the task. (Nonetheless, we would like your solution to keep any control state information it uses in the **goal** buffer and separate from the problem representation in the **imaginal** buffer.)

In later units we will be comparing the model’s performance to data from real experiments. Then, how well the model fits the data can be used as a way to decide between different representations and models, but that is not the only way to decide. Cognitive plausibility is another important factor when modeling human performance – you want the model to do the task like a person does the task. A model that fits the data perfectly using a method completely unlike a person is usually not a very useful model of the task.

References

Anderson, J. R., Matessa, M., & Lebiere, C. (1997). ACT-R: A theory of higher level cognition and its relation to visual attention. *Human Computer Interaction*, *12(4)*, 439-462.

Byrne, M. D., (2001). ACT-R/PM and menu selection: Applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies*, *55*, 41-84.

Kieras, D. & Meyer, D.E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, *12*, 391-438.

Unit 2 Code Description

This document will describe the code that implements the experiments from unit 2, how ACT-R is interfaced to them, and some commands that can be used from the prompt to get information about the model instead of using the Environment tools. Before getting into the specifics of the code and commands however we will first describe a component of the ACT-R software which is used to create the experiments and provide a little more information on using the Python interface. Finally, there is also a section at the end of the document with some details about the underlying interface which enables the connection between the core ACT-R software and arbitrary tasks or tools (like the interactive Python interface and the ACT-R Environment).

ACT-R GUI Interface

All of the experiments which are built for the models in the tutorial will be created using a set of interface tools provided with the ACT-R software which we call the AGI (ACT-R GUI Interface). The AGI allows for the creation of simple tasks which can be composed of text, buttons, and lines and interacted with using the keyboard and mouse. When the ACT-R Environment is running, the AGI tasks can be displayed in real windows which can be interacted with by either a person or an ACT-R model (as we saw in the experiments of this unit). Whether the ACT-R Environment is running or not, the AGI can also create a virtual interface which does not display a real window but which can still be interacted with by an ACT-R model exactly the same way as it does with the real window – there is no difference between the real and virtual interface from a model's perspective. The advantage of using a virtual interface for the model is that it is much faster to run the task with a virtual interface than it is a real one, but the downside is that you cannot see the task to monitor what the model is doing which can be important while developing the model and working out any problems in its operation. That is why the AGI provides the model the exact same interface regardless of whether it is a real or virtual window – you can create the task using a real window while developing the model and then change it to a virtual window once the model is working correctly to be able to run it through the task faster for collecting data over multiple trials.

It is not required that one create tasks for an ACT-R model using the AGI. It is also possible to provide features directly to the vision module, and have the model use the virtual keyboard and mouse without an AGI window as well as create new motor interface devices. However, that level of interaction will not be shown in the tutorial.

One final note on the AGI is that it was designed for creating tasks for ACT-R models. The tasks it creates can be interacted with by real participants, but it was not designed with that use in mind. In particular, when running with a real participant it does not make any claims as to the accuracy of the timing information it can collect or the latency of the visual presentations and input responses. For interaction with the model those are not an issue since the model runs in a simulated time frame where the exact time is always available instantly and the clock can pause arbitrarily long before advancing to allow for instantaneous presentations and no-latency on input responses. Therefore, we do not recommend using the AGI to create experiments for real participants if any timing information is to be collected.

Additional Python interface information

Before describing the details of both the Lisp and Python implementations of the experiments, we will provide a little more information on the Python interface to ACT-R. We will also show how you can use some of the ACT-R commands, which the unit 1 code document described being used at the ACT-R prompt, from the Python prompt as well.

The actual interface between Python and the ACT-R software is provided by another Python module called actr which is also located in the python directory of the software. That module provides the code that handles the remote interface described in the appendix of this text, and also defines Python functions which correspond to many of the ACT-R commands available through the remote interface to make them easier to use. That module gets imported by all of the modules for the experiments to enable the interface, and it could also be imported directly if you want easier access to the functions for interacting with ACT-R from the prompt. Once you have done that you can then use the available functions from that module. In general, the Python functions will have the same name as the corresponding command in ACT-R, but with all of the "-" characters replaced with "_" characters to make them valid Python function names. We will describe many of the available functions as we progress through the tutorial, and we will start here with the ones that correspond to the commands used at the ACT-R prompt that where shown in the unit 1 code document: **reset**, **reload**, **run**, **load-act-r-model**, **buffer-chunk**, **dm**, **sdm**, and **whynot**.

The first four of those work the same as the commands described for the ACT-R prompt, and here is an example showing the addition model from unit 1 being loaded, run for 1 second, reloaded, run for .1 seconds, and then being reset.

```
>>> import actr
ACT-R connection has been started with default parameters.
>>> actr.load_act_r_model("ACT-R:tutorial;unit1;addition.lisp")
True
>>> actr.run(1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL SECOND-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK F
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK A
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL A
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK G
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL G
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.350   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            start-retrieval
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK B
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL B
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   DECLARATIVE            start-retrieval
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   DECLARATIVE            RETRIEVED-CHUNK H
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL H
     0.500   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
7
     0.500   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   ------                 Stopped because no events left to process
[0.5, 74, None]
>>> actr.reload()
True
>>> actr.run(.1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL SECOND-GOAL
NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK F
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   ------                 Stopped because time limit reached
[0.1, 20, None]
>>> actr.reset()
True
```

The other ACT-R commands from unit 1, **dm**, **sdm**, **buffer-chunk**, and **whynot**, require a slightly different syntax when called from Python compared to the version called from the ACT-R prompt. In the ACT-R version we could just specify the arguments for the commands without any additional syntactic markers, for example, here is the **buffer-chunk** command being used to get the chunk from the goal buffer at this time:

```
? (buffer-chunk goal)
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1

(SECOND-GOAL-0)
```

From Python however we must specify the arguments as strings. Here we will show those being used, continuing from where we left off in the example above. First, we will run the model for .3 seconds so that there are some chunks in the buffers to view:

```
>>> actr.run(.3)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL SECOND-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK F
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            start-retrieval
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK A
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL A
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            start-retrieval
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK G
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL G
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   ------                 Stopped because time limit reached
[0.3, 46, None]
```

Here is the **buffer_chunk** function being called to print the chunk from the goal buffer:

```
>>> actr.buffer_chunk('goal')
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1
['SECOND-GOAL-0']
```

The return value is a list of the names of the chunks in the requested buffers (also represented as strings). That can be called with any number of buffers (including 0). Here is what happens if we request the retrieval and visual buffers now:

```
>>> actr.buffer_chunk('retrieval','visual')
RETRIEVAL: G-0 [G]
G-0
```

```
   FIRST  6
   SECOND  7
VISUAL: NIL
['G-0', None]
>>>
```

The return value for the visual buffer is None because the buffer is empty. If we call it with no buffers it prints out the contents of all buffers and returns a list of lists where each sublist contains the name of a buffer as the first element, and if the buffer contains a chunk the second element is the name of that chunk:

```
>>> actr.buffer_chunk()
RETRIEVAL: G-0 [G]
IMAGINAL: NIL
MANUAL: NIL
GOAL: SECOND-GOAL-0
IMAGINAL-ACTION: NIL
VOCAL: NIL
AURAL: NIL
PRODUCTION: NIL
VISUAL-LOCATION: NIL
AURAL-LOCATION: NIL
TEMPORAL: NIL
VISUAL: NIL
[['RETRIEVAL', 'G-0'], ['IMAGINAL'], ['MANUAL'], ['GOAL', 'SECOND-GOAL-
0'], ['IMAGINAL-ACTION'], ['VOCAL'], ['AURAL'], ['PRODUCTION'], ['VISUAL-
LOCATION'], ['AURAL-LOCATION'], ['TEMPORAL'], ['VISUAL']]
>>>
```

The **dm** function can be used to print all of the chunks in declarative memory and return a list of their names, or to print only those chunks specified (again using strings to provide the names):

```
>>> actr.dm()
SECOND-GOAL
   ARG1  5
   ARG2  2
J
   FIRST  9
   SECOND  10
I
   FIRST  8
   SECOND  9
H
   FIRST  7
   SECOND  8
G
   FIRST  6
   SECOND  7
F
   FIRST  5
   SECOND  6
E
   FIRST  4
   SECOND  5
D
```

```
   FIRST  3
   SECOND  4
C
   FIRST  2
   SECOND  3
B
   FIRST  1
   SECOND  2
A
   FIRST  0
   SECOND  1
['SECOND-GOAL', 'J', 'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']

>>> actr.dm('a','b','c')
A
   FIRST  0
   SECOND  1
B
   FIRST  1
   SECOND  2
C
   FIRST  2
   SECOND  3
['A', 'B', 'C']
```

For the **sdm** function to search declarative memory we again need to specify the constraints using strings. Here is a search for all of the items which do not have the value of 1 in their first slot:

```
>>> actr.sdm('-','first','1')
SECOND-GOAL
   ARG1  5
   ARG2  2
A
   FIRST  0
   SECOND  1
C
   FIRST  2
   SECOND  3
D
   FIRST  3
   SECOND  4
E
   FIRST  4
   SECOND  5
F
   FIRST  5
   SECOND  6
G
   FIRST  6
   SECOND  7
H
   FIRST  7
   SECOND  8
I
   FIRST  8
   SECOND  9
J
   FIRST  9
   SECOND  10
['SECOND-GOAL', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

Because the value for the slot is an integer it could actually be specified without encoding it into a string and get the same result:

```
>>> actr.sdm('-','first',1)
SECOND-GOAL
   ARG1  5
   ARG2  2
A
   FIRST  0
   SECOND  1
C
   FIRST  2
   SECOND  3
D
   FIRST  3
   SECOND  4
E
   FIRST  4
   SECOND  5
F
   FIRST  5
   SECOND  6
G
   FIRST  6
   SECOND  7
H
   FIRST  7
   SECOND  8
I
   FIRST  8
   SECOND  9
J
   FIRST  9
   SECOND  10
['SECOND-GOAL', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

The **whynot** function also requires that you specify the production names to test using strings and it returns a list of strings which name the productions which do match the current state (regardless of whether they were in the list being tested):

```
>>> actr.whynot('initialize-addition','increment-count')

Production INITIALIZE-ADDITION does NOT match.
(P INITIALIZE-ADDITION
   =GOAL>
       ARG1 =NUM1
       ARG2 =NUM2
       SUM NIL
```

```
 ==>
   =GOAL>
       SUM =NUM1
       COUNT 0
   +RETRIEVAL>
       FIRST =NUM1
)
It fails because:
The chunk in the GOAL buffer has the slot SUM.

Production INCREMENT-COUNT does NOT match.
(P INCREMENT-COUNT
   =GOAL>
       SUM =SUM
       COUNT =COUNT
   =RETRIEVAL>
       FIRST =COUNT
       SECOND =NEWCOUNT
 ==>
   =GOAL>
       COUNT =NEWCOUNT
   +RETRIEVAL>
       FIRST =SUM
)
It fails because:
The value in the FIRST slot of the chunk in the RETRIEVAL buffer does
not satisfy the constraints.
['INCREMENT-SUM']
>>>
```

One thing which you may have noticed is that when you call those functions from Python the output from ACT-R is shown in both the ACT-R window and in your Python session. The same is true if you call the corresponding function from the ACT-R prompt – both interfaces will display the output regardless of how it was generated. If you find that distracting or confusing when working from the Python prompt you can disable the output in the ACT-R window by calling the turn-off-act-r-output command at the ACT-R prompt:

```
? (turn-off-act-r-output)
```

Now that we have shown how ACT-R commands can be used from Python we will describe the Lisp and Python programs which implement the experiments from this unit.

Experiment Code

Below we will show the code which implements the experiments from the unit (in both Lisp and Python) and describe how it works. Before that however we will describe some general details about the structure of the experiment code which has been written for all of the tutorial tasks.

When writing these experiments we have tried to keep the implementations of the tasks fairly straight forward to make it easy to follow how they work. We have also tried to keep the two implementations as similar as possible for comparison purposes. That might not always lead to the most efficient or best looking code, but should help to facilitate the objective of this tutorial – to demonstrate how to use the ACT-R software for creating models and experiments for those models. For many of the experiments in the tutorial there will typically be one function that runs the experiment for either a model or a person, and that function will take an optional parameter which if specified as true (**t** in Lisp and **True** in Python) will run a person instead of the model. Most of the code to perform the task will be the same regardless of whether it is a person or model doing the task, but the code necessary to actually "run" the model and person are different. The code could have been written using separate functions for a model and a human participant, but by using one function it is easier to see where the differences are.

Now we will look at the code for the experiments and provide some description of what it is doing, and also highlight the new ACT-R and AGI functions used in this unit. Those new functions will be described in greater detail at the end of this text. The actual code will be displayed in the same font that has been used for the examples and the descriptions will be in the same font as this paragraph.

Demo2 Lisp

The first thing that the code does is load the corresponding model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit2;demo2-model.lisp")
```

It creates a global variable which will be set to the string naming the key which is pressed.

```
(defvar *response* nil)
```

A function is defined which will be called with two parameters. The first of those parameters is not used, but the second one will be the name of a key which is pressed. That key is stored in the *response* variable and then it clears the experiment window using the AGI function clear-exp-window.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (setf *response* key)
  (clear-exp-window))
```

Here is the demo2-experiment function which runs the task. It takes one optional parameter which can be specified as t (or actually any non-nil value) to indicate a person is doing the task.

```
(defun demo2-experiment (&optional human)
```

It starts by resetting the system using the reset function described in unit 1 to make sure that the model is restored to its starting state.

```
(reset)
```

Then it creates three local variables. Items is a randomly permuted list of the possible letters to display. Text1 is set to the first element from the items list, and window is set to the result of creating a new window for the task titled “Letter recognition” using the open-exp-window function.

```
(let* ((items (permute-list '("B" "C" "D" "F" "G" "H" "J" "K" "L" "M" "N"
                              "P" "Q" "R" "S" "T" "V" "W" "X" "Y" "Z")))
       (text1 (first items))
       (window (open-exp-window "Letter recognition")))
```

The letter from text1 is added to that window using the add-text-to-exp-window function.

```
(add-text-to-exp-window window text1 :x 125 :y 150)
```

The next two function calls are how we get the respond-to-key-press function defined above to be called when a key is pressed in the experiment. First, it uses the add-act-r-command function to create a new command in ACT-R named demo2-key-press which is associated with our respond-to-key-press function and it provides a documentation string to go along with that new command. That new command “demo2-key-press” is then set to monitor the output-key command using the monitor-act-r-command function. A command that monitors another command in this way will be called after that other command and it will be provided the same parameters that the monitored command was. In this case we are monitoring the output-key command because that is how the keyboard associated with experiment windows of the AGI indicate that a key was pressed.

```
(add-act-r-command "demo2-key-press" 'respond-to-key-press
                   "Demo2 task output-key monitor")
(monitor-act-r-command "output-key" "demo2-key-press")
```

The global variable *response* is set to nil to indicate that no key has been pressed.

```
(setf *response* nil)
```

Here we check the value of human to determine how to run the experiment.

```
(if human
```

If it is a human doing the task then we first need to check whether there is a visible window available for the human to see and use to respond:

```
(when (visible-virtuals-available?)
```

If there is then we loop using the while function until the *response* variable is set to a non-nil value calling the process-events command of the AGI repeatedly to allow the system to respond to external input.

```
(while (null *response*)
   (process-events)))
```

If the model is doing the task (not a human) then we need to first tell the model what interface it is interacting with, which in this case is the experiment window which we have created, using the install-device function. Then we run the model for up to 10 seconds. You will also notice that in addition to specifying the time we have also passed the value t to the run command. The second parameter to run is optional, but if a true value is given then the model is run in "real time" which means that its processing of events is synchronized with the actual passing of time instead of running with a simulated clock that goes as fast as possible. When running models with a visible window it is often helpful to run them in real time so that you can actually see what they are doing.

```
(progn
  (install-device window)
  (run 10 t)))
```

Now that the experiment is done we stop monitoring the output-key command with our demo2-key-press command using the remove-act-r-command-monitor function, and then remove our demo2-key-press command from the system using the remove-act-r-command function. These two steps are done as cleanup for safety reasons because we do not want our function to be called when keys are output in any other experiment windows in the future (for example if you load the assignment experiment and try to run it), and to be extra safe we remove the command that is tied to our respond-to-key-press function entirely.

```
(remove-act-r-command-monitor "output-key" "demo2-key-press")
(remove-act-r-command "demo2-key-press")
```

The global variable *response* is then returned from the demo2-experiment function.

```
*response*))
```

Demo2 Python

The first thing that this code does is import the actr module to provide the interface to ACT-R.

```
import actr
```

Then it loads the corresponding model which is able to perform this task.

```
actr.load_act_r_model("ACT-R:tutorial;unit2;demo2-model.lisp")
```

It creates a global variable which will be set to the string naming the key which is pressed.

```
response = False
```

A function is defined which will be called with two parameters. The first of those parameters is not used, but the second one will be the name of a key which is pressed. That key is stored in the global response variable and then it clears the experiment window using the AGI function clear_exp_window.

```
def respond_to_key_press (model,key):
    global response

    response = key
    actr.clear_exp_window()
```

Here is the experiment function which runs the task. It takes one optional parameter which can be specified as True to indicate a person is doing the task.

```
def experiment (human=False):
```

It starts by resetting the system using the reset function described in unit 1 to make sure that the model is restored to its starting state.

```
    actr.reset()
```

Then it creates three local variables. Items is a randomly permuted list of the possible letters to display. Text1 is set to the first element from the items list, and window is set to the result of creating a new window for the task titled “Letter recognition” using the open_exp_window function.

```
    items = actr.permute_list(["B","C","D","F","G","H","J","K","L",
                               "M","N","P","Q","R","S","T","V","W",
                               "X","Y","Z"])
    text1 = items[0]
```

```
window = actr.open_exp_window("Letter recognition")
```

The letter from text1 is added to that window using the add_text_to_exp_window function.

```
actr.add_text_to_exp_window(window, text1, x=125, y=150)
```

The next two function calls are how we get the respond_to_key_press function defined above to be called when a key is pressed in the experiment. First, it uses the add_command function to create a new command in ACT-R named demo2-key-press which is associated with our respond_to_key_press function and it provides a documentation string to go along with that new command. That new command demo2-key-press is then set to monitor the output-key command using the monitor_command function. A command that monitors another command in this way will be called after that other command and it will be provided the same parameters that the monitored command was. In this case we are monitoring the output-key command because that is how the keyboard associated with experiment windows of the AGI indicate that a key was pressed.

```
actr.add_command("demo2-key-press",respond_to_key_press,
                 "Demo2 task output-key monitor")
actr.monitor_command("output-key","demo2-key-press")
```

The global variable response is set to False to indicate that no key has been pressed.

```
global response
response = False
```

Here we check the value of human to determine how to run the experiment.

```
if human == True:
```

If it is a human doing the task then we first need to check whether there is a visible window available for the human to see and use to respond:

```
if actr.visible_virtuals_available():
```

If there is then we loop until the response variable is not False calling the process_events function of ACT-R repeatedly to allow the system to respond to external input.

```
while response == False:
    actr.process_events()
```

If it is not a human doing the task then perform the steps needed to run the model.

```
else:
```

Tell the model what interface it is interacting with, which in this case is the experiment window which we have created, using the install_device function. Then we run the model for up to 10 seconds. You will also notice that in addition to specifying the time we have also passed the value True to the run function. The second parameter to run is optional, but if a True value is given then the model is run in "real time" which means that its processing of events is synchronized with the actual passing of time instead of running with a simulated clock that goes as fast as possible. When running models with a visible window it is often helpful to run them in real time so that you can actually see what they are doing.

```
actr.install_device(window)
actr.run(10,True)
```

Now that the experiment is done we stop monitoring the output-key command with our demo2-key-press command using the remove_command_monitor function, and then remove our demo2-key-press command from the system using the remove_command function. These two steps are done as cleanup for safety reasons because we do not want our function to be called when keys are output in any other experiment windows in the future (for example if you load the assignment experiment and try to run it), and to be extra safe we remove the command that is tied to our respond_to_key_press function entirely.

```
actr.remove_command_monitor("output-key","demo2-key-press")
actr.remove_command("demo2-key-press")
```

The global variable response is then returned from the experiment function.

```
return response
```

Unit2 code

The code to present the assignment's experiment is very similar to the code for the **demo2** model. The only real differences are that more items are displayed and the response is checked for correctness at the end. That only involves the use of one ACT-R function which was not shown above, and only the sections of the unit2 functions which use that new function will be described here. Also, since both the Lisp and Python versions are so similar we will use the same description to cover both code segments.

After permuting the list of letters the target variable is set to the first letter and the foil variable is set to the second letter. Three variables called text1, text2, and text3 are initially set to the foil letter. A random integer from 0 to 2 is generated using a random function from ACT-R (either act-r-random in Lisp or random from the actr module in

Python). That number is used to determine which of the three text variables should hold the target item, and then the three letters are added to the window.

Lisp

```
  (let* ((items (permute-list '("B" "C" "D" "F" "G" "H" "J" "K" "L" "M" "N" 
                                "P" "Q" "R" "S" "T" "V" "W" "X" "Y" "Z")))
         (target (first items))
         (foil (second items))
         (window (open-exp-window "Letter difference"))
         (text1 foil)
         (text2 foil)
         (text3 foil)
         (index (act-r-random 3)))

    (case index
      (0 (setf text1 target))
      (1 (setf text2 target))
      (2 (setf text3 target)))

    (add-text-to-exp-window window text1 :x 125 :y 75)
    (add-text-to-exp-window window text2 :x 75 :y 175)
    (add-text-to-exp-window window text3 :x 175 :y 175)
```

Python

```
    items = actr.permute_list(["B","C","D","F","G","H","J","K","L",
                               "M","N","P","Q","R","S","T","V","W",
                               "X","Y","Z"])
    target = items[0]
    foil = items[1]
    window = actr.open_exp_window("Letter difference")
    text1 = foil
    text2 = foil
    text3 = foil
    index = actr.random(3)

    if index == 0:
       text1 = target
    elif index == 1:
       text2 = target
    else:
       text3 = target

    actr.add_text_to_exp_window(window, text1, x=125, y=75)
    actr.add_text_to_exp_window(window, text2, x=75, y=175)
    actr.add_text_to_exp_window(window, text3, x=175, y=175)
```

The reason that we use ACT-R's random function instead of any native random function that may be available is because we can set the seed of the ACT-R random function using the :seed parameter in the sgp call of the model. That allows us to reproduce the same 'random' sequence with a model and task regardless of how it is being run which can be extremely useful when trying to debug a random problem with a model or when creating consistent example runs.

Safety note

Before describing the new ACT-R commands in detail there is one note about the code for these experiments which should be pointed out. The functions that are handling the key press are being called during the execution of other functions (process-events when a person does the task and run when the model does the task). That happens because the monitoring functions get called in a separate thread from the one which is running the main task with the way the ACT-R interfaces have been created. Since both threads are accessing the same global variable (response) the safe thing to do would be to include the appropriate protection on that to avoid problems (a lock, semaphore, or some other construct available in the language used). However, since all we are looking for here is a simple change to the value and only one of the threads sets the variable there should not be any problems with the operation of this code and that protection has been ignored for the purpose of keeping the example task easy to read. If you are creating more complicated tasks which are using monitors for things like key presses and mouse clicks or other multi-threaded actions then you may need to put in the necessary protection for access to shared resources (like global variables) to avoid problems.

New ACT-R commands

Below we will provide more details on the new functions used in creating these tasks, and they will be grouped based on their general purpose. Many of these commands will be used in most of the tasks throughout the tutorial.

AGI commands

creating a window

The **open-exp-window** and **open_exp_window** functions are used to open a window to display a task which can be interacted with by either an ACT-R model or a real person. The function requires one parameter which is a string containing the title for that window, and that title should be unique i.e. only one window with a given title may be open at a time. If the name specified is the name of a window which was created previously then that existing window will be closed first, and then a new window created. The return value of that function is a window description which can be passed to other AGI functions for indicating which window to operate on and it is also a valid device list that can be installed for the model to interact with. There are also several other parameters which may be provided when creating a window and those will be described in later units when used by the tasks involved.

displaying text in a window

The **add-text-to-exp-window** and **add_text_to_exp_window** functions display text in a window that was opened using **open-exp-window** or **open_exp_window**. It has two required parameters. The first is a window description to indicate which window, and the second is a string of the text to display. It has multiple additional parameters which are accessed using keyword parameters in Lisp and keyword arguments in Python. Two of them are used in this unit's tasks: x and y. Those are the x and y coordinate within the window at which the upper-left corner of the text to be displayed will be positioned and should be integers (the upper-left corner of the window is 0,0 with x increasing to the right and y increasing toward the bottom). It returns a descriptor for the text item which can be used to remove or change that item, but the details of that descriptor are not part of the specification and it should not be used for any other purpose.

clearing a window

The **clear-exp-window** and **clear_exp_window** functions are used to clear all items from a window which was opened using **open-exp-window** or **open_exp_window**. It has one optional parameter which if provided should be a window description. If only one window has been opened then the optional parameter is not needed and that open window will be the one cleared. It removes all of the items that have been added to that window.

key presses

When a key is pressed in a window which was created with **open-exp-window** or **open_exp_window** by a person or on the corresponding virtual keyboard device which is installed for models interacting with those windows, the ACT-R command **output-key** is called. That command is called with two parameters. The first is the name of a model which made the key press if it was made by a model or a value of **nil** (Lisp) or **None** (Python) if it was by a person interacting with the window. The second will be a string indicating the key which was pressed. When the model makes the action the event is also shown in the trace as seen in the last line of this segment of the trace from running the demo2 task:

```
     0.485   PROCEDURAL        MODULE-REQUEST MANUAL
     0.485   PROCEDURAL        CLEAR-BUFFER IMAGINAL
     0.485   PROCEDURAL        CLEAR-BUFFER MANUAL
     0.485   MOTOR             PRESS-KEY KEY v
     0.485   PROCEDURAL        CONFLICT-RESOLUTION
     0.735   MOTOR             PREPARATION-COMPLETE
     0.735   PROCEDURAL        CONFLICT-RESOLUTION
     0.785   MOTOR             INITIATION-COMPLETE
     0.785   PROCEDURAL        CONFLICT-RESOLUTION
     0.885   KEYBOARD          output-key DEMO2 v
```

To detect and record key presses from the experiment windows one will need to monitor the output-key command as described below.

checking for a visible window

The **visible-virtual-available?** and **visible_virtual_available** functions are used to test whether the AGI is able to open a visible window. If so then the functions return a true value otherwise they return a false value. If the ACT-R Environment application is running then visible windows can be opened. There is another visible virtual window tool available which works in a browser and it is also possible to create your own window handler for the AGI, but those are not described in the tutorial.

Model interaction with tasks

To have a model interact with a task it must be told which 'devices' to use by using the **install-device** or **install_device** function. That function requires one parameter which must be a specification of a device for the model. A device is a general term for the types of things that a model can interact with and a window created by the **open-exp-window** or **open_exp_window** functions is a valid device. The device (or possibly multiple devices) which are installed for a model indicate where its percepts come from and/or where its output actions will go. The windows of the AGI provide visual percepts and they also install virtual keyboard and mouse devices with which the model can interact as well as a virtual microphone for recording the model's speech. Creating new devices for a model is one way to interface a model to new environments, but that is beyond the scope of the tutorial.

Running a model

The **run** function was described in unit 1, but here we see that it takes an optional second parameter. The **run** function causes the simulated clock for the ACT-R system to advance which allows the model(s) to perform actions. The first parameter to **run** indicates the maximum amount of time that the system will be allowed to run specified in seconds. The second parameter is optional, but if provided as true (**t** in Lisp or **True** in Python) then the simulated clock for ACT-R will advance in step with the real passage of time instead of as fast as possible. The optional parameter can also be specified as a number to indicate a desired scaling of model time to real time (a value of 1 is the same as specifying true), but there are no guarantees on the ability to achieve a desired scaling – you can request a million to one scaling but it is unlikely to actually be able to achieve that.

Miscellaneous ACT-R functions

The **act-r-random** or **random** function can be used to return a pseudo-random number based on an initial seed which can be specified using the **sgp** command in a model. It requires one parameter which must be a number. If the number provided is an integer then the return value will be an integer chosen uniformly from 0 to that number minus 1. If it is a non-integer real number, N, then a real number uniformly chosen from the range of [0,N) will be returned. Using the random function provided by ACT-R when creating tasks allows one to generate the same "random" sequence of events for a model and task regardless of how the task is being run by specifying the same initial seed in the model definition. The specific algorithm used for the ACT-R random numbers is currently the MT19937 generator.

The **permute-list** or **permute_list** function can be used to permute the items in a list using the ACT-R random number generator to do so. They require one parameter which must be a list of items and it returns a randomly ordered copy of that list.

When a waiting loop is needed to collect a real response from an AGI window the **process-events** or **process_events** function should be called in that loop to allow the system a chance to handle the user interactions. It takes no parameters. It ensures that other threads are allowed to run which may be necessary for the system to be able to handle things like calling a monitoring function in response to a user pressing a key.

For the Lisp interface to ACT-R we have provided a **while** macro as a looping construct. It takes an arbitrary number of parameters. The first parameter specifies the test condition, and the rest specify the body of the loop. The test is evaluated and if it returns anything other than **nil** all of the forms in the body are executed in order. This is repeated until the test returns **nil**. Thus, while the test is true (non-nil) the body is executed. This is not really necessary because there are several other looping constructs available in Lisp, but a simple while can be easier to understand for novice Lisp programmers and it makes it easier to create a nearly line-to-line correspondence between the Lisp and Python versions of the tasks.

Like the buffer-chunk function which was shown in the previous unit to access the contents of a buffer, there are also **buffer-status** and **buffer_status** functions which will provide the status of the queryable information from the buffer (which is also shown in the “Buffers” tool of the Environment using the “Status” button described in this unit). It can be called with the names of any number of buffers and will print out the current status information for the queries of those buffers. Here are examples of calling it at the ACT-R prompt and in Python.

```
? (buffer-status goal retrieval)
GOAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
RETRIEVAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  recently-retrieved nil: NIL
  recently-retrieved t  : NIL

(GOAL RETRIEVAL)

>>> actr.buffer_status('goal','retrieval')
GOAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
RETRIEVAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  recently-retrieved nil: NIL
  recently-retrieved t  : NIL
['GOAL', 'RETRIEVAL']
```

ACT-R Software Interface

Before describing the specific functions that were used to have ACT-R call a function we had written for collecting a key press response, we will first describe the general mechanism which makes this possible at a fairly high level. The underlying details, which would be necessary if one wanted to extend things to create an interface to a different language, are well beyond the scope of the tutorial, but are available in a manual called remote in the docs directory of the distribution.

The key feature of the current ACT-R software which allows for the communication between ACT-R and arbitrary ‘other’ code is that it has been built around a central RPC (remote procedure call) system. The central RPC system (which we will refer to as the dispatcher) is responsible for accepting connections from clients (which include the core ACT-R code), maintaining the set of commands which those clients have made available, and coordinating the communication between a client that wants to execute a command and the client which has provided that command. The clients can be connected to the dispatcher directly through Lisp (as is the case for the core ACT-R code and any code loaded directly into the Lisp running ACT-R) or through a TCP/IP socket connection to the dispatcher (which is how the Python and ACT-R Environment connections are made), and the dispatcher allows for an unlimited number of clients to be connected at any time (theoretically at least since there are of course computational constraints). Any of the connected clients can add a command to the set available from the dispatcher, and that command can then be used by any of the connected clients with the dispatcher responsible for handling the communication between them.

In addition to supporting the communication between clients, the dispatcher also provides a 'monitoring' mechanism through which any command which has been added can get called automatically when another command is used. This monitoring mechanism allows a client to provide commands which other clients can then detect and respond to without that original client needing to know about every other client that wants to be notified. For example, this is why the output of the model trace is shown in both an interactive Python session and the ACT-R window – both of those clients are monitoring the commands responsible for printing the trace and then displaying the results.

Both the Lisp and Python interfaces to ACT-R provide the modeler with access to that central dispatcher. That allows the modeler to add new commands which can be called by code in ACT-R (which we will see in unit 5 when providing a function to create similarity values) and which can monitor other commands, as is done in this unit to monitor the output-key command provided by the default keyboard device. Here we will describe the functions that provide access to the dispatcher.

To add a new command one uses the **add-act-r-command** or **add_command** function. It requires one parameter and has four optional parameters (only two of which will be described here). The first parameter must be a string which is the name of the new command to add. That name is case sensitive and it must not match the name of a command which already exists. The second parameter is optional, and specifies the local function which should be called when that command is evaluated (if no command is indicated then there is no activity associated with that command but it can still be called and monitored). The third parameter is also optional, but if given should be a string which provides some documentation about the command being added. If the command is added successfully then the function returns a true result (t or True) and if not, a warning is displayed and a null result (nil or False) is returned.

Once a command has been added it can be removed using the **remove-act-r-command** or **remove_command** function. That requires a single parameter which is the string that names a command. If it is successfully removed a true result is returned and if not a null result is returned.

To monitor a command the **monitor-act-r-command** or **monitor_command** function is used. It requires two parameters. The first parameter should be a string that names a command available from the dispatcher. That is the command which is being monitored. The second parameter should be a string which names another command available from the dispatcher. That is the command which is monitoring the other. The monitoring command will be called after every call to the monitored command and it will be passed the same parameters which the monitored command was given. If the monitoring is set up successfully then a true result will be returned and if not a null result will be returned.

To stop monitoring a command the **remove-act-r-command-monitor** or **remove_command_monitor** function is used. It requires two parameters which are the same as were specified when monitoring was initiated. The first is a string which names the command to monitor and the second is a string naming the command which is currently monitoring it but should stop monitoring now. If the monitoring is successfully removed then a true result is returned and if not a null result is returned.

Unit 3: Attention

This unit is concerned with developing a better understanding of how perceptual attention works in ACT-R, particularly as it is concerned with visual attention.

3.1 Visual Locations

When a visual display such as this is presented to ACT-R

V	N	T	Z
C	R	Y	K
W	J	G	F

a representation of all the visual information is made accessible to the model in a visual icon maintained by the vision module. One can view the contents of this visual icon using the “Visicon” button in the ACT-R Environment or with the **print-visicon** command (using the print-visicon function at the ACT-R prompt or the print_visicon function in the actr module of the Python interface):

```
? (print-visicon)

>>> actr.print_visicon()
```

```
Name                Att  Loc             Text  Kind  Color  Width  Value  Height  Size
-----------------   ---  --------------  ----  ----  -----  -----  -----  ------  ----------
VISUAL-LOCATION0    NEW  (380 406 1080)  T     TEXT  BLACK  7      "v"    10      0.19999999
VISUAL-LOCATION1    NEW  (380 456 1080)  T     TEXT  BLACK  7      "c"    10      0.19999999
VISUAL-LOCATION2    NEW  (380 506 1080)  T     TEXT  BLACK  7      "w"    10      0.19999999
VISUAL-LOCATION3    NEW  (430 406 1080)  T     TEXT  BLACK  7      "n"    10      0.19999999
VISUAL-LOCATION4    NEW  (430 456 1080)  T     TEXT  BLACK  7      "r"    10      0.19999999
VISUAL-LOCATION5    NEW  (430 506 1080)  T     TEXT  BLACK  7      "j"    10      0.19999999
VISUAL-LOCATION6    NEW  (480 406 1080)  T     TEXT  BLACK  7      "t"    10      0.19999999
VISUAL-LOCATION7    NEW  (480 456 1080)  T     TEXT  BLACK  7      "y"    10      0.19999999
VISUAL-LOCATION8    NEW  (480 506 1080)  T     TEXT  BLACK  7      "g"    10      0.19999999
VISUAL-LOCATION9    NEW  (530 406 1080)  T     TEXT  BLACK  7      "z"    10      0.19999999
VISUAL-LOCATION10   NEW  (530 456 1080)  T     TEXT  BLACK  7      "k"    10      0.19999999
VISUAL-LOCATION11   NEW  (530 506 1080)  T     TEXT  BLACK  7      "f"    10      0.19999999
```

That command prints the information of all the features that are available for the model to see, and those features are represented as chunks in the vision module. For each feature it shows the name which has been created for that chunk, its attentional status, and the location of the item. The other visual features of an item can vary based on the type of the item and are determined by the source of those features. The features shown above: kind, color, width, value, height, and size, are the ones that are created for text items by the AGI experiment window device.

3.1.1 Visual-Location Requests

Those low-level features are what is searched when a **visual-location** request is made. When requesting the visual location of an object any of the features available may be used in the request. In the last unit we only used the request parameter :attended when making a **visual-location** request. We will expand on the use of :attended in this unit. We will also provide details on specifying the slots in a **visual-location** request, and show another request parameter available for the **visual-location** requests called :nearest.

3.1.2 The Attended Test in More Detail

The :attended request parameter was introduced in unit 2. It tests whether or not the model has attended the object at that location, and the possible values are **new**, **nil**, and **t**. Very often we use the fact that attention tags elements in the visual display as attended to enable us to draw attention to the previously unattended elements. Consider the following production:

```
(p find-random-letter
   =goal>
     isa      read-letters
     state    find
==>
   +visual-location>
     :attended nil
   =goal>
     state    attending)
```

In its action, this production requests the location of an object that has not yet been attended. Otherwise, it places no preference on the location to be found. When there is more than one item in the visicon that matches the **visual-location** request, the one most recently added to the visual icon (the newest one) will be chosen. If multiple items also match on their recency, then one will be picked randomly among those. If there are no objects which meet the constraints, then a **failure** will be signaled for the **visual-location** buffer. After a feature is attended (with a **visual** buffer request to move-attention), it will be tagged as attended **t** and this production's request will not return the location of such an object.

3.1.2.1 Finsts

There is a limit to the number of objects which can be tagged as attended **t**, and there is also a time limit on how long an item will remain marked as attended **t**. These attentional markers are called finsts (INSTantiation FINgers) and are based on the work of Zenon Pylyshyn. The default number of finsts provided by the vision module is four, and the default decay time is three seconds. Thus, at any time there can be no more than four visual objects marked as attended **t**, and after three seconds the attended state of an item will revert from **t** to **nil**. Also, when attention is shifted to an item that would require more finsts than there are available the oldest one is reused for the new item i.e. if there are four items marked with finsts and you move attention to a fifth item the first item that had been marked as attended will no longer be marked as attended so that the fifth item can be marked as attended. Because the default value is small, productions like the one above are not very useful for modeling tasks with a large number of items on the screen because the model will end up revisiting items very quickly. The number of finsts and the length of time that they persist can be changed using the parameters :visual-num-finsts and :visual-finst-span respectively in the model. Thus, one solution is to just set :visual-num-finsts to a value which is large enough to work for your task. However, changing architectural parameters like those is not recommended without a good reason, and keeping the number of parameters one needs to change for a model as small as possible is generally desired. After discussing some of the other specifications one can use in a request we will come back to ways to work with the limited set of finsts.

3.1.3 Visual-location slots

This is the chunk-type used to specify the location chunks for the text items created by the experiment window device:

```
(chunk-type visual-location screen-x screen-y distance height width
            size color kind value)
```

Those slots hold the information shown in the visicon above and that chunk-type could be used to declare the desired slots when making visual-location request. The screen-x and screen-y slots represent the location based on its x and y position on the screen and are measured in pixels. The upper left corner of the screen is screen-x 0 and screen-y 0 with x increasing from left to right and y increasing from top to bottom. The location of an item depends both upon where it is located within its window and where that window itself is located. The distance slot will always have a value of 1080, which is also measured in pixels, and represents a distance of 15 inches from the model to the screen (the screen resolution of the experiment windows is assumed to be 72 pixels per inch).

The height and width slots hold the dimensions of the item measured in pixels. The size slot holds the approximate area covered by the item measured in degrees of visual angle squared. These values provide the general shape and size of the item on the display.

The color slot holds a representation of the color of the item. This will be a symbolic value like black or red which names a chunk that has been created by the vision module.

The kind slot specifies a general classification of the item, like text or line, which are also chunks created by the vision module.

The information shown for the value by the visicon is actually the information which will be available to the model once it has shifted attention to the item. That information may be used when making the request for a visual location, but it is typically not available to the model in the chunk which is returned in response to a **visual-location** request i.e. the model can request a visual location which has a value of “v” on the display (it can look for a “v”) which will return a chunk that represents a location where there might be a “v”, and then the model must attend to that location using a **visual** request to move-attention and encode the letter “v”.

3.1.4 Visual-location request specification

One can specify constraints for a **visual-location** request using any of the slots which have been created for visual features. Any of the slots may be specified using any of the modifiers (-, <, >, <=, or >=) in much the same way one specifies a **retrieval** request, and any of the slots may be specified any number of times. In addition, there are some special tests which one can use that will be described below. All of the constraints specified will be used to find a location in the visicon to be placed into the **visual-location** buffer. If there is no location in the visicon which satisfies all of the constraints then the **visual-location** buffer will indicate a failure.

3.1.4.1 Exact values

If you know the exact values for the slots you are interested in then you can specify those values directly and you can also use the negation test, -, with those values:

```
+visual-location>
   isa visual-location
   screen-x 50
   screen-y 124
   color    black
 - kind     text
```

Often however, one does not know the specific information about the location of visual items in advance and things need to be specified more generally in the model.

3.1.4.2 General values

When the slot being tested holds a number it is possible to use the slot modifiers <, <=, >, and >= to test a slot’s value. Thus to request a location that is to the right of screen-x 50 and at or above screen-y 124 one could use the request:

```
+visual-location>
 >  screen-x 50
 <= screen-y 124
```

In fact, one could use two modifiers for each of the slots to restrict a request to a specific range of values. For instance to request an object which was located somewhere within a box bounded by the corners 10,10 and 100,150 one could specify:

```
+visual-location>
  > screen-x 10
  < screen-x 100
  > screen-y 10
  < screen-y 150
```

3.1.4.3 Production variables

It is also possible to use variables from the production in the requests instead of specific values. Consider this production which uses a value from a slot in the **goal** buffer to request the location with a specific color:

```
(p find-by-color
   =goal>
    target =color
==>
  +visual-location>
    color  =color)
```

Variables from the production can be used just like specific values along with modifiers. Assuming that the LHS of the production binds the variables =x, =y, and =kind this would be a valid request:

```
  +visual-location>
    kind     =kind
 <  screen-x =x
 -  screen-x 0
 >= screen-y =y
 <  screen-y 400
```

3.1.4.4 Relative values

If you are not concerned with any specific values, but care more about relative properties then there are also ways to specify that. You can use the values **lowest** and **highest** in the specification of any slot which has a numeric value. Of the chunks which match the other constraints the one with the numerically lowest or highest value for that slot will then be the one found.

In terms of screen-x and screen-y, remember that for the experiment window device the x coordinates increase from left to right, so **lowest** corresponds to leftmost and **highest** rightmost, while y coordinates increase from top to bottom, so **lowest** means topmost and **highest** means bottommost.

If this is used in combination with :attended it can allow the model to find things on the screen in an ordered manner. For instance, to read the screen from left to right a model could use a visual-location request like this:

```
+visual-location>
   :attended nil
   screen-x lowest
```

assuming of course that it also moves attention to the items so that they become attended and that there are sufficient finsts to tag everything along the way.

If multiple slots in the request specify the relative constraints of **lowest** and/or **highest** then first, all of the non-relative values are used to determine the set of items to be tested for relative values. Then the relative tests are performed one at a time in the order provided to reduce the matching set. Thus, this request:

```
+visual-location>
   width    highest
   screen-x lowest
   color    red
```

will first consider all items which are red because that is not a relative test. Then it would reduce that to the set of items with the highest width (widest) and then from those it would pick the one with the lowest screen-x coordinate (leftmost). That may not produce the same result as this request given the same set of visicon chunks since the screen-x and width constraints will be applied in a different order:

```
+visual-location>
   screen-x lowest
   width    highest
   color    red
```

3.1.4.5 The current value

It is also possible to use the special value **current** in a **visual-location** request. That means the value of the slot must be the same as the value for the corresponding slot of the location of the currently attended object (the one attention was last shifted to with a **visual** request to move-attention). The value **current** may also be tested using the modifiers. Therefore, this request:

```
+visual-location>
     screen-x current
   < screen-y current
   - color    current
```

attempts to find a location which has the same x position as the currently attended item, is higher on the screen than the currently attended item (the y coordinate is lower), and is a different color than the currently attended item.

If the model does not have a currently attended object (it has not yet attended to anything) then the tests for **current** are ignored.

3.1.4.6 Request variables

A special component of **visual-location** requests is the ability to use variables to compare the features within a particular location to each other in the same way that the LHS tests of a production use variables to match the contents of slots. If a value for a slot in a **visual-location** request starts with the character & then it is considered to be a variable in the request in the same way that values starting with = are considered to be variables on the LHS of a production. The request variables can be combined with the modifiers and any of the other values allowed to be used in the requests. Here is an example which may not be the most practical, but shows many of the mechanisms available for **visual-location** requests in use together:

```
+visual-location>
   screen-x current
   screen-x &x
 > screen-y 100
   screen-y lowest
 - screen-y &x
```

That request would try to find a location which has a screen-x value that is the same as the currently attended location's screen-x and a screen-y value which is the lowest one greater than 100 but not the same as its screen-x value.

Request variables for the **visual-location** requests are not very useful with the features provided by the experiment window device, but could become very useful if one creates custom visual feature representations.

3.1.5 The :nearest request parameter

Like :attended, there is another request parameter available in **visual-location** requests. The :nearest request parameter can be used to find the items closest to the currently attended location or to some other location. To find the location of the object nearest to the currently attended location we can again use the value **current**:

```
+visual-location>
   :nearest current
```

If a location nearest to some other location is desired that other location can be provided as the value for :nearest instead of **current**:

```
+visual-location>
   :nearest  =some-location
```

If there are constraints other than :nearest specified in the request then they are all tested first and the nearest of the locations that matches all of those other constraints is the one that will be placed into the buffer. The determination of "nearest" is based on the straight line distance using the coordinates of the items, which would be the screen-x, screen-y, and distance slots for the experiment window device's features.

3.1.6 Ordered Search

Above it was noted that a production using this **visual-location** request (in conjunction with appropriate attention shifts) could be used to read words on the screen from left to right:

```
   +visual-location>
```

```
      :attended nil
      screen-x  lowest
```

However, if there are fewer finsts available than words to be read that production will result in a loop that reads only one more word than there are finsts. For instance, if there are six words on the line and the model only has four finsts (the default) then when it attends the fifth word the finst on the first word will be removed to use because it is the oldest. Then the sixth request will result in finding the location of the first word again because it is no longer marked as attended. If it is attended it will get the finst from the second word, and so on.

By using the tests for current and lowest one could have the model perform the search from left to right without using the :attended test:

```
   +visual-location>
    > screen-x current
      screen-x lowest
```

That will always be able to find the next word to the right of the currently attended one regardless of how many finsts are available and in use.

To deal with multiple lines of items to be read left to right one could add ‘screen-y current’ to that request along with an additional production for moving to the next line when the end of the current one is reached. By using the relative constraints along with the :nearest request parameter and the **current** indicator a variety of ordered search strategies can be implemented in a model.

3.2 The Sperling Task

The example model for this unit can perform the Sperling experiment which demonstrates the effects of perceptual attention. The model is found in the sperling-model.lisp file in unit3 of the tutorial and the experiment code is in the sperling file for each of the provided implementations. Loading or importing the experiment code (as appropriate) will automatically load the model into ACT-R. In the Sperling experiment subjects are briefly presented with a set of letters and must try to report them. Subjects see displays of 12 letters such as:

```
V   N   T   Z

C   R   Y   K

W   J   G   F
```

and we are modeling the partial report version of the experiment. In this condition, subjects are cued sometime after the display comes on as to which of the three rows they must report. The delay of the cue is either 0, .15, .3, or 1 second after the display appears. Then, after 1 second of total display time, the screen is cleared and the subject is to report the letters from the cued row. In the version we have implemented the responses are to be typed in and the space bar pressed to indicate completion of the reporting. For the cueing, the original experiment used a tone with a different frequency for each row and the model will hear simulated tones while it is doing the task. This task does not have a version which you can perform because the AGI does not currently provide a means of generating real tones.

In the original experiment the display was only presented for 50 ms and it is generally believed that there is an iconic visual memory that continues to hold the stimuli for some time after features are removed from the actual display which people can continue to process. ACT-R's vision module does not currently provide a persistent visual iconic memory – it can only process the items immediately available from the device. Thus, for this task we have simulated this persistent visual memory for ACT-R by having the display actually stay on for longer than 50ms. It will be visible for a random period of time between 0.9 to 1.1 seconds to simulate that process.

One thing you may notice when looking at this model is that it does not use the imaginal module, as described in the previous unit, to hold the problem representation separate from the control state. Instead, all of the task relevant information is kept in the **goal** buffer's chunk. That was done primarily to keep the productions simpler so that it is easier to follow the details of the attention mechanisms which are the focus of this unit. As an additional task for this unit you could rewrite the productions for this example model to represent the information more appropriately using both the **goal** and **imaginal** buffers.

To run the model through a single trial of the task you can use the sperling-trial function in the Lisp version and the trial function in the sperling module of the Python version. Those functions require a single parameter which indicates the delay in seconds for the signal tone which indicates the row to be reported. Here are examples of running that in both implementations specifying a delay of .15 seconds:

```
? (sperling-trial .15)
```

```
>>> sperling.trial(.15)
```

This is the trace which is generated when that trial is run (the :trace-detail parameter is set to low in the model). In this trial the sound is presented .15 seconds after onset of the display and the target row is the middle one.

```
0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL NIL
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
0.050   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.185   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3
0.200   AUDIO          SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0 NIL
0.235   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.285   PROCEDURAL     PRODUCTION-FIRED DETECTED-SOUND
0.320   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
0.370   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.370   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2
0.420   PROCEDURAL     PRODUCTION-FIRED ATTEND-LOW
0.505   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
```

```
0.555   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.555   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0
0.570   AUDIO          SET-BUFFER-CHUNK AURAL TONE0
0.605   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.655   PROCEDURAL     PRODUCTION-FIRED SOUND-RESPOND-MEDIUM
0.690   VISION         SET-BUFFER-CHUNK VISUAL TEXT3
0.740   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.740   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION4
0.790   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
0.875   VISION         SET-BUFFER-CHUNK VISUAL TEXT4
0.925   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.925   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
0.975   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
1.110   PROCEDURAL     PRODUCTION-FIRED START-REPORT
1.110   GOAL           SET-BUFFER-CHUNK GOAL CHUNK0
1.110   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL TEXT0-0
1.160   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.160   MOTOR          PRESS-KEY KEY c
1.160   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL TEXT4-0
1.760   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.760   MOTOR          PRESS-KEY KEY r
1.760   DECLARATIVE    RETRIEVAL-FAILURE
2.260   PROCEDURAL     PRODUCTION-FIRED STOP-REPORT
2.260   MOTOR          PRESS-KEY KEY SPACE
2.560   ------         Stopped because no events left to process
```

Although the sound is presented at .150 seconds into the trial it does not actually affect the model until the **sound-respond-medium** production fires at .655 seconds to encode the tone. One of the things we will discuss is what determines the delay of that response. Prior to that time the model is finding letters anywhere on the screen, but after the sound is encoded the search is restricted to the target row indicated. After the display disappears, the production **start-report** fires which initiates the keying of the letters which the model remembers having attended from the target row.

3.3 Visual Attention

As in the models from the last unit there are three steps that the model must perform to encode visual objects. It must find the location of an object, shift attention to that location, and then harvest the chunk which encodes the attended object. In the last unit this was done with three separate productions, but in this unit, because the model is trying to do this as quickly as possible the encoding and request to find the next are actually combined into a single production, **encode-row-and-find**, which will be described later. In addition, for the first item's location there is no production that does an initial find.

3.3.1 Buffer Stuffing

Looking at the trace we see that the first production to fire in this model is **attend-medium**:

```
0.050   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
```

Here is the definition of that production:

```
(p attend-medium
   =goal>
     isa        read-letters
     state      attending
   =visual-location>
     isa        visual-location
   > screen-y   450
   < screen-y   470

   ?visual>
     state      free
==>
   =goal>
     location   medium
     state      encode
   +visual>
     cmd        move-attention
     screen-pos =visual-location)
```

On its LHS it has a test for a chunk in the **visual-location** buffer, and it matches and fires even though there has not been a prior production firing to make a request to find a location chunk to place into the **visual-location** buffer. However, there is a line in the trace prior to that which indicates that a chunk was placed into the **visual-location** buffer:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
```

This process is referred to as “buffer stuffing” and it occurs for both visual and aural percepts. It is intended as a simple approximation of a bottom-up mechanism of attention. When the **visual-location** buffer is empty and there is a change in the visual scene it can automatically place the location of one of the visual objects into the **visual-location** buffer. The “nil” at the end of the line in the trace indicates that this setting of the chunk in the buffer was not the result of a production’s request.

You can specify the conditions used to determine which location, if any, gets selected for the **visual-location** buffer stuffing using the same conditions you would use to specify a **visual-location** request in a production. Thus, when the screen is processed, if there is a visual-location that matches that specification and the **visual-location** buffer is empty that location will be stuffed into the **visual-location** buffer.

The default specification for a location to be stuffed into the buffer 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 location there is in fact already one in the buffer, and it is the leftmost new item on the screen.

Using buffer stuffing allows the model to detect changes to the screen automatically. The alternative method would be to continually request a location that was marked as :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 will only occur if the buffer is empty. If the model is busy doing something with a chunk in the **visual-location** buffer then it will not automatically notice a change to the display. If you want to take advantage of buffer stuffing in a model then you must make sure that all requested chunks are cleared from the buffer (the vision module will erase a stuffed chunk automatically from the visual-location buffer after .5 seconds). That is typically not a problem because the strict harvesting mechanism that was described in the last unit causes buffers to be cleared automatically when they are tested in a production.

3.3.2 Testing and Requesting Locations with Slot Modifiers

Something else to notice about this production is that the buffer test of the **visual-location** buffer shows modifiers being used when testing slots for values. These tests allow you to do a comparison when the slot value is a number, and the match is successful if the comparison is true. The first one (>) is a greater-than test. If the chunk in the **visual-location** buffer has a value in the screen-y slot that is greater than 450, it is a successful match. The second test (<) is a less-than test, and works in a similar fashion. If the screen-y slot value is less than 470 it is a successful match. Testing on a range of values like this is important for the visual locations because the exact location of a piece of text in the visual icon is determined by its “center” which is dependant on the font type and size. Thus, instead of figuring out exactly where the text is at in the icon (which can vary from letter to letter or even for the same letter under different fonts) the model is written to accept the text in a range of positions to indicate which row it occupies.

After attention shifts to a letter on the screen, the production **encode-row-and-find** can harvest the visual representation of that object. It modifies that chunk to indicate which row it is in and requests the next location:

```
(p encode-row-and-find
   =goal>
     isa       read-letters
     location  =pos
     upper-y   =uy
     lower-y   =ly
   =visual>
==>
   =visual>
     status    =pos
   -visual>
   =goal>
     location  nil
     state     attending
   +visual-location>
     :attended nil
   > screen-y  =uy
   < screen-y  =ly)
```

The production places the row of the letter, which is in the variable =pos and bound to the value from the location slot of the chunk in the **goal** buffer, into the status slot of the chunk currently in the **visual** buffer. Later, when retrieving the chunks from declarative memory, the model will restrict itself to recalling items from only the target row. The status slot is used because the visual objects created by the experiment window device use this chunk-type to create the chunks for the **visual** buffer:

```
(chunk-type visual-object screen-pos value status color height width distance)
```

but it does not actually place a value into the status slot when it creates the chunk so that slot is available for the modeler to use as needed. We would not have to use that slot, but since it is already defined it is convenient to do so. Later in the tutorial we will show how a model can extend chunks with arbitrary slots as it runs.

In addition to modifying the chunk in the **visual** buffer, it also explicitly clears the **visual** buffer. This is done so that the now modified chunk goes into declarative memory. Remember that declarative memory holds the chunks that have been cleared from the buffers. Typically, strict harvesting will clear the buffers automatically, but because the chunk in the **visual** buffer is modified on the RHS of this production it will not be automatically cleared. Thus, to ensure that this chunk enters declarative memory at this time we explicitly clear the buffer.

The production also updates the **goal** buffer's chunk to remove the location slot and update the state slot, and makes a request for a new visual location. The **visual-location** request uses the < and > modifiers for the screen-y slot to restrict the visual search to a particular region of the screen. The range is defined by the values from the upper-y and lower-y slots of the chunk in the **goal** buffer. The initial values for the upper-y and lower-y slots are shown in the initial goal created for the model:

```
(goal isa read-letters state attending upper-y 400 lower-y 600)
```

which includes the whole window, thus the location of any letter that is unattended will be potentially chosen initially. When the tone is encoded those slots will be updated so that only the target row's letters will be found, and the **visual-location** request also used the :attended request parameter to ensure that it finds an item which it has not attended previously (at least not one of the last 4 items attended since that is the default count of finsts).

3.4 Auditory Attention

There are a number of productions responsible for processing the auditory signal in this model and they serve as our first introduction to the audio module. Like the vision module, there are also two buffers in the audio module. The **aural-location** holds the location of an aural message and the **aural** buffer holds the representation of a sound that is attended. However, unlike the visual system we typical need only two steps to encode a sound and not three. This is because usually the auditory field of the model is not crowded with sounds and we can rely on buffer stuffing to place the sound's location into the **aural-location** buffer without having to request it. If a new sound is presented, and the **aural-location** buffer is empty, then the audio-event for that sound (the auditory equivalent of a visual-location) is placed into the buffer automatically. However, there is a delay between the initial onset of the sound and when the audio-event becomes available. The length of the delay depends on the type of sound being presented (tone, digit, word, or other) and represents the time necessary to encode its content. This is unlike the visual-locations which are immediately available.

In this task the model will hear one of the three possible tones on each trial. The default time it takes the model's audio module to encode a tone sound is .050 seconds. The **detected-sound** production responds to the appearance of an audio-event in the **aural-location** buffer:

```
(p detected-sound
   =aural-location>
   ?aural>
     state   free
   ==>
   +aural>
     event   =aural-location)
```

Notice that this production does not test the **goal** buffer. If there is a chunk in the **aural-location** buffer and the **aural** state is free this production can fire. It is not specific to this, or any task. On its RHS it makes a request to the **aural** buffer specifying the event slot. That is a request to shift attention and encode the event provided. The result of that encoding will be a chunk with slots specified by the chunk-type sound being placed into the **aural** buffer:

```
(chunk-type sound kind content event)
```

The kind slot is used to indicate the type of sound encoded which could be tone, digit, or word by default, but custom kinds of sound can also be generated for a model. The value of the content slot will be a representation of the sound heard, and how that is encoded is different for different kinds of sounds (the default encoding is that tones encode the frequency, words are encoded as strings, and digits are encoded as a number). The event slot contains a chunk which relates to the event that was used to attend the sound.

Our model for this task has three different productions to process the encoded sound chunks, one for each of the high, medium, and low tones. The following is the production for the low tone:

```
(p sound-respond-low
   =goal>
     isa      read-letters
     tone     nil
   =aural>
     isa      sound
     content  500
==>
   =goal>
     tone     low
     upper-y  500
     lower-y  520)
```

For this experiment a 500 Hertz sound is considered low, a 1000 Hertz sound medium, and a 2000 Hertz sound high. On the RHS this production records the type of tone presented in the **goal** buffer's chunk and also updates the restrictions on the y coordinates for the search to constrain it to the appropriate row (the range of which we have explicitly encoded in this production based on where the experiment code displays the items to keep things simple).

Now we will look at the trace from the above trial where the sound was initiated at .15 seconds into the trial to see how the processing of that auditory information progresses. The first action performed by the audio module occurs at time .2 seconds:

```
...
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.185   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3
0.200   AUDIO          SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0 NIL
0.235   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
```

Although the sound was initiated at .150 seconds, it takes the audio module .05 seconds to detect and encode that a tone has occurred so at time .2 seconds it stuffs the audio-event into the **aural-location** buffer since it is empty. At .235 seconds the **detected-sound** production can be selected in response to the audio event that happened. It could not be selected sooner because the **attend-high** production was selected at .185 seconds (before the tone was available) and takes 50 milliseconds to complete firing at time .235 seconds. When the **detected-sound** production completes at .285 seconds aural attention is then shifted to the sound. The next audio module event in the trace occurs at time .57 seconds when the **aural** buffer gets the chunk which represents the attended sound. So it took .285 seconds from when the request was made until the sound was attended and encoded:

```
0.285   PROCEDURAL     PRODUCTION-FIRED DETECTED-SOUND
0.320   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
0.370   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.370   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2
0.420   PROCEDURAL     PRODUCTION-FIRED ATTEND-LOW
0.505   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
0.555   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.555   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0
0.570   AUDIO          SET-BUFFER-CHUNK AURAL TONE0
0.605   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.655   PROCEDURAL     PRODUCTION-FIRED SOUND-RESPOND-MEDIUM
0.690   VISION         SET-BUFFER-CHUNK VISUAL TEXT3
```

```
0.740   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
```

The production which harvests that aural buffer chunk, **sound-respond-medium,** is then selected at .605 seconds (after the **attend-high** production completes which had been selected before the sound chunk was available) and finishes firing at .655 seconds. The next production to be selected and fire is **encode-row-and-find** at time .69 seconds because it is waiting for the **visual** buffer's chunk to be available. It encodes the last letter that was read and issues a request to find a letter that is in the target row instead of an arbitrary letter since the coordinate locations were updated by **sound-respond-medium**. Thus, even though the sound is presented at .150 seconds it is not until .690 seconds, when **encode-row-and-find** is selected, that it has any effect on the processing of the visual array.

3.5 Typing and Control

After encoding as many letters as it can the model must respond, and this is the production which initiates that:

```
(P start-report
   =goal>
     isa     read-letters
     tone    =tone
   ?visual>
     state   free
   ==>
   +goal>
     isa     report-row
     row     =tone
   +retrieval>
     status  =tone)
```

It makes a request for the goal module to create a new chunk to be placed into the **goal** buffer rather than just modifying the chunk that is currently there (as indicated by the +goal rather than an =goal). The goal module creates a new chunk immediately in response to a request, unlike the imaginal module which takes time to create a new chunk. This production also makes a **retrieval** request for a chunk from declarative memory which has the required row in its status slot (as was set by the **encode-row-and-find** production).

This production's conditions are fairly general and it can match at many points in the model's run, but we do not want it to apply as long as there are letters to be processed. We only want this rule to apply when there is nothing else to do. Each production has a quantity associated with it called its utility. The productions' utilities determine which production gets selected during conflict resolution if there is more than one that matches. We will discuss utility in more detail in later units. For now, the important thing to know is that the production with the highest utility among those that match is the one selected and fired. Thus, we can make this production less preferred by setting its utility value low. The command for setting production parameters in a model is **spp** (set production parameters). It is similar to **sgp** which is used for the general parameters as discussed earlier in the tutorial. The utility of a production is set with the :u parameter, so the following call found in the model sets the utility of the **start-report** production to -2:

```
(spp start-report :u -2)
```

The default utility for productions is 0. So, this production will not be selected as long as there are other productions with a higher utility that match, and in particular that will be as long as there is still something in the target row on the screen to be processed by the productions that encode the screen.

You may also notice that the productions that process the sound are given higher utility values than the default in the model:

```
(spp detected-sound  :u 10)
(spp sound-respond-low :u 10)
(spp sound-respond-medium :u 10)
(spp sound-respond-high :u 10)
```

That is so that the sound will be processed as soon as possible – these productions will be preferred over others that match at the same time.

Once the model starts to retrieve the letters, the following production is responsible for reporting all the letters recalled from the target row:

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?manual>
     state    free
   ==>
   +manual>
     cmd      press-key
     key      =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

This production fires when a correct item has been retrieved and the motor module is free. As actions, it presses the key corresponding to the letter retrieved and requests a **retrieval** of another letter. Notice that it does not modify the chunk in the **goal** buffer (which is the only buffer that does not get cleared by strict harvesting) and thus this production can fire again once the other conditions are met. Here is a portion of the trace showing the results of this production firing twice in succession:

```
1.160   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.160   MOTOR          PRESS-KEY KEY c
1.160   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL TEXT4-0
1.760   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.760   MOTOR          PRESS-KEY KEY r
```

Something new that you may notice in that production is the request parameter in the **retrieval** request (:recently-retrieved). We will discuss that in the next section.

When there are no more letters to be reported (a **retrieval** failure occurs because the model can not retrieve any more letters from the target row), the following production applies to indicate it is done by pressing the space bar:

```
(p stop-report
   =goal>
     isa     report-row
     row     =row
   ?retrieval>
     buffer  failure
   ?manual>
     state   free
==>
   +manual>
     cmd     press-key
     key     space
   -goal>)
```

It also clears the chunk from the **goal** buffer which results in no more productions being able to match and the run terminates.

3.6 Declarative Finsts

While doing this task the model only needs to report the letters it has seen once each. One way to do that easily is to indicate which chunks have been retrieved previously so that they are not retrieved again. However, one cannot modify the chunks in declarative memory. Modifying the chunk in the **retrieval** buffer will result in a new chunk being added to declarative memory with that modified information, but the original unmodified chunk will also still be there. Thus some other mechanism must be used.

The way this model handles that is by taking advantage of the declarative finsts built into the declarative memory module. Like the vision module, the declarative module marks items that have been retrieved with tags that can be tested in the **retrieval** request. These finsts are not part of the chunk, but can be tested with the :recently-retrieved request parameter in a **retrieval** request as shown in the **do-report** production:

```
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

If **:recently-retrieved** is specified as **nil**, then only a chunk that has not been recently retrieved (marked with a finst) will be retrieved. In this way the model can exhaustively search declarative memory for items without repeating. That is not always necessary and there are other ways to model such tasks, but it is a convenient mechanism that can be used when needed.

Like the visual system, the number and duration of the declarative finsts is also configurable through parameters. The default is four declarative finsts which last 3 seconds each, and those can be set using the :declarative-num-finsts and :declarative-finst-span parameters respectively. In this model the default of four finsts is sufficient, but the duration of 3 seconds is potentially too short because of the time it takes to make the responses. Thus in this model the span is simply set to 10 seconds to avoid potential repeats to keep the model easier to understand as an example:

```
(sgp :v t :declarative-finst-span 10)
```

3.7 Data Fitting

One can see the average performance of the model run over a large number of trials by using the function **sperling-experiment** in the Lisp version or the **experiment** function from the Python version's sperling module. It requires one parameter which indicates how many presentations of each condition should be performed. However, there are a couple changes to the model that one should make first. The first thing to change is to remove the sgp call that sets the :seed parameter which causes the model to always perform the same trial in the same way, otherwise the performance is going to be identical on every trial. The easiest way to remove that call is to place a semi-colon at the beginning of the line like this:

```
;(sgp :seed (100 0))
```

In Lisp syntax a semi-colon designates a comment and everything on the line after the semi-colon is ignored, and since the models are based on Lisp syntax you can use a semi-colon to comment out lines of text.

After making that change to the model and then reloading it the experimental will present different stimuli for each trial and the model's performance can differ from trial to trial. The other change that can be made to the model will decrease the length of time it takes to run the model through the experiment (the real time that passes not the simulated performance timing of the model). That change is to turn off the ACT-R trace so that it does not have to print out all the events as they are occurring, which is done by setting the :v parameter in the model to **nil**:

```
(sgp :v nil :declarative-finst-span 10)
```

There are also some changes which should be made to the experiment code to significantly reduce the time it takes to run the experiment. Instead of having a real window displayed the model can interact with a virtual window which is faster, but you will not be able to watch it do the task. Also, because the model is interacting with a real window for you to watch it perform the task it is also currently running in step with real time, but you can remove the real time constraint to significantly improve how long it takes to run the model through the task. The details on making those changes are not going to be covered here, but can be found in the code document for this unit.

After making the necessary changes you can run the experiment many times to see the model's average performance and comparison to the human data with respect to the number of items correctly recalled by tone onset condition. Here are the results from a run of 100 trials in both the Lisp and Python implementations:

```
? (sperling-experiment 100)
CORRELATION:  0.998
MEAN DEVIATION:  0.164

Condition    Current Participant   Original Experiment
 0.00 sec.           3.21                   3.03
 0.15 sec.           2.54                   2.40
 0.30 sec.           2.24                   2.03
 1.00 sec.           1.61                   1.50
```

```
>>> sperling.experiment(100)
CORRELATION:  0.994
MEAN DEVIATION:  0.137
Condition    Current Participant   Original Experiment
 0.00 sec.           3.14                   3.03
 0.15 sec.           2.44                   2.40
 0.30 sec.           2.24                   2.03
 1.00 sec.           1.63                   1.50
```

When it is done running the model through the experiment it prints out the correlation and mean deviation between the experimental data and the average of the 100 ACT-R simulated runs along with the results for the model and the original experiment data. You may notice that the results differ between those two runs. That is not because of the different code to implement the versions of the task, but because each run of the model varies so the average performance will also vary.

From this point on in the tutorial most of the examples and assignments will compare the performance of the model to the data collected from people doing the task to provide a measure of how well the models correspond to human performance.

3.8 The Subitizing Task

Your assignment for this unit is to write a model for a subitizing task. This is an experiment where you are presented with a set of marks on the screen (in this case Xs) and you have to count how many there are. The code to implement the experiment is in the subitize file for each of the implementations. If you load/import that file then you can run yourself through the experiment by running the subitize-experiment function from Lisp or the experiment function from the subitize module in Python and specify the optional parameter with a true value:

```
? (subitize-experiment t)

>>> subitize.experiment(True)
```

In the experiment you will be run through 10 trials and on each trial you will see from 1 to 10 objects on the screen. The number of items for each trial will be chosen randomly, and you should press the number key that corresponds to the number of items on the screen unless there are 10 objects in which case you should press 0. The following is the outcome from one of my runs through the task:

```
CORRELATION:  0.829
MEAN DEVIATION:  0.834
Items    Current Participant   Original Experiment
  1         1.70  (True )               0.60
  2         1.70  (True )               0.65
  3         1.13  (True )               0.70
  4         1.66  (True )               0.86
  5         1.28  (True )               1.12
  6         2.43  (True )               1.50
  7         2.15  (True )               1.79
  8         2.25  (True )               2.13
  9         3.70  (True )               2.15
 10         3.19  (True )               2.58
```

This provides a comparison between my data and the data from an experiment by Jensen, Reese, & Reese (1950) for the length of time (in seconds) which it takes to respond. The value in parenthesis after the time indicates whether or not the answer the participant gave was correct (T or True is correct, and NIL or False is incorrect).

3.8.1 The Vocal System

We have already seen that the default ACT-R mechanism for pressing keys can take a considerable amount of time and can vary based on which key is pressed. That could have an effect on the results of this model. One solution would be to more explicitly control the hand movements to provide faster and consistent responses, but that is beyond the scope of this unit. For this task the model will instead provide a vocal response i.e. it will speak the number of items on the screen instead of pressing a key, which is how the participants in the data being modeled also responded. This is done by making a request to the speech module (through the **vocal** buffer) and is very similar to the requests to the motor module through the **manual** buffer which we have already seen.

Here is a production from the sperling model that presses a key:

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?manual>
     state    free
   ==>
   +manual>
     cmd      press-key
     key      =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

With the following changes it would speak the response instead (note however that the **sperling** experiment is not written to accept a vocal response so it would not properly score those responses if you attempted to run the model after making these modifications):

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?vocal>
     state    free
   ==>
   +vocal>
     cmd      speak
     string   =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

The primary change is that instead of the **manual** buffer we use the **vocal** buffer. On the LHS we query the **vocal** buffer to make sure that the speech module is not currently in use:

```
?vocal>
   state    free
```

Then on the RHS we make a request of the **vocal** buffer to speak the value from the =val variable:

```
+vocal>
  cmd      speak
  string   =val
```

Like the **manual** and **visual** buffer requests we specify the cmd slot to indicate the action to perform, which in this case is to speak, and the **vocal** request requires the string slot to specify the text to be spoken. The default timing for speech acts is .15 seconds per syllable (where the number of syllables is determined solely by the length of the text to speak). That timing will not affect the model for the subitizing task since we are recording the time at which the vocal response starts not when it ends.

3.8.2 Exhaustively Searching the Visual Icon

When the model is doing this task it will need to exhaustively search the display. To make the assignment easier, the number of finsts has been set to 10 in the starting model. Thus, your model only needs to use the :attended specification in the **visual-location** requests instead of having to create a search pattern for the model to use. Of course, once you have a model working which relies upon the 10 finsts you may want to see if you can change it to use a different approach so that it could also work with the default of only four finsts.

The important issue regardless of how it searches for the items is that it must detect when there are no more locations (either none that are unattended or no location found when using a search strategy other than just the attended status). That will be signaled by a failure when a request is made of the **visual-location** buffer that cannot be satisfied. That is the same as when the **retrieval** buffer reports a failure when no chunk that matches a **retrieval** request can be retrieved. A query of the buffer for a failure is true in that situation, and the way for a production to test for that would be to have a query like this on the LHS along with any other tests that are needed for the state or task information:

```
(p no-location-found
...
   ?visual-location>
     buffer   failure
...
==>
...)
```

3.8.3 The Assignment

Your task is to write a model for the subitizing task that always responds correctly by **speaking** the number of items on the display and also fits the human data well. The following are the results from my ACT-R model:

```
CORRELATION:  0.980
MEAN DEVIATION:  0.230
Items    Current Participant   Original Experiment
  1         0.54  (T  )               0.60
  2         0.77  (T  )               0.65
  3         1.01  (T  )               0.70
  4         1.24  (T  )               0.86
  5         1.48  (T  )               1.12
  6         1.71  (T  )               1.50
  7         1.95  (T  )               1.79
  8         2.18  (T  )               2.13
  9         2.42  (T  )               2.15
 10         2.65  (T  )               2.58
```

You can see this does a fair job of reproducing the range of the data. However, the human data shows little effect of set size (approx. 0.05-0.10 seconds) in the range 1-4 and a larger effect (approx. 0.3 seconds) above 4 in contrast to this model which increases about .23 seconds for each item. The small effect for little displays probably reflects the ability to perceive small numbers of objects as familiar patterns and the larger effect for large displays probably reflects the time to retrieve counting facts. Both of those effects could be modeled, but would require ACT-R mechanisms which have not been described to this point in the tutorial. Therefore the linear response pattern produced by this model is a sufficient approximation for our current purposes, and provides a fit to the data that you should aspire to match.

There is a start to a model for this task found in the unit3 directory of the tutorial. It is named subitize-model.lisp and it is loaded automatically by the subitizing experiment code. The starting model defines chunks that encode numbers and their ordering from 0 to 10 similar to the count and addition models from unit 1 of the tutorial:

```
(add-dm (zero isa number number zero next one vocal-rep "zero")
        (one isa number number one next two vocal-rep "one")
        (two isa number number two next three vocal-rep "two")
        (three isa number number three next four vocal-rep "three")
        (four isa number number four next five vocal-rep "four")
        (five isa number number five next six vocal-rep "five")
        (six isa number number six next seven vocal-rep "six")
        (seven isa number number seven next eight vocal-rep "seven")
        (eight isa number number eight next nine vocal-rep "eight")
        (nine isa number number nine next ten vocal-rep "nine")
        (ten isa number number ten next eleven vocal-rep "ten")
        (eleven isa number number eleven)
        (goal isa count state start)
        (start))
```

In addition to the number and next slots as used for the numbers in unit 1, the number chunks also contain a slot called vocal-rep that holds the word string of the number which can be used by the model to speak it.

The model also creates a chunk-type which can be used for maintaining state information in the goal buffer:

```
(chunk-type count count state)
```

It has a slot to maintain the current count and a slot to hold the current model state. An initial chunk named goal which has a state slot value of start is also placed into the **goal** buffer in the starting model. As with the demonstration model for this unit, you may use only the **goal** buffer for holding the task information instead of splitting the representation between the **goal** and **imaginal** buffers. Also, as always, the provided chunk-types and chunks are only a recommended starting point and one is free to use other representations and control mechanisms if desired.

There are two functions provided to run the experiment for the model in each implementation. The **subitize-experiment** function in the Lisp version and the **experiment** function in the subitize module of the Python version were described above and can be called without any parameters to perform one pass through all of the trials in a random order. Because there is no randomness in the timing of the experiment and we have not enabled any variability in the model's actions, it is not necessary to run the model multiple times and average the results to assess the model's performance (however there is randomness in where the items are displayed so if you choose to use a visual search strategy other than relying on the finsts you may want to test the model over several runs to make sure there are no problems with how it searches the display). The other function is called **subitize-trial** in the Lisp version and **trial** in the subitize module of the Python version. It can be used to run a single trial of the experiment. It takes one parameter, which is the number of items to display, and it will run the model through that single trial and return a list of the time of the response and whether or not the answer given was correct:

```
? (subitize-trial 3)
(1.005 T)

>>> subitize.trial(3)
[1.005, True]
```

As with the other models you have worked with so far, this model will be reset before each trial. Thus, you do not need to have the model detect the screen change to know when to transition to the next trial because it will always start the trial with the initial goal chunk. Also, like the sperling task, this experiment starts with the ACT-R trace enabled and runs by default with a real window and in real time. If you would like to make the task complete faster you can disable the trace as described above and change it to use a virtual window and not run in real time as described in the code description document for this unit. However, you will probably want to wait until you are fairly certain that it is performing the task correctly before doing so because having the trace and being able to watch the model do the task are very useful when developing and debugging the model.

References

Jensen, E. M., Reese, E. P., & Reese, T. W. (1950). The subitizing and counting of visually presented fields of dots. *Journal of Psychology, 30*, 363-392.

Pylyshyn, Z. (1989). The role of location indexes in spatial perception: A sketch of the FINST spatial-index model. *Cognition, 32(1)*, 65-97.

Sperling, G.A. (1960). The information available in brief visual presentation [Special issue]. *Psychological Monographs, 74* (498).

Unit 3 Code Description

The experiments in this unit are more complicated than those in the previous unit, and they also involve collecting data and comparing it to existing experimental results. Many of the ACT-R commands that were used in the last unit's tasks will be seen again and there will be a few more introduced in this unit.

Sperling experiment

First we will describe the code which implements the sperling task and the new ACT-R commands will be highlighted in red.

Lisp version

The first thing that the code does is load the corresponding model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit3;sperling-model.lisp")
```

Three global variables are defined. *responses* will hold the list of keys pressed by the participant and *show-responses* is used as a flag to indicate whether or not to print out the participant's responses every trial. *sperling-exp-data* holds the results of the original experiment so that the model's performance can be compared to it.

```
(defvar *responses* nil)
(defparameter *show-responses* t)

(defvar *sperling-exp-data* '(3.03 2.40 2.03 1.50))
```

The sperling-trial function is now defined which will run one trial of the task. It takes one parameter which is the delay time at which to present the auditory cue in seconds. It presents the array of letters, waits for the responses, and returns the number of letters correctly recalled in the target row.

```
(defun sperling-trial (onset-time)
```

It starts by resetting the system to return the model to its starting state.

```
(reset)
```

Some local variables are created. Letters holds a randomly ordered list of the possible letters to display. Answers is set to nil, but will hold the list of letters which are the correct responses. Row is set to a random number between 0 and 2 and will determine which row is the correct one and which tone to present. Window is set to a window which is opened to display the task. Freq is created without a value, and will be used to hold the frequency of the tone presented.

```
(let* ((letters (permute-list '("B" "C" "D" "F" "G" "H" "J"
                                "K" "L" "M" "N" "P" "Q" "R"
                                "S" "T" "V" "W" "X" "Y" "Z")))
       (answers nil)
       (row (act-r-random 3))
       (window (open-exp-window "Sperling Experiment" :visible t))
       freq)
```

It displays the first 12 letters from the letters list in 3 rows of 4 and records the letters from the target row in a list stored in the answers variable.

```
(dotimes (i 3)
  (dotimes (j 4)
    (let ((txt (nth (+ j (* i 4)) letters)))
      (when (= i tone)
        (push txt answers))
      (add-text-to-exp-window window txt :x (+ 75 (* j 50))
                                         :y (+ 100 (* i 50))))))
```

The model is told which window to interact with using the install-device function.

```
(install-device window)
```

The freq variable is set to the appropriate value based on which row is the target.

```
(case row
  (0
   (setf freq 2000))
  (1
   (setf freq 1000))
  (2
   (setf freq 500)))
```

A tone is scheduled to occur for the model at the onset time specified, and which lasts for .5 seconds and has the appropriate frequency.

```
(new-tone-sound freq .5 onset-time)
```

To simulate the persistent visual memory we will clear the screen after a randomly chosen time between .9 and 1.1 seconds have passed. This is done by scheduling an action to happen for the model at that time, and the action we want to perform is the ACT-R clear-exp-window command.

```
(schedule-event-relative (+ 900 (act-r-random 200)) "clear-exp-window"
                              :params (list window) :time-in-ms t)
```

The variable to hold the model's responses is cleared.

```
(setf *responses* nil)
```

A command is added for the respond-to-key-press function (defined below) and then that function is set to monitor the output-key command so that we can record the keys which are pressed.

```
(add-act-r-command "sperling-response" 'respond-to-key-press
                   "Sperling task key press response monitor")
(monitor-act-r-command "output-key" "sperling-response")
```

The model is run for up to 30 seconds in real time mode.

```
(run 30 t)
```

We stop monitoring the output-key command and remove our new command.

```
(remove-act-r-command-monitor "output-key" "sperling-response")
(remove-act-r-command "sperling-response")
```

If the *show-responses* variable is true then we print out the correct answers for the trial and the responses which the model made.

```
(when *show-responses*
  (format t "~%~%answers: ~S~%responses: ~S~%" answers *responses*))
```

Finally we call the compute-score function (defined below) and return its result.

```
(compute-score answers)))
```

The compute-score function is defined. It requires one parameter which is the list of correct answers and it returns the number of correct responses that were made as recorded in the *responses* variable (ignoring duplicates).

```
(defun compute-score (answers)
  (let ((score 0))
    (dolist (x answers score)
      (when (member x *responses* :test 'string-equal)
        (incf score)))))
```

The respond-to-key-press function is defined. It is monitoring the output-key command and thus will be passed two parameters. The first will be the name of a model which makes a key press and the second will be the name of that key. If the key pressed is not the space bar then it is recorded in the *responses* variable.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (unless (string-equal key "space")
    (push key *responses*)))
```

The report-data function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be compared to the original experimental data in the *sperling-exp-data* variable and the print-results function will be called to display the results.

```
(defun report-data (data)

  (correlation data *sperling-exp-data*)
  (mean-deviation data *sperling-exp-data*)
  (print-results data))
```

The print-results function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be printed along with the data in the *sperling-exp-data* variable.

```
(defun print-results (data)

  (format t "~%Condition    Current Participant   Original Experiment~%")
  (do ((condition '(0.00 0.15 0.30 1.00) (cdr condition))
       (temp1 data (cdr temp1))
       (temp2 *sperling-exp-data* (cdr temp2)))
      ((null temp1))
    (format t " ~4,2F sec.          ~6,2F                ~6,2F~%"
              (car condition) (car temp1) (car temp2))))
```

The one-block function is defined which takes no parameters. It runs the model through one trial of each condition in the experiment in a random order and returns the list of results ordered by condition.

```
(defun one-block ()
  (let ((result nil))

    (dolist (x (permute-list '(0.0 .15 .30 1.0)))
      (push (cons x (sperling-trial x)) result))
    (mapcar 'cdr (sort result '< :key 'car))))
```

The sperling-experiment function is defined which takes one parameter. It runs the model through that specified number of blocks of the experiment averaging the results returned by one-block and then passes that average data to the report-data function defined above to compare it to the original task and print them.

```
(defun sperling-experiment (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar '+ results (one-block))))
    (report-data (mapcar (lambda (x) (/ x n)) results))))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

Then it loads the corresponding model which can perform the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit3;sperling-model.lisp")
```

Three global variables are defined. responses will hold the list of keys pressed by the participant and show-responses is used as a flag to indicate whether or not to print out the participant’s responses every trial. exp-data holds the results of the original experiment so that the model’s performance can be compared to it.

```
responses = []
show_responses = True

exp_data = [3.03,2.4,2.03,1.5]
```

The trial function is defined which will run one trial of the task. It takes one parameter which is the delay time at which to present the auditory cue in seconds. It presents the array of letters, waits for the responses, and returns the number of letters correctly recalled in the target row.

```
def trial(onset_time):
```

It starts by resetting the system to return the model to its starting state.

```
    actr.reset()
```

Some local variables are created. letters holds a randomly ordered list of the possible letters to display. answers is set to an empty list, but will hold the list of letters which are the correct responses. row is set to a random number between 0 and 2 and will determine which row is the correct one and which tone to present. window is set to a window which is opened to display the task.

```
    letters = actr.permute_list(["B","C","D","F","G","H","J",
                                 "K","L","M","N","P","Q","R",
                                 "S","T","V","W","X","Y","Z"])
    answers = []
    row = actr.random(3)
```

```
window = actr.open_exp_window("Sperling Experiment", visible=True)
```

It displays the first 12 letters from the letters list in 3 rows of 4 and records the letters from the target row in the list stored in the answers variable.

```
for i in range(3):
    for j in range(4):
        txt = letters[j + (i * 4)]
        if i == row:
            answers.append(txt)
        actr.add_text_to_exp_window(window,txt, x=(75 + (j * 50)),
                                    y=(100 + (i * 50)))
```

The model is told which window to interact with using the install-device function.

```
actr.install_device(window)
```

The freq variable is set to the appropriate value based on which row is the target.

```
if row == 0:
    freq = 2000
elif row == 1:
    freq = 1000
else:
    freq = 500
```

A tone is scheduled to occur for the model at the onset time specified, and which lasts for .5 seconds and has the appropriate frequency.

```
actr.new_tone_sound(freq,.5,onset_time)
```

To simulate the persistent visual memory we will clear the screen after a randomly chosen time between .9 and 1.1 seconds have passed. This is done by scheduling an action to happen for the model at that time, and the action we want to perform is the ACT-R clear-exp-window command.

```
 actr.schedule_event_relative(900 + actr.random(200),
                                   "clear-exp-window",
                                   params=[window],time_in_ms=True)
```

The global variable to hold the model’s responses is cleared.

```
global responses
responses = []
```

A command is added for the respond_to_key_press function (defined below) and then that function is set to monitor the output-key command so that we can record the keys which are pressed.

```
actr.add_command("sperling-response",respond_to_key_press,
                     "Sperling task key press response monitor")
    actr.monitor_command("output-key","sperling-response")
```

The model is run for up to 30 seconds in real time mode.

```
    actr.run(30,True)
```

We stop monitoring the output-key command and remove our new command.

```
    actr.remove_command_monitor("output-key","sperling-response")
    actr.remove_command("sperling-response")
```

If the show-responses variable is true then we print out the correct answers for the trial and the responses which the model made.

```
    if show_responses:
        print("answers: %s"%answers)
        print("responses: %s"%responses)
```

Finally we call the compute-score function (defined below) and return its result.

```
    return(compute_score(answers))
```

The compute_score function is defined. It requires one parameter which is the list of correct answers and it returns the number of correct responses that were made as recorded in the responses variable (ignoring duplicates).

```
def compute_score(answers):

    score = 0

    for s in responses:
        if s.upper() in answers:
            score += 1

    return(score)
```

The respond_to_key_press function is defined. It is monitoring the output-key command and thus will be passed two parameters. The first will be the name of a model which makes a key press and the second will be the name of that key. If the key pressed is not the space bar then it is recorded in the responses variable.

```
def respond_to_key_press (model,key):
    global responses

    if not(key.lower() == "space"):
        responses.append(key)
```

The report_data function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be compared to the original experimental data in the exp_data variable and the print_results function will be called to display the results.

```
def report_data(data):

    actr.correlation(data,exp_data)
    actr.mean_deviation(data,exp_data)
    print_results (data)
```

The print_results function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be printed along with the data in the exp-data variable.

```
def print_results(data):

    print("Condition    Current Participant   Original Experiment")
    for (c,d,o) in zip([0.0,0.15,0.3,1.0],data,exp_data):
        print(" %4.2f sec.          %6.2f                %6.2f"%(c,d,o))
```

The one_block function is defined which takes no parameters. It runs the model through one trial of each condition in the experiment in a random order and returns the list of results ordered by condition.

```
def one_block():

    result = []

    for t in actr.permute_list([0.0,.15,.3,1.0]):
        result.append((t,trial(t)))

    result.sort()
    return (list(map(lambda x: x[1],result)))
```

The experiment function is defined which takes one parameter. It runs the model through that specified number of blocks of the experiment averaging the results returned by one_block and then passes that average data to the report_data function defined above to compare it to the original task and print them.

```
def experiment(n):
    results=[0,0,0,0]

    for i in range(n):
        results=list(map(lambda x,y: x + y,results,one_block()))

    report_data(list(map(lambda x: x/n,results)))
```

New ACT-R commands

open-exp-window and **open_exp_window**. This was introduced in the last unit. Here we see it getting passed a keyword parameter which was not done in the last unit. That parameter is a flag as to whether the window should be visible or virtual. If the visible parameter has a true value (as appropriate for the language) then a real window will be displayed, and that is the default if not provided (which is how the previous unit used it). If it is not true (again as appropriate) then a virtual window will be used and that will be demonstrated below in the section on running the experiments faster.

new-tone-sound and **new_tone_sound**. This function takes 2 required parameters and a third optional parameter. The first parameter is the frequency of a tone to be presented to the model which should be a number. The second is the duration of that tone measured in seconds. If the third parameter is specified then it indicates at what time the tone is to be presented (measured in seconds), and if it is omitted then the tone is to be presented immediately. At that requested time a tone sound will be made available to the model's auditory module with the requested frequency and duration.

schedule-event-relative and **schedule_event_relative**. This function takes 2 required parameters and several keyword parameters (only two of which will be described here). It is used to schedule ACT-R commands to be called during the running of the ACT-R system. The first parameter specifies an offset from the current ACT-R time at which the function should be called and is measured in seconds (by default). The second parameter is the name of the ACT-R command to call specified in a string. The parameters to pass to that command are provided as a list using the keyword parameter params. If no parameters are provided then no parameters are passed to that command. If one wants to set the time using milliseconds instead of seconds then the keyword parameter time-in-ms in Lisp or time_in_ms in Python needs to be specified with a true value. By scheduling commands to be called during the running of the model it is possible to have actions occur without having to stop the running model to do so which often makes writing experiments much easier, and can also make debugging a broken model/task easier because the ACT-R stepper will pause on those scheduled actions in the same way it will for other model actions.

In this experiment we are passing the current task window to the clear-exp-window command to have it cleared at that time. If you set the :trace-detail level of the model to high (not low which is how it is initially set) then you will actually see this command being executed in the trace on a line like this:

```
 0.941   NONE  clear-exp-window (vision exp-window Sperling Experiment)
```

Since this was not generated by one of the ACT-R modules it specifies "NONE" where the module name is normally shown and then it shows the command which was evaluated and the parameter it was passed, which in this case is the representation of the experiment window.

correlation. This function takes 2 required parameters which must be equal length lists of numbers. This function computes the correlation between the two lists of numbers. That correlation value is returned. There is an optional third parameter which indicates whether or not to also print the correlation value. If the optional parameter is true or not specified then it is output, and if it is not true then it does not.

mean-deviation and **mean_deviation**. This function operates just like correlation, except that the calculation performed is the root mean square deviation between the data lists.

Subitize experiment

Now we will look at the subitizing experiment code again highlighting the new commands in red.

Lisp

The first thing that the code does is load the starting model for this task from the unit3 directory of the tutorial.

```
(load-act-r-model "ACT-R:tutorial;unit3;subitize-model.lisp")
```

It defines some global variables to hold the response the participant makes, the time at which that response occurs, and the data from the original experiment for comparison.

```
(defvar *response* nil)
(defvar *response-time* nil)

(defvar *subitize-exp-data* '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58))
```

Two functions are defined to convert a number into a string which are useful for comparing a participant's response to the correct answer. Number-to-word converts the number to the corresponding word e.g. 1 into "one", and number-to-string converts the number to the single digit string of that number e.g. 1 into "1" (with 10 being converted to the single digit "0" since that is the key a person is to press for that answer).

```
(defun number-to-word (n)
  (format nil "~r" n))

(defun number-to-string (n)
  (if (= n 10)
      "0"
    (princ-to-string n)))
```

The subitize-trial function is defined and it takes one required parameter which is the number of items to present and an optional parameter which indicates whether a person or model is doing the task. It presents one trial to either the model or a person as indicated and returns a list of two items. The first item is the response time in seconds if the response was correct or 30 if it was not and the second item indicates whether or not the response was correct (t) or incorrect (nil).

```
(defun subitize-trial (n &optional human)
```

First it resets the system to its initial state.

```
(reset)
```

It creates some local variables. points is set to the result of generate-points (defined below) which will be a list of x,y coordinates for the items in the window. window is set to an experiment window opened to present the trial. start is set to the current time (determined appropriately for a person or the model as described later) and answer is created but not set.

```
(let ((points (generate-points n))
      (window (open-exp-window "Subitizing Experiment"))
      (start (get-time (if human nil t)))
      answer)
```

It loops through the points placing an x in the display for each.

```
(dolist (point points)
  (add-text-to-exp-window window "x" :x (first point) :y (second point)))
```

The global variables which will hold the response and time are cleared.

```
(setf *response* nil)
(setf *response-time* nil)

(if human
```

If a person is doing the task and visible windows can be shown

```
(when (visible-virtuals-available?)
```

Create a command to record the person’s response using the respond-to-key-press function and then set that to monitor the output-key command.

```
(add-act-r-command "subitize-response" 'respond-to-key-press
                   "Subitize task human response")
(monitor-act-r-command "output-key" "subitize-response")
```

Convert the number of items in the trial to the corresponding key press string and record that in the answer variable.

```
(setf answer (number-to-string n))
```

Wait for the person to respond making sure to call process-events.

```
(while (null *response*)
  (process-events))
```

Stop monitoring output-key and remove the command we created.

```
(remove-act-r-command-monitor "output-key" "subitize-response")
(remove-act-r-command "subitize-response"))
```

If the model is performing the task.

```
(progn
```

Create a command to record the model's response using the record-model-speech function and then set that to monitor the output-speech command.

```
(add-act-r-command "subitize-response" 'record-model-speech
                   "Subitize task model response")
(monitor-act-r-command "output-speech" "subitize-response")
```

Convert the number of items in the trial to the corresponding word string and record that in the answer variable.

```
(setf answer (number-to-word n))
```

Tell the model which window to interact with.

```
(install-device window)
```

Run the model for up to 30 seconds in real time mode.

```
(run 30 t)
```

Stop monitoring output-speech and remove the command we created.

```
(remove-act-r-command-monitor "output-speech" "subitize-response")
(remove-act-r-command "subitize-response")))
```

If there is a response, compare it to the correct answer. If it is correct then return a list of the difference between the *response-time* and the start time converted to seconds and t, and if it is not correct return a list of 30 and nil.

```
(if (and *response* (string-equal answer *response*))
    (list (/ (- *response-time* start) 1000.0) t)
  (list 30 nil))))
```

The subitize-experiment function takes one optional parameter which indicates whether a person or model will be performing the task. It presents each of the 10 possible conditions once in a random order collecting the data, reorders the data based on the condition, and passes that data to report-data.

```
(defun subitize-experiment (&optional human)
  (let (results)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (subitize-trial items human)) results))

    (setf results (sort results '< :key 'car))
    (report-data (mapcar 'second results))))
```

The report-data function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints the comparison of the response times to the experimental data and then passes the data to the print-results function to output a table of the response times and correctness.

```
(defun report-data (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *subitize-exp-data*)
    (mean-deviation rts *subitize-exp-data*)
    (print-results data)))
```

The print-results function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints a table of the response times and correctness along with the original data.

```
(defun print-results (data)
  (format t "Items    Current Participant   Original Experiment~%")
  (dotimes (i (length data))
    (format t "~3d        ~5,2f  (~3s)              ~5,2f~%"
      (1+ i) (car (nth i data)) (second (nth i data))
      (nth i *subitize-exp-data*))))
```

The generate-points function takes 1 parameter which specifies how many points to generate. It returns a list of n randomly generated points (lists of x and y coordinates) to use for displaying the items. The points are generated such that they are not too close to each other and within the default experiment window size (300x300 pixels).

```
 (defun generate-points (n)
   (let ((points nil))
    (dotimes (i n points)
      (push (new-distinct-point points) points))))
```

The new-distinct-point function takes one parameter which is a list of points. It returns a new point that is randomly generated within the default experiment window boundary and which is not too close to any of the points on the list provided.

```
(defun new-distinct-point (points)
  (do ((new-point (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))
                  (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))))
```

```
      ((not (too-close new-point points)) new-point)))
```

The too-close function takes two parameters. The first is a point and the second is a list of points. It returns true if the first point is within 40 pixels in either the x or y direction of any of the points on the list, otherwise it returns nil.

```
(defun too-close (new-point points)
  (some (lambda (a) (and (< (abs (- (car new-point) (car a))) 40)
                         (< (abs (- (cadr new-point) (cadr a))) 40)))
        points))
```

The respond-to-key-press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press or nil if the key was pressed by a person interacting with the window and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get-time function.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (setf *response-time* (get-time nil))
  (setf *response* key))
```

The record-model-speech function is set to monitor the output-speech command. That command is called whenever a model performs a speak action and is passed the name of the model and the string which the model spoke. Here we are recording the word which is spoken and the time at which that happens as determined by the get-time function.

```
(defun record-model-speech (model string)
  (declare (ignore model))

  (setf *response-time* (get-time t))
  (setf *response* string))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

Then it loads the starting model for this task from the unit3 directory of the tutorial.

```
actr.load_act_r_model("ACT-R:tutorial;unit3;subitize-model.lisp")
```

It defines some global variables to hold the response the participant makes, the time at which that response occurs, and the data from the original experiment for comparison.

```
response = False
response_time = False

exp_data = [.6,.65,.7,.86, 1.12,1.5,1.79,2.13,2.15,2.58]
```

Two functions are defined to convert a number into a string which are useful for comparing a participant's response to the correct answer. number_to_word converts the number to the corresponding word e.g. 1 into 'one', and number_to_string converts the number to the single digit string of that number e.g. 1 into "1" (with 10 being converted to the single digit "0" since that is the key a person is to press for that answer).

```
def number_to_word (n):
    map = ['','one','two','three','four','five','six','seven','eight','nine','ten']
    return map[n]

def number_to_string (n):
    if n == 10:
        return "0"
    else:
        return str(n)
```

The trial function is defined and it takes one required parameter which is the number of items to present and an optional parameter which indicates whether a person or model is doing the task. It presents one trial to either the model or a person as indicated and returns a list of two items. The first item is the response time in seconds if the response was correct or 30 if it was not and the second item indicates whether or not the response was correct (True) or incorrect (False).

```
def trial (n,human=False):
```

First it resets the system to its initial state.

```
    actr.reset()
```

It creates some local variables. points is set to the result of generate_points (defined below) which will be a list of x,y coordinates for the items in the window. window is set to an experiment window opened to present the trial, and start is set to the current time (determined appropriately for a person or the model as described later).

```
    points = generate_points(n)
    window = actr.open_exp_window("Subitizing Experiment")
    start = actr.get_time(not(human))
```

It loops through the points placing an x in the display for each.

```
    for point in points:
        actr.add_text_to_exp_window(window, "x", x=point[0], y=point[1])
```

The global variables which will hold the response and time are cleared.

```
global response,response_time

response = ''
response_time = False

if human:
    if actr.visible_virtuals_available():
```

If a person is doing the task and a visible window can be displayed.

Create a command to record the person's response using the respond_to_key_press function and then set that to monitor the output-key command.

```
        actr.add_command("subitize-response",respond_to_key_press,
                         "Subitize task human response")
        actr.monitor_command("output-key","subitize-response")
```

Convert the number of items in the trial to the corresponding key press string and record that in the answer variable.

```
        answer = number_to_string(n)
```

Wait for the person to respond making sure to call process-events.

```
        while response == '':
            actr.process_events()
```

Stop monitoring output-key and remove the command we created.

```
        actr.remove_command_monitor("output-key","subitize-response")
        actr.remove_command("subitize-response")

else:
```

If the model is performing the task.

Create a command to record the model's response using the record_model_speech function and then set that to monitor the output-speech command.

```
    actr.add_command("subitize-response",record_model_speech,
                     "Subitize task model response")
    actr.monitor_command("output-speech","subitize-response")
```

Convert the number of items in the trial to the corresponding word string and record that in the answer variable.

```
answer = number_to_word(n)
```

Tell the model which window to interact with.

```
actr.install_device(window)
```

Run the model for up to 30 seconds in real time mode.

```
actr.run(30,True)
```

Stop monitoring output-speech and remove the command we created.

```
actr.remove_command_monitor("output-speech","subitize-response")
actr.remove_command("subitize-response")
```

If there is a response, compare it to the correct answer. If it is correct then return a list of the difference between the response_time and the start time converted to seconds and True, and if it is not correct return a list of 30 and False.

```
if response != '' and response.lower() == answer.lower():
    return [(response_time - start) / 1000.0, True]
else:
    return [30, False]
```

The experiment function takes one optional parameter which indicates whether a person or model will be performing the task. It presents each of the 10 possible conditions once in a random order collecting the data, reorders the data based on the condition, and passes that data to report_data.

```
def experiment(human=False):

    results = []
    for items in actr.permute_list([10,9,8,7,6,5,4,3,2,1]):
      results.append((items,trial(items,human)))

    results.sort()
    report_data(list(map(lambda x: x[1],results)))
```

The report_data function takes one parameter which is a list of response lists as are returned by the subitize_trial function. It prints the comparison of the response times to the experimental data and then passes the data to the print_results function to output a table of the response times and correctness.

```
def report_data(data):

    rts = list(map(lambda x: x[0],data))
    actr.correlation(rts,exp_data)
    actr.mean_deviation(rts,exp_data)
    print_results(data)
```

The print_results function takes one parameter which is a list of response lists as are returned by the subitize_trial function. It prints a table of the response times and correctness along with the original data.

```
def print_results(data):

    print("Items    Current Participant   Original Experiment")
    for count, d, original in zip(range(len(data)),data,exp_data):
        print("%3d        %5.2f  (%-5s)            %5.2f" %
              (count+1,d[0],d[1],original))
```

The generate_points function takes 1 parameter which specifies how many points to generate. It returns a list of n randomly generated points (lists of x and y coordinates) to use for displaying the items. The points are generated such that they are not too close to each other and within the default experiment window size (300x300 pixels).

```
def generate_points(n):
    p=[]
    for i in range(n):
        p.append(new_distinct_point(p))
    return p
```

The new_distinct_point function takes one parameter which is a list of points. It returns a new point that is randomly generated within the default experiment window boundary and which is not too close to any of the points on the list provided.

```
def new_distinct_point(points):

    while True:
        x = actr.random(240)+20
        y = actr.random(240)+20
        if not(any(too_close(x,y,p) for p in points)):
            break;
    return [x,y]
```

The too_close function takes three parameters. The first and second are the x and y positions for a proposed point and the second is a list of points. It returns True if the proposed point would be within 40 pixels in either the x or y direction of any of the points on the list, otherwise it returns False.

```
def too_close (x,y,p):
    if (abs(x-p[0]) < 40) and (abs(y-p[1]) < 40):
        return True
    else:
        return False
```

The respond_to_key_press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press or None if the key was pressed by a person interacting with the window and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get_time function.

```
def respond_to_key_press (model,key):
    global response,response_time

    response_time = actr.get_time(False)
    response = key
```

The record_model_speech function is set to monitor the output-speech command. That command is called whenever a model performs a speak action and is passed the name of the model and the string which the model spoke. Here we are recording the word which is spoken and the time at which that happens as determined by the get_time function.

```
def record_model_speech (model,string):
    global response,response_time

    response_time = actr.get_time(True)
    response = string
```

New commands

get-time and **get_time**. This function takes an optional parameter and it returns the current time in milliseconds. If the optional parameter is not specified or specified as a true value, then the current ACT-R simulated time is returned. If the optional parameter is specified as a non-true value then the time is taken from a real time clock.

output-speech. The output-speech command in ACT-R is very similar to the output-key command which we used in the previous unit. The output-speech command is called automatically by a microphone device (which is installed automatically when the AGI window device is installed) whenever a model performs a speak action using the **vocal** buffer. The command is executed at the time when the sound of that speech starts to occur i.e. it is essentially the same time that would be recorded if we were using a microphone to detect a person's speech output. It is passed two values which are the name of the model which is speaking and the string containing the text being spoken.

Response recording note

As was mentioned in the last unit the monitoring functions are called in a separate thread from the main task execution. As will be the case throughout the tutorial we are not using any special protection for accessing the globally defined variables in the different threads to keep the example code simple, but this time there is actually the possibility for a problem since we are setting two different variables in the monitoring function and both are used in the main thread. Without protecting them it is possible for the code in the main thread to try and use them both after only one has been set which could result in an error. To avoid that here we set the response-time variable first since the main thread is waiting for the response variable to change before it tries to use the response-time value. That is sufficient to avoid problems in this task (we could have also waited for both to be set before continuing as an alternative), but the best solution would really be to use appropriate thread protection tools.

Buffer stuffing

The buffer stuffing mechanism was introduced in this unit, and with regard to the **visual-location** buffer it mentioned that one can change the default conditions that are checked to determine which item (if any) will be stuffed into the buffer. The ACT-R command which can be used to change that is called **set-visloc-default.** The specification that you pass to it is the same as you would specify in a request to the **visual-location** buffer in a production. Here are a few examples:

```
(set-visloc-default :attended new screen-x lowest)

(set-visloc-default screen-x current > screen-y 100)

(set-visloc-default kind text color red width highest)
```

set-visloc-default – This command sets the conditions that will be used to select the visual feature that gets stuffed into the **visual-location** buffer. When the visual scene changes for the model (denoted by the proc-display event which is only shown when the :trace-detail parameter is set to high) if the **visual-location** buffer is empty a visual-location that matches the conditions specified by this command will be placed into the **visual-location** buffer. Effectively, what happens is that when the proc-display event occurs, if the **visual-location** buffer is empty, a **visual-location** request is automatically executed using the specification indicated with set-visloc-default.

Speeding up the experiments

Because it is often necessary to run a model multiple times and average the results for data comparison, running the model quickly can be important. One thing that can greatly improve the time it takes to run the model (i.e. the real time that passes not the simulated time which the model experiences) is to have it interact with a virtual window instead of displaying a real window on the computer. The virtual windows are an abstraction of a window interface that is internal to ACT-R, and from the model's perspective there is no difference between a virtual window and a real window which is generated by the AGI commands. Another significant factor with respect to how long it takes is whether or not the model is being run in real time mode. In real time mode the model's actions are synchronized with the actual passing of time, but when it is not in real time mode it uses its own simulated clock which can run much faster than real time. Typically, when the model is running with a real window it is also running in real time so that one can actually watch it performing the task.

Since it is usually very helpful to use a real window when creating a model for a task and debugging any problems which occur when running it all of the AGI tools for building and manipulating windows work exactly the same for real windows and virtual windows. All that is necessary to switch between them is one parameter when the window is opened. Thus, you can build the experiment and debug the model using a real window that you can see, and then with one change make the window virtual and run the model more quickly for data collection.

The experiment code as provided for both of the tasks in this unit uses a real window and the model is run in real time mode so that you can watch it interact with that window.

To change the windows in those tasks to virtual windows requires changing the call to open the window to specify the visible parameter as not true (**nil** in Lisp and **False** in Python).

Here is the initial code from the sperling tasks:

Lisp

```
(let* (...
       (window (open-exp-window "Sperling Experiment" :visible t))
```

Python

```
    window = actr.open_exp_window("Sperling Experiment", visible=True)
```

Here is what would need to be changed to make those virtual windows instead:

Lisp

```
(let* (...
       (window (open-exp-window "Sperling Experiment" :visible nil))
```

Python

```
    window = actr.open_exp_window("Sperling Experiment", visible=False)
```

A similar change could be made in the subitize experiment.

To change the code to run the models in simulated time requires not specifying the second parameter as true in the call to run.

Here is the call that currently exists in both experiments:

Lisp

```
(run 30 t)
```

Python

```
actr.run(30,True)
```

Removing that second parameter will run the models in simulated time:

Lisp

```
(run 30)
```

Python

```
actr.run(30)
```

Something else which can improve the time which it takes to run a task is to turn off any output which it generates. As was mentioned in the unit you can turn the model trace off by setting the :v parameter to **nil**. If there is any other output in the experiment code as the task is running you will also want to disable that.

For the sperling experiment the given code prints out the correct answers and model responses for each trial, and turning that off will help if you want to run a lot of trials to see the average performance. In that code we have already added a variable which is used as a flag to control that output. The global variables *show-responses* and show_responses control whether that information is printed. In the given code they are set to true values:

Lisp

```
(defparameter *show-responses* t)
```

Python

```
show_responses = True
```

Change those to non-true values will eliminate that output:

Lisp

```
(defparameter *show-responses* nil)
```

Python

```
show_responses = False
```

Adding a variable to control whether any output from the experiment is printed can be a handy thing to add when creating tasks for models that you want to run many times.

One last thing to note is that if you change the code in the experiment file then you will need to load that file again to have the changes take effect. For the Lisp versions of the task you can simply load the file again to have the changes take effect, but for Python importing the module again in the same session will not work. One way to deal with that is to also import the importlib module. That provides a reload function which can be used to force the module to be reimported to reflect the changes and here is an example of using that to load the sperling module again after changing it:

```
>>> import importlib
>>> importlib.reload(sperling)
```

Debugging a Model Which Has Perceptual and Motor Components

In this text we are going to implement a model which can perform a simple experiment which requires visual and motor actions. While doing so we will encounter some problems which we will walk through debugging. We will also discuss some of the additional things that one should be careful about with respect to visual tasks in particular and also introduce some additional tools in the ACT-R Environment which may be helpful in debugging and analyzing models.

The Task

The experiment which this model will have to perform involves the following steps:

- A letter is presented on the screen for between 1.5 and 2.5 seconds
- After the letter goes away either the word “next” or the word “previous” is displayed
- When the prompt is displayed the participant must type the letter which was presented followed by the appropriate letter in the alphabet based on the prompt

The perceptual-motor-issues-model.lisp file in this unit contains the model which is supposed to perform this task, and the task is implemented in the pm-issues.lisp and pm_issues.py files in the corresponding directory. To run the task you need to call the pm-issues-task function in Lisp or the task function in the pm_issues module in Python. Those functions have one optional parameter which if provided should be either the string “next” or “previous” to pick which prompt to display, or if neither is given a random prompt will be chosen for the trial. The return value from the function will be a list of the prompt displayed and an indication of whether the task was completed successfully or not. We will not discuss the experiment code here because it does not use any new ACT-R commands. It should be loaded/imported just like all of the other experiments seen in the tutorial and it will automatically load the corresponding model.

The Model Design

For most models, including this one, there are two important pieces to the model’s design. The first is how to represent the knowledge necessary for the model to be able to perform the task, and the other is the steps the model will perform to actually do the task. How the knowledge is represented for the model will affect how the model has to perform the task, and knowing what the model needs to do will affect what needs to be encoded in the knowledge representation. In general, these two design issues are intertwined and one will typically need to work on both of them together when starting the planning for the model. Below we will describe those two pieces of the model we have written for this task along with some explanation as to why we have made some of the choices we did. As we run the model and encounter problems we may find that our initial design choices are not sufficient to perform the task and thus we will have to adjust our design.

Knowledge representation

Because this model is performing a very simple task, we are not concerned with fitting human performance data, and we are only using the symbolic level of ACT-R's declarative memory (we have not yet seen the subsymbolic level in the tutorial) we can choose a representation which should make the modeling task easier. If we were concerned about fitting human performance, we would have to consider the consequences of the representation more thoroughly and would likely require something more involved than what we will use here.

This model needs to know about letters of the alphabet and their ordering. We will represent that in chunks in the model's declarative memory. The first choice to make is how we will distinguish letters, and we will use the simple assumption that each letter will be represented as a separate chunk using a chunk-type called letter. Now we have to decide on what slots the letter type needs and what information will be contained in those slots. Since this model will be reading a letter from the screen and typing keys it will be important to have the letter's visual representation included in the chunk as well as a representation which can be used to type the letter. Both the visual and motor representations use a string to represent a letter. Thus that single representation is all we need to have in the chunk and we will store it in a slot called name. The other thing which the model needs to be able to determine is the next and previous letter of the alphabet given a particular letter. There are many ways that one could represent that, but because we are writing a simple symbolic model we will explicitly encode that information in the chunks for a given letter in slots called next and previous. In fact, to make things even easier for the model we will encode the next and previous information using the same perceptual/motor representation as we do for the name of the letter (a string). Here is what the letter chunk-type and a chunk representation for the letter B look like in the model:

```
(chunk-type letter name next previous)

(b isa letter name "b" next "c" previous "a")
```

A more plausible model would likely represent the next and previous values with a reference to the other chunks instead of directly encoding the perceptual representations. In fact, it might even only encode the next value instead of both next and previous if we believe that most people only encode the alphabet in the forward direction. After we work through this example, as an exercise, you may want to try changing the model's representation to something like that and see if you can then adjust the model so that it can still do the task.

We also need a way to represent the information needed to perform the task. Because this is a very simple task, we are not going to use a goal chunk to hold explicit state information and will instead rely on the perceptual input, buffer contents, and module states to determine the state of the model. We will however create a chunk to maintain the letter which we have read from the screen in the **imaginal** buffer. Since that letter is the only information we need in that chunk the chunk-type only needs that one slot and we can create a new type called task to use:

```
(chunk-type task letter)
```

Actions to perform

Now we will describe how we want our model to perform the task. As noted above we are not going to use an explicit goal state to control the model. Instead we will rely on the **visual-location** buffer stuffing mechanism to have the model know when the screen changes and use the contents of the buffers and states of the modules to determine what to do next. Here is the high-level description of the steps which the model will perform:

- When it detects a letter on the screen attend it and record it in the **imaginal** buffer
- When it sees next or previous on the screen press the current key and retrieve the appropriate letter chunk from declarative memory
- Once a chunk is retrieved press the appropriate key

There are other ways one could choose to perform this task, and as with the representation issues noted above, after working through the debugging of this model you may want to consider other ways of performing the task and try to model them.

To implement that sequence of actions we have written five productions. This is what each production is intended to do:

find-letter – responds to the appearance of the letter due to buffer stuffing of the **visual-location** buffer and then requests a visual attention shift to the letter and create a new chunk in the **imaginal** buffer

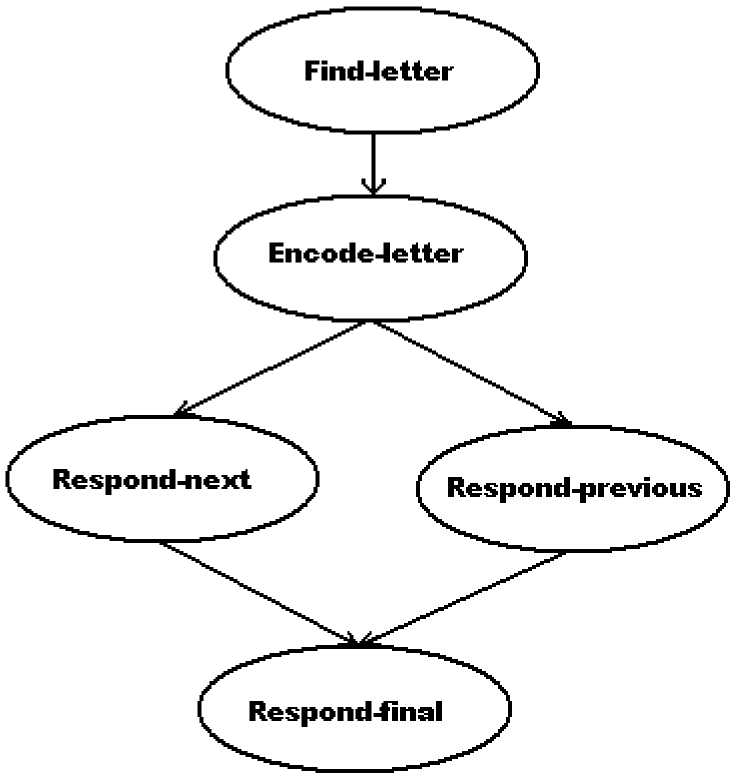
encode-letter – when the chunks resulting from the actions of find-letter are available in the **imaginal** and **visual** buffers update the **imaginal** buffer with the letter that is seen

respond-next – when the model sees the word “next” on the screen press the current letter’s key and make a retrieval request for the letter which occurs after the current one

respond-previous – when the model sees the word “previous” on the screen press the current letter’s key and make a retrieval request for the letter which occurs before the current one

respond-final – when a letter chunk has been retrieved press the corresponding key

This is how we expect them to fire to do the task where the choice of whether it is respond-next or respond-previous depends on the prompt displayed:



If you look over the productions you may see some potential problems in them or with the overall design of the model, but please don't get ahead of the exercise and just leave them alone until we encounter the problems during the testing walkthrough below.

Load and Run the initial model

There are no warnings when this model is loaded because we have turned off the style warnings so that we can focus on particular issues in the model. There is a section at the end where we show the style warnings which would be displayed if they were enabled and describe how seeing those may have affected how we worked through fixing the model. Since there are no syntax errors or other problems which we must fix before trying to run it we will run the model to see how it performs. To keep the testing consistent we will run it through trials for the "next" item until we have that working and then move on to testing the "previous" trials. Also, for consistency, we have set a seed parameter in the model. That way it will always be seeing the same letter and perform the same way. Once we are satisfied with its performance with the seed fixed we will remove that and test it under more variable conditions.

Here is the trace we get when we run the model with either of these functions as appropriate:

```
? (pm-issues-task "next")
```

```
>>> pm_issues.task('next')
```

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
```

```
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CLEAR-BUFFER IMAGINAL
2.225   PROCEDURAL     CONFLICT-RESOLUTION
2.310   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.310   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.310   PROCEDURAL     CONFLICT-RESOLUTION
2.425   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK1
2.425   PROCEDURAL     CONFLICT-RESOLUTION
2.475   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
2.475   PROCEDURAL     CLEAR-BUFFER VISUAL
2.475   PROCEDURAL     CONFLICT-RESOLUTION
2.475   ------         Stopped because no events left to process
```

The return values from either of those indicates that it did not respond correctly because the second value is not true, with the Lisp version returning ("next" NIL) and the Python version returning ['next', False]. Looking at the trace we see that the productions did not fire in the sequence we expected. There are a couple of things we could investigate, but we will start by determining where the model first deviated from our plan and address that first.

The first issue appears to be at time 2.225 when find-letter fires a second time. Here is the find-letter production:

```
(p find-letter
   =visual-location>
   ?visual>
     state       free
 ==>
   +visual>
     cmd         move-attention
     screen-pos  =visual-location
   +imaginal>
)
```

Before trying to fix the production we should make sure we understand why it fired again. If we look at the conditions of the production all it requires to fire is that there is a chunk in the **visual-location** buffer and that the **visual** buffer query of the vision module indicates it is free. Looking at the trace we see that at time 2.090 when the display of the "next" prompt occurs there is a new chunk placed into the **visual-location** buffer. That happens because every time there is a change to the screen the **visual-location** buffer will be stuffed with a chunk if the buffer is empty. At time 2.175 we see that the vision module completes the re-encoding of the display and thus at that point the module is free (we could check that by using the Stepper and inspecting the buffer status at that time, but for now we will assume that's the case). Those are the only two conditions for the find-letter production and since they are satisfied it can be selected and fired again.

There are a few things we can do to correct that at this point: we could add additional tests to the production so that it only fires when we want it to (the start of the task), we could change it or other productions so that its conditions are not satisfied at time 2.175, or we could consider redesigning the steps that we want the model to perform and rewrite this and other productions. As a first step, we will take the first of those options and adjust this production to only fire when we expect it to. After testing things further we may find that that is not sufficient and other changes to our design and/or the model's productions are necessary, but progressing in small steps is often a good way to start.

Now we will consider what we can add to the production to make it only fire at the start. One option would be to add a **goal** chunk to the model so that we could explicitly mark a start state, but we would like to avoid doing that if possible because not having a **goal** chunk was part of our design. Thus, we need to find something else which we can test. One place to look for something like that is in the actions of the production itself – what does it do to change things that can be tested to prevent it from firing again? A good candidate for that would be the **imaginal** request since that is a change in the model which we expect to only occur once, whereas the **visual** buffer is going to be used in multiple places and thus is not a change unique to this production. This production is making a request to put a chunk into the **imaginal** buffer (all requests to the **imaginal** buffer are to create a chunk, even if there are no slots specified) and prior to that the buffer will be empty. If we test that the **imaginal** buffer is empty in the conditions of find-letter that might be sufficient to prevent it from firing again later when we don't want it to. We could just make that change and run the model again to see if it'll work, but instead we will first run the model again and use the Stepper to see if that change will help at time 2.175 when the production is selected the second time. [Because this is such a small model which runs quickly we don't really need to perform that verification because we could determine it from the trace or really just try it and see what happens, but in the interest of completeness we will do so because in larger or more complicated models it may not be as easy to determine.]

To perform the test we will open the Stepper and then run the task again. Since we know what time the production is selected, the conflict-resolution at time 2.175, we can use the "Run Until" button in the Stepper to advance the model to that time and then step until the conflict-resolution event is the next one. Once we are there we can open a Buffers window and look at the **imaginal** buffer. Looking at the contents we see that it does indeed have a chunk in it:

```
IMAGINAL: CHUNK0-0
```

```
CHUNK0-0
   LETTER  "n"
```

Therefore adding a test that the **imaginal** buffer is empty to find-letter should help. Here is the new find-letter production with a query for the imaginal buffer being empty added:

```
(p find-letter
   =visual-location>
   ?visual>
     state        free
   ?imaginal>
     buffer       empty
 ==>
   +visual>
     cmd          move-attention
     screen-pos   =visual-location
   +imaginal>
)
```

We need to save that change and then reload the model.

Second version of the model

Here is the trace we get when we run the updated model:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.590   PROCEDURAL     CONFLICT-RESOLUTION
2.590   ------         Stopped because no events left to process
```

We don’t have a second firing of find-letter, but the model still doesn’t do the task correctly. We are expecting the respond-next production to fire at this point, but it does not. Since the model is stopped we can immediately use whynot to find out what the issue is. We can do that from the Procedural tool in the Environment or call the command directly from the ACT-R prompt or Python (here we assume that the actr module has been imported at the Python prompt):

```
? (whynot respond-next)

>>> actr.whynot('respond-next')
```

Here is the result of that regardless of the mechanism used to access it:

```
Production RESPOND-NEXT does NOT match.
(P RESPOND-NEXT
   =IMAGINAL>
       LETTER =LETTER
   =VISUAL>
       VALUE "next"
   ?MANUAL>
       STATE BUSY
 ==>
   +RETRIEVAL>
       PREVIOUS =LETTER
   +MANUAL>
       CMD PRESS-KEY
       KEY =LETTER
)
It fails because:
The STATE BUSY query of the MANUAL buffer failed.
```

Looking at the reason given and the production it should be fairly obvious that the issue is a mistake in the production. We should be testing that the **manual** buffer’s module state is free instead of busy. If this were a more complicated model that may not be so obvious, and in that situation we would likely want to investigate that further. To do so we would use the “Buffers” tool in the Environment to view the status information or the buffer-status command to show us all of the current status information for the given buffer/module and we may need to do so in conjunction with the Stepper to see how it changes as the model progresses. In this case we don’t need to do so, but here is what it shows for the **manual** buffer at that time:

```
MANUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy             : NIL
  state error            : NIL
  preparation free       : T
  preparation busy       : NIL
  processor free         : T
  processor busy         : NIL
  execution free         : T
  execution busy         : NIL
  last-command           : NONE
```

There we can see that the state busy query is NIL at this time whereas the state free query is T. We need to change that test from busy to free in the production, save the model, and reload it.

```
(p respond-next
   =imaginal>
     isa         task
     letter      =letter
   =visual>
     isa         visual-object
     value       "next"
   ?manual>
     state       free
 ==>
   +retrieval>
     isa         letter
     previous    =letter
   +manual>
     cmd         press-key
     key         =letter
)
```

Model version 3

Here is the trace we get from running the model now:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
```

```
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.225   PROCEDURAL     CLEAR-BUFFER IMAGINAL
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.225   PROCEDURAL     CLEAR-BUFFER MANUAL
2.225   MOTOR          PRESS-KEY KEY n
2.225   DECLARATIVE    start-retrieval
2.225   DECLARATIVE    RETRIEVED-CHUNK O
2.225   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL O
2.225   PROCEDURAL     CONFLICT-RESOLUTION
2.275   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
2.275   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.275   PROCEDURAL     CLEAR-BUFFER VISUAL
2.275   PROCEDURAL     CLEAR-BUFFER IMAGINAL
2.275   PROCEDURAL     CONFLICT-RESOLUTION
2.325   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.325   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.325   PROCEDURAL     CLEAR-BUFFER MANUAL
2.325   MOTOR          PRESS-KEY KEY o
#|Warning: Module :MOTOR jammed at time 2.325 |#
2.325   PROCEDURAL     CONFLICT-RESOLUTION
2.360   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.360   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.360   PROCEDURAL     CONFLICT-RESOLUTION
2.475   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK1
2.475   PROCEDURAL     CONFLICT-RESOLUTION
2.525   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
2.525   PROCEDURAL     CLEAR-BUFFER VISUAL
2.525   PROCEDURAL     CONFLICT-RESOLUTION
2.625   PROCEDURAL     CONFLICT-RESOLUTION
2.775   PROCEDURAL     CONFLICT-RESOLUTION
2.775   ------         Stopped because no events left to process
```

The model still did not complete the task correctly, but it does appear to have fired the productions we expected in order (we will ignore the extra productions fired at the end of the run for now) and it attempted to press the correct keys: n and o. However the warning that is printed at time 2.325 seems to be a problem:

```
#|Warning: Module :MOTOR jammed at time 2.325 |#
```

A module gets "jammed" when there are multiple concurrent requests which it is unable to process. That is usually not something which the model should do, thus eliminating the cause of that warning seems like the next step to take. Looking at the trace, the respond-final production is the last one to fire before the warning and since we know that that production is supposed to press a key that makes it the likely candidate for having caused the problem. Before looking at the production itself, we will first look at the state of the motor module at the time that production fires. To do that we will open the Stepper again, start the task, and then pick production for the Run Until option, enter respond-final, and then hit the Run Until button. The model will then be stopped just before the production fires and we can open the Buffers tool to look at status information to see the motor module's state as reported by the **manual** buffer:

```
MANUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : NIL
  state busy            : T
  state error           : NIL
  preparation free      : NIL
  preparation busy      : T
  processor free        : NIL
  processor busy        : T
  execution free        : T
  execution busy        : NIL
  last-command          : PRESS-KEY
```

There we see that the module is busy at that time and respond-final should not be making a request to the **manual** buffer because it is not ready. Here is the text of our respond-final production:

```
(p respond-final
   =retrieval>
     isa         letter
     name        =letter
   ==>
   +manual>
     cmd         press-key
     key         =letter
)
```

It does not have a condition to make sure that the motor module is not busy, but because it is making a **manual** buffer request it should have such a check to avoid the jamming which occurs. Here is an updated version of that production which has a query of the state to avoid the jamming:

```
(p respond-final
   =retrieval>
     isa         letter
     name        =letter
   ?manual>
     state       free
   ==>
   +manual>
     cmd         press-key
     key         =letter
)
```

We need to save that change and again reload the model. One thing to note however is that you will need to close the Stepper to allow the model to finish running or use the stop button to end the run before you can reload the model because you can't redefine a model while it is running and if you try you will get an error message (the message will vary depending on how you try to reload the model file).

Model version 4

Here is the trace of the model running after that change:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.225   PROCEDURAL     CLEAR-BUFFER IMAGINAL
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.225   PROCEDURAL     CLEAR-BUFFER MANUAL
2.225   MOTOR          PRESS-KEY KEY n
2.225   DECLARATIVE    start-retrieval
2.225   DECLARATIVE    RETRIEVED-CHUNK O
2.225   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL O
2.225   PROCEDURAL     CONFLICT-RESOLUTION
2.275   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
2.275   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.275   PROCEDURAL     CLEAR-BUFFER VISUAL
2.275   PROCEDURAL     CLEAR-BUFFER IMAGINAL
2.275   PROCEDURAL     CONFLICT-RESOLUTION
2.360   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.360   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.360   PROCEDURAL     CONFLICT-RESOLUTION
2.475   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK1
2.475   PROCEDURAL     CONFLICT-RESOLUTION
2.525   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
2.525   PROCEDURAL     CLEAR-BUFFER VISUAL
2.525   PROCEDURAL     CONFLICT-RESOLUTION
2.625   PROCEDURAL     CONFLICT-RESOLUTION
2.775   PROCEDURAL     CONFLICT-RESOLUTION
2.825   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
```

```
2.825   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.825   PROCEDURAL     CLEAR-BUFFER MANUAL
2.825   MOTOR          PRESS-KEY KEY o
2.825   PROCEDURAL     CONFLICT-RESOLUTION
2.975   PROCEDURAL     CONFLICT-RESOLUTION
3.025   PROCEDURAL     CONFLICT-RESOLUTION
3.125   PROCEDURAL     CONFLICT-RESOLUTION
3.275   PROCEDURAL     CONFLICT-RESOLUTION
3.275   ------         Stopped because no events left to process
```

and the return value indicates that the model performed the task correctly. However, if we look at the trace thoroughly we see that there are still unexpected firings of the find-letter and encode-letter productions. So, while the model has performed the task correctly, it still isn't running the way we expect it to. We will have to again look into why find-letter is firing unexpectedly.

Here is our current find-letter production which has the additional constraint that the imaginal buffer is empty which we added previously to prevent it from firing when we didn't want it to:

```
(p find-letter
   =visual-location>
   ?visual>
     state       free
   ?imaginal>
     buffer      empty
 ==>
   +visual>
     cmd         move-attention
     screen-pos  =visual-location
   +imaginal>
)
```

So the question is why is it now firing at time 2.275? We could use the Stepper to get the model up to that point and watch what happens with the Buffers tool, which you may find to be a useful exercise for practice, but we can also look at the trace for clues. Looking at the trace shows this event at time 2.225:

```
     2.225   PROCEDURAL             CLEAR-BUFFER IMAGINAL
```

which indicates that a production has cleared the **imaginal** buffer. As we saw the last time we adjusted this production the screen change is resulting in the **visual-location** buffer being stuffed with a chunk. Since there are no visual requests pending at that time that means that all of the conditions in the production are again satisfied and it can be selected to fire.

The first question raised here is why does the **imaginal** buffer get cleared at time 2.225? Looking at the trace, the respond-next production is the one which caused that action to occur because it's the production which fired at the same time as the buffer was cleared. Here is the text of that production:

```
(p respond-next
   =imaginal>
     isa         task
     letter      =letter
   =visual>
     isa         visual-object
     value       "next"
   ?manual>
     state       free
 ==>
   +retrieval>
     isa         letter
     previous    =letter
   +manual>
     cmd         press-key
     key         =letter
)
```

The reason why that causes the **imaginal** buffer to be cleared is the strict harvesting mechanism – if a production tests a buffer on the LHS and does not modify the chunk in that buffer on the RHS then it will automatically be cleared.

Now we have to decide how we are going to fix this in the model. There are a lot of options available and we should consider the possibilities and their implications instead of just applying the first option that comes to mind.

One possibility would be to abandon our design plan of not using a goal and embed explicit goal states into all of the productions. That would definitely allow us to avoid these unexpected production firings. The downside is that the model then becomes less flexible since it must follow those states. In the task which we are modeling here that would not be a serious problem, but in other tasks flexibility is necessary and for the purpose of this exercise we would like to keep the model flexible as an example.

Another option would be to find another automatic state indicator, like the buffer being empty which we used before, that we could add to find-letter to prevent it from firing now. Given the overall design of our task however (which has very little in the way of state changes) and the fact that we are already testing conditions on both of the buffers for which the find-letter production performs actions (the state that it changes directly) this doesn't seem like a good path to go down. While we may be able to find some other implicit state test that we could perform to block it from matching at time 2.225 that's likely just going to push the problem off to yet another time for which we will have to find another state test to add.

Instead of finding another state marker to test, we could modify other productions which fire so that they don't create the state which is problematic. In particular, if the **imaginal** buffer were not cleared then the existing conditions in the find-letter production would prevent it from not firing again. Based on the design of our model the **imaginal** buffer does not need to be cleared and thus this seems like it might be a good option. In other models however clearing of the buffer may be important because it might be necessary for learning (as we'll see in unit 4) or we may need to clear it to put a different chunk in there.

Something else to consider is that perhaps the overall design we've chosen for performing the task itself needs to be modified. We may not have chosen a sequence of actions which the model can perform to adequately complete this task. Often when building models one may want to reevaluate the initial design. Some reasons for that would be because of unexpected situations which are discovered that the design did not address, because one finds that there were assumptions made in the design which weren't apparent before trying to run it, or perhaps because the design leads to a model which is unable to meet the desired performance objectives. While there are almost always small adjustments that can be made to the model to try to get it working "better", if there are lots of adjustments being made it might be a sign that the design itself needs to be evaluated.

In this case, we're going to go with the easy option for now (not clearing the **imaginal** buffer), but if we have any more problems we will look at our design before adjusting the model further. There are multiple things we could do to keep the chunk in the **imaginal** buffer for this model since we are not really constrained by other productions which use the buffer or the chunk that's created there. What seems like the easiest option here is to just change the respond-next production so that it keeps the chunk in the buffer instead of allowing strict harvesting to clear it. To do that, we need to perform a modification action on the RHS of respond-next. There isn't a meaningful modification that we need to make, but that's alright because a production is allowed to make what's called an empty modification for exactly this purpose. To do that one just adds an = buffer action on the RHS without specifying any slots and values to modify. Here is what the updated respond-next production looks like:

```
(p respond-next
   =imaginal>
     isa         task
     letter      =letter
   =visual>
     isa         visual-object
     value       "next"
   ?manual>
     state       free
   ==>
   =imaginal>
   +retrieval>
     isa         letter
     previous    =letter
   +manual>
     cmd         press-key
     key         =letter
)
```

We should make a similar change to the respond-previous production while we are modifying the model since we will likely encounter the same issue there.

If we didn't want to make that change or if there were lots of productions or instances where this was an issue in the model we could alternatively turn off the strict harvesting mechanism for the

imaginal buffer. That can be done using the :do-not-harvest parameter in the system. In this simple mode that would not cause any issues, but for larger models one would have to consider that carefully because it may affect other productions which also use the buffer and then require the model to explicitly clear that buffer in some places.

Now, again, we will save that change and reload the model.

Model version 5

This is what the trace looks like now when we run it:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.225   PROCEDURAL     CLEAR-BUFFER MANUAL
2.225   MOTOR          PRESS-KEY KEY n
2.225   DECLARATIVE    start-retrieval
2.225   DECLARATIVE    RETRIEVED-CHUNK O
2.225   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL O
2.225   PROCEDURAL     CONFLICT-RESOLUTION
2.475   PROCEDURAL     CONFLICT-RESOLUTION
2.525   PROCEDURAL     CONFLICT-RESOLUTION
2.590   PROCEDURAL     CONFLICT-RESOLUTION
2.625   PROCEDURAL     CONFLICT-RESOLUTION
2.775   PROCEDURAL     CONFLICT-RESOLUTION
2.825   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.825   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.825   PROCEDURAL     CLEAR-BUFFER MANUAL
2.825   MOTOR          PRESS-KEY KEY o
2.825   PROCEDURAL     CONFLICT-RESOLUTION
2.975   PROCEDURAL     CONFLICT-RESOLUTION
3.025   PROCEDURAL     CONFLICT-RESOLUTION
3.125   PROCEDURAL     CONFLICT-RESOLUTION
3.275   PROCEDURAL     CONFLICT-RESOLUTION
3.275   ------         Stopped because no events left to process
```

Here we see that the model has performed the task correctly and that it performed the steps which we expected. Before moving on and trying the "previous" trials however we may want to perform some more tests so that we are confident that it works well for the "next" prompt. In particular, this model has the :seed parameter set to keep things consistent while debugging. We should try removing that from the model and running it a couple of times so that we can see if it is able to perform the task for letters other than "N" and when the prompt is displayed at times other than 2.090. Instead of actually removing that line from the model however it is probably best to just "comment it out" so that we can easily restore it for testing if things go wrong and when we start testing the "previous" prompt. In Lisp syntax, the semi-colon character is used to create comments and everything on a line after the semi-colon will be ignored. Thus, we should put a semi-colon at the start of the line where the seed is set:

```
;    (sgp :seed (101 1))
```

In addition, we may also want to turn the trace-detail down to low since we expect to just be checking a correctly function model at this point and don't need all the extra details. After making those changes, save the model and reload it. Running it a few times seems to indicate that it is still able to perform the task correctly and as expected with varying letters and different prompting times. So, now we can move on to test trials with the "previous" prompt.

Testing "previous" trial

Before starting to test the "previous" trials it is probably best to uncomment the :seed parameter setting by removing the semi-colon and set the trace-detail level back to medium. After making those changes, saving and then loading the model we can run a trial specifying previous:

```
? (pm-issues-task "previous")
```

```
>>> pm_issues.task('previous')
```

```
0.000   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION          visicon-update
0.000   PROCEDURAL      CONFLICT-RESOLUTION
0.050   PROCEDURAL      PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL      CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL      CLEAR-BUFFER VISUAL
0.050   PROCEDURAL      CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL      CONFLICT-RESOLUTION
0.135   VISION          Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION          SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL      CONFLICT-RESOLUTION
0.250   IMAGINAL        SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL      CONFLICT-RESOLUTION
0.300   PROCEDURAL      PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL      CLEAR-BUFFER VISUAL
0.300   PROCEDURAL      CONFLICT-RESOLUTION
2.090   NONE            pm-issue-display (vision exp-window Simple task) previous
2.090   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION          visicon-update
```

```
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         No visual-object found
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.590   PROCEDURAL     CONFLICT-RESOLUTION
2.590   ------         Stopped because no events left to process
```

The model failed to do the task so now we need to investigate why. The next production which we expect to fire is respond-previous and we can request the whynot information about it now since the model has stopped when we expect it to be selected:

```
Production RESPOND-PREVIOUS does NOT match.
(P RESPOND-PREVIOUS
   =IMAGINAL>
       LETTER =LETTER
   =VISUAL>
       VALUE "previous"
       TEXT T
   ?MANUAL>
       STATE FREE
 ==>
   =IMAGINAL>
   +RETRIEVAL>
       NEXT =LETTER
   +MANUAL>
       CMD PRESS-KEY
       KEY =LETTER
)
It fails because:
The VISUAL buffer is empty.
```

It's failing to match because the **visual** buffer is empty. So, now the question becomes why is the **visual** buffer empty since it worked for the prompt "next"? Before looking at the model trace we might want to make sure that there isn't a bug in the code which presented the experiment to the model. To do that we can look at the experiment window which was presented and make sure it has the word previous displayed in it, which it does. Then the next thing to check would be the model's visicon to make sure that it has properly updated with the current information. That can be done using the print-visicon command or with the "Visicon" button in the Environment. Here is what that displays:

```
Name              Att  Loc             Text  Kind  Color  Width  Value       Height  Size
----------------  ---  --------------  ----  ----  -----  -----  ----------  ------  -----
VISUAL-LOCATION1  NEW  (454 456 1080)  T     TEXT  BLACK  56     "previous"  10      1.579
```

So, indeed the vision module has processed that the word previous is visible on the screen and thus the experiment code appears to be working correctly and the problem must be with the model. Doing a simple check like that before proceeding can be very helpful to make sure you know what is happening before trying to fix a problem in the model which might not even exist.

One more thing that we'll do before trying to change the model is compare what happens in the vision module after the prompt appears on a "next" trial to what happens on a "previous" trial. Here is the trace for the correct "next" trial:

```
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
```

and here is the trace from the same segment of the "previous" trial:

```
2.090   NONE           pm-issue-display (vision exp-window Simple task) previous
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         No visual-object found
2.175   PROCEDURAL     CONFLICT-RESOLUTION
```

In the "next" trial we see a chunk placed into the **visual** buffer, but in the "previous" trial the module reports that there is no visual-object found. The first question to ask would seem to be why is there any visual activity at all, since there isn't a request made by a production? The answer to that is that in addition to stuffing a chunk in the **visual-location** buffer when there is a change to the visual scene the vision module will automatically re-encode the location where it is currently attending. So, that is what causes the encoding which completes at time 2.175. That hadn't actually been taken into account in our original design, but by chance we got lucky with the "next" prompt. Before deciding what to do about it we should first figure out why it works for "next" and see how that differs from "previous". To investigate that we should look at the visicon for the three different items which get displayed: the initial letter, "next", and "previous". To do that we'll use the Stepper to pause the model at the start of the task to see the letter information and then advance to the time when the screen changes to see what things look like there.

Here are the visicon entries for those items:

```
Name              Att  Loc             Text  Kind  Color  Width  Value  Height  Size
----------------  ---  --------------  ----  ----  -----  -----  -----  ------  ----------
VISUAL-LOCATION0  NEW  (435 456 1080)  T     TEXT  BLACK  7      "n"    10      0.19999999

Name              Att  Loc             Text  Kind  Color  Width  Value   Height  Size
----------------  ---  --------------  ----  ----  -----  -----  ------  ------  ----------
VISUAL-LOCATION1  NEW  (440 456 1080)  T     TEXT  BLACK  28     "next"  10      0.78999996

Name              Att  Loc             Text  Kind  Color  Width  Value       Height  Size
----------------  ---  --------------  ----  ----  -----  -----  ----------  ------  -----
VISUAL-LOCATION1  NEW  (454 456 1080)  T     TEXT  BLACK  56     "previous"  10      1.579
```

In addition to what the model sees, we can also look at the experiment code which generate those displays. Here is how the letter and prompts are displayed:

Lisp

```
(add-text-to-exp-window window letter :x 130 :y 150)

(add-text-to-exp-window window prompt :x 125 :y 150)
```

Python

```
    actr.add_text_to_exp_window(window,letter,x=130,y=150)

    actr.add_text_to_exp_window(window,prompt,x=125,y=150)
```

Notice how each visicon entry is at a different location and those locations do not exactly match where the text was displayed. That's because the locations in the visicon are determined by the center of the item (which is meaningful to the model) and the location of the window in which they are displayed (which defaults to position 300,300), but the display functions only specify the upper left corner of the item for creating the display. That still doesn't directly answer why "next" gets attended but "previous" does not. The missing piece to the puzzle is what it means for the model to re-encode the currently attended location. The re-encoding action which the vision module automatically performs when there is a scene change allows for some movement of items in the visual scene. As long as there is some object "close" to where it is attending that new object will be attended automatically. What it means to be close is controlled by a parameter in the vision module. We won't discuss the details here, but they can be found in the reference manual. The important thing for our current purposes is to notice that "next" is closer to the letter than "previous" is and thus apparently "next" is close enough to be re-encoded but "previous" is not.

After working through that, now the question becomes what do we do about it? Looking back at the design of our model, we see that we hadn't actually built in a way for the model to attend to the prompt. That's a flaw in the design of the model which we should address so that it can correctly perform the task.

Before doing so however, we will consider some other possible fixes for the model. Since it works correctly for "next" we could modify the code that presents the experiment so that it also displays "previous" close enough to the letter that it gets attended automatically. Alternatively, we could adjust the parameter that controls how close something needs to be to be automatically re-attended so that both prompts work. Either of those should be sufficient to have the model complete the task, but are they good things to do? If one believes that that aspect of the task is not relevant to the data being collected then perhaps one could consider those to be reasonable changes, but it does then mean that there is an assumption in the design of the model – it can only perform the task if the prompts are displayed in the "same" location as the letter (where same means within the re-encoding range of the vision module). If one is trying to build a model which can perform the more general task which we have described here (there is no constraint on where the prompts are displayed in the task description) then such a model is not sufficient to do that task. In general, engineering the experiment or support code so that the model performs "better" or just adjusting parameters without a good reason is not a good approach to modeling. The model should be robust enough that it can perform the task regardless of particular details in the code with which it is interacting and it should not be dependent on assumptions which are not true of the task it is supposed to be performing. Similarly, it is generally better to have a model which works well with the default parameters for aspects of the model which are not relevant to the task than it is to have a model which only works well because of specific parameter settings which are changing things that aren't directly relevant to the current task. Thus, we will not attempt either of those fixes for this model.

Reconsidering the model design

Now we will consider how we need to change the design for the model. Here is the design which we had originally planned:

- When it detects a letter on the screen attend it and then store it in the **imaginal** buffer
- When it sees next or previous press the current key and retrieve the appropriate letter chunk from declarative memory
- Once a chunk is retrieved press that key

There are many ways to go about changing that design, but since it was almost working we will first consider the simple addition of the step which we seem to be missing. Thus, we will add an additional step to explicitly attend to the prompt when we see the screen change:

- When it detects a letter on the screen attend it and then store it in the **imaginal** buffer
- When it detects the screen change attend to the location of the new item
- When it sees next or previous press the current key and retrieve the appropriate letter chunk from declarative memory
- Once a chunk is retrieved press that key

That change seems to be sufficient to address the problem we had and does not require changing any of the other assumptions we have in the design. Thus, we should be able to keep the model we have and just add productions as necessary to implement that new step. Other changes to the design would likely require more changes to the model or adjustments of our design assumptions so we will not consider those for now.

Adding the new step

To implement the new step we need another production which should look a lot like the production needed for the first step, except that it will not need to initialize the **imaginal** buffer. We will call that production find-prompt and here is what it looks like which is almost identical to find-letter except it tests for the **imaginal** buffer being full instead of empty since it will have the chunk we created in find-letter in it and it doesn't make a request to the **imaginal** buffer on the RHS:

```
(p find-prompt
   =visual-location>
   ?visual>
     state       free
   ?imaginal>
     buffer      full
 ==>
   +visual>
     cmd         move-attention
     screen-pos  =visual-location
)
```

Again, we are relying on buffer stuffing to put a chunk into the **visual-location** buffer automatically because that is the way that the model can detect a change to the visual scene, and if we look at the traces above we see that indeed a chunk is stuffed into the buffer at that time:

```
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
```

Note that we could also have tested for a chunk in the **imaginal** buffer by simply having an =imaginal> test like we do for the visual-location buffer, but the query is used here for consistency with the find-letter production.

We need to save that change to the model and load it again.

Model version 6

Here is the trace for running the updated model on a trial with previous:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) previous
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         No visual-object found
```

```
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED FIND-PROMPT
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CONFLICT-RESOLUTION
2.310   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.310   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
2.310   PROCEDURAL     CONFLICT-RESOLUTION
2.360   PROCEDURAL     PRODUCTION-FIRED RESPOND-PREVIOUS
2.360   PROCEDURAL     CLEAR-BUFFER VISUAL
2.360   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.360   PROCEDURAL     CLEAR-BUFFER MANUAL
2.360   MOTOR          PRESS-KEY KEY n
2.360   DECLARATIVE    start-retrieval
2.360   DECLARATIVE    RETRIEVED-CHUNK M
2.360   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL M
2.360   PROCEDURAL     CONFLICT-RESOLUTION
2.610   PROCEDURAL     CONFLICT-RESOLUTION
2.660   PROCEDURAL     CONFLICT-RESOLUTION
2.760   PROCEDURAL     CONFLICT-RESOLUTION
2.910   PROCEDURAL     CONFLICT-RESOLUTION
2.960   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.960   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.960   PROCEDURAL     CLEAR-BUFFER MANUAL
2.960   MOTOR          PRESS-KEY KEY m
2.960   PROCEDURAL     CONFLICT-RESOLUTION
3.010   PROCEDURAL     CONFLICT-RESOLUTION
3.060   PROCEDURAL     CONFLICT-RESOLUTION
3.160   PROCEDURAL     CONFLICT-RESOLUTION
3.310   PROCEDURAL     CONFLICT-RESOLUTION
3.310   ------         Stopped because no events left to process
```

The model successfully completed the task for “previous” and performed the steps which we expected it to. Now we should test it on a trial for “next” to make sure that it can still do those trials as well:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.090   NONE           pm-issue-display (vision exp-window Simple task) next
2.090   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.090   VISION         visicon-update
2.090   PROCEDURAL     CONFLICT-RESOLUTION
2.175   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.175   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.175   PROCEDURAL     CONFLICT-RESOLUTION
2.225   PROCEDURAL     PRODUCTION-FIRED FIND-PROMPT
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.225   PROCEDURAL     CLEAR-BUFFER VISUAL
2.225   PROCEDURAL     CONFLICT-RESOLUTION
```

```
2.310   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.310   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.310   PROCEDURAL     CONFLICT-RESOLUTION
2.360   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.360   PROCEDURAL     CLEAR-BUFFER VISUAL
2.360   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.360   PROCEDURAL     CLEAR-BUFFER MANUAL
2.360   MOTOR          PRESS-KEY KEY n
2.360   DECLARATIVE    start-retrieval
2.360   DECLARATIVE    RETRIEVED-CHUNK O
2.360   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL O
2.360   PROCEDURAL     CONFLICT-RESOLUTION
2.610   PROCEDURAL     CONFLICT-RESOLUTION
2.660   PROCEDURAL     CONFLICT-RESOLUTION
2.760   PROCEDURAL     CONFLICT-RESOLUTION
2.910   PROCEDURAL     CONFLICT-RESOLUTION
2.960   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.960   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.960   PROCEDURAL     CLEAR-BUFFER MANUAL
2.960   MOTOR          PRESS-KEY KEY o
2.960   PROCEDURAL     CONFLICT-RESOLUTION
3.110   PROCEDURAL     CONFLICT-RESOLUTION
3.160   PROCEDURAL     CONFLICT-RESOLUTION
3.260   PROCEDURAL     CONFLICT-RESOLUTION
3.410   PROCEDURAL     CONFLICT-RESOLUTION
3.410   ------         Stopped because no events left to process
```

Here again it did the task correctly and fired the productions which we expected. At this point one might consider the model done, but we should remove the seed parameter setting (or comment it out) and perform some more tests to make sure that the model doesn’t have a dependence on that particular parameter setting.

Further tests of the working model

For the trials with “previous” everything still seems to work after running a few trials, but for next occasionally we get a trial where it does not complete the task correctly and looks something like this:

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.300   NONE           pm-issue-display (vision exp-window Simple task) next
2.300   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.300   VISION         visicon-update
2.300   PROCEDURAL     CONFLICT-RESOLUTION
2.385   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
```

```
2.385   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.385   PROCEDURAL     CONFLICT-RESOLUTION
2.435   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.435   PROCEDURAL     CLEAR-BUFFER VISUAL
2.435   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.435   PROCEDURAL     CLEAR-BUFFER MANUAL
2.435   MOTOR          PRESS-KEY KEY d
2.435   DECLARATIVE    start-retrieval
2.435   DECLARATIVE    RETRIEVED-CHUNK E
2.435   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL E
2.435   PROCEDURAL     CONFLICT-RESOLUTION
2.485   PROCEDURAL     PRODUCTION-FIRED FIND-PROMPT
2.485   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.485   PROCEDURAL     CLEAR-BUFFER VISUAL
2.485   PROCEDURAL     CONFLICT-RESOLUTION
2.570   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.570   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.570   PROCEDURAL     CONFLICT-RESOLUTION
2.585   PROCEDURAL     CONFLICT-RESOLUTION
2.635   PROCEDURAL     CONFLICT-RESOLUTION
2.645   PROCEDURAL     CONFLICT-RESOLUTION
2.735   PROCEDURAL     CONFLICT-RESOLUTION
2.785   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.785   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.785   PROCEDURAL     CLEAR-BUFFER MANUAL
2.785   MOTOR          PRESS-KEY KEY e
2.785   PROCEDURAL     CONFLICT-RESOLUTION
3.035   PROCEDURAL     CONFLICT-RESOLUTION
3.085   PROCEDURAL     CONFLICT-RESOLUTION
3.185   PROCEDURAL     CONFLICT-RESOLUTION
3.335   PROCEDURAL     CONFLICT-RESOLUTION
3.385   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
3.385   PROCEDURAL     CLEAR-BUFFER VISUAL
3.385   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
3.385   PROCEDURAL     CLEAR-BUFFER MANUAL
3.385   MOTOR          PRESS-KEY KEY d
3.385   DECLARATIVE    start-retrieval
3.385   DECLARATIVE    RETRIEVED-CHUNK E
3.385   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL E
3.385   PROCEDURAL     CONFLICT-RESOLUTION
3.535   PROCEDURAL     CONFLICT-RESOLUTION
3.585   PROCEDURAL     CONFLICT-RESOLUTION
3.595   PROCEDURAL     CONFLICT-RESOLUTION
3.685   PROCEDURAL     CONFLICT-RESOLUTION
3.735   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
3.735   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
3.735   PROCEDURAL     CLEAR-BUFFER MANUAL
3.735   MOTOR          PRESS-KEY KEY e
3.735   PROCEDURAL     CONFLICT-RESOLUTION
3.985   PROCEDURAL     CONFLICT-RESOLUTION
4.035   PROCEDURAL     CONFLICT-RESOLUTION
4.135   PROCEDURAL     CONFLICT-RESOLUTION
4.285   PROCEDURAL     CONFLICT-RESOLUTION
4.285   ------         Stopped because no events left to process
```

Looking at the trace we see that the respond-next production fired when we expected our find-prompt production to fire, and then find-prompt fired after that which caused respond-next to fire again after respond-final. That caused the model to press each of the keys twice and thus failing the task.

While it may be possible with this simple model to determine why this occurred from the trace and looking at the productions, in other cases one may need to investigate that further with the

Stepper and the inspection tools. Because it only happens on some of the trials that can become a difficult task since one may have to go through things several times before seeing the problem again. Before discussing ways to fix this model we will cover a couple of things that can be done to help with investigating randomly occurring problems like this.

Techniques for working with randomly occurring problems

The first thing that one can do is have additional information displayed in the trace. That might be enough to help fix things without having to use the Stepper and other tools because then one can just run the model until a problem trial occurs and inspect the additional information in the trace. Some modules provide extra trace information which can be turned on to show more details about what they are doing. In this case, we could take advantage of two traces which the procedural module provides. They are called the "conflict set trace" and the "conflict resolution trace" and can be enabled by setting the :cst and :crt parameters respectively in the model. If those parameters are set to t then details about which productions match are shown in the trace for each conflict resolution action. We will not describe those traces further here, but you can try them out with this model to see the type of information they provide.

Another thing that can be done is to use the :seed parameter to force the model to repeat a particular sequence of actions. We've seen that used often in the tutorial to provide consistent examples, but the problem is how do you find a seed for a "bad" trial so that you can replay it for further inspection? One approach is to just run the model repeatedly letting it pick its own seed and have it display that initial seed at the start of the task. Then, when you find a trial that doesn't work correctly you can take the seed value that was displayed and set it in the model so that you can repeat that broken trial to inspect it further. The easy way to do that is to just add a call to sgp specifying the :seed parameter as the first command in the model definition like this:

```
(sgp :seed)
```

If a value isn't provided for a parameter to the sgp command it prints out the current value of that parameter along with the default value and some documentation. Thus, if we add that to the top of our current model and turn the trace off so that things run faster we should be able to quickly find a seed value which will allow us to repeat a broken trial for further inspection. For example, here is a sample of what that might look like for the current task running from the ACT-R prompt (your seed values are likely to differ from those shown below since the starting seed is pseudo-randomly determined if one is not provided):

```
? (pm-issues-task "next")
:SEED (43116477826 0) (default NO-DEFAULT) : Current seed of the random number generator
("next" T)

? (pm-issues-task "next")
:SEED (43116477826 38) (default NO-DEFAULT) : Current seed of the random number generator
("next" NIL)
```

In this case we found that a seed of (43116477826 38) leads to the model failing the task. Now we can set that seed in the model definition like this:

```
(sgp :seed (43116477826 38))
```

and the model will always perform that same bad trial which we can then investigate further.

Using the seed parameter like that can be very convenient, not only for debugging but for demonstration purposes to find a situation that one wants to repeat (as is done for the tutorial models). However, there is one requirement of the model and experiment code to be able to use it that way. It will only work if all of the randomness in both the model and the experiment depends on the ACT-R provided randomness functions. If the task or model uses some other source of random numbers then setting the ACT-R seed parameter will not guarantee that the same sequence of actions will occur and one will also have to control that other random source as well to guarantee a repeatable trial. All of the tasks in the tutorial satisfy the constraint of only using the ACT-R randomness functions.

The broken “next” trial

Now that we have a way to recreate a non-working trial we can investigate it further. The first thing we want to do is turn the trace back on and run it to look at what happens. Here is the trace we get with the seed found above (43116477826 38):

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL
0.300   PROCEDURAL     CONFLICT-RESOLUTION
2.300   NONE           pm-issue-display (vision exp-window Simple task) next
2.300   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.300   VISION         visicon-update
2.300   PROCEDURAL     CONFLICT-RESOLUTION
2.385   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
2.385   VISION         SET-BUFFER-CHUNK VISUAL TEXT1 NIL
2.385   PROCEDURAL     CONFLICT-RESOLUTION
2.435   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
2.435   PROCEDURAL     CLEAR-BUFFER VISUAL
2.435   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.435   PROCEDURAL     CLEAR-BUFFER MANUAL
2.435   MOTOR          PRESS-KEY KEY d
2.435   DECLARATIVE    start-retrieval
2.435   DECLARATIVE    RETRIEVED-CHUNK E
2.435   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL E
2.435   PROCEDURAL     CONFLICT-RESOLUTION
2.485   PROCEDURAL     PRODUCTION-FIRED FIND-PROMPT
2.485   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
2.485   PROCEDURAL     CLEAR-BUFFER VISUAL
```

```
2.485   PROCEDURAL     CONFLICT-RESOLUTION
2.570   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
2.570   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
2.570   PROCEDURAL     CONFLICT-RESOLUTION
2.585   PROCEDURAL     CONFLICT-RESOLUTION
2.635   PROCEDURAL     CONFLICT-RESOLUTION
2.645   PROCEDURAL     CONFLICT-RESOLUTION
2.735   PROCEDURAL     CONFLICT-RESOLUTION
2.785   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
2.785   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
2.785   PROCEDURAL     CLEAR-BUFFER MANUAL
2.785   MOTOR          PRESS-KEY KEY e
2.785   PROCEDURAL     CONFLICT-RESOLUTION
3.035   PROCEDURAL     CONFLICT-RESOLUTION
3.085   PROCEDURAL     CONFLICT-RESOLUTION
3.185   PROCEDURAL     CONFLICT-RESOLUTION
3.335   PROCEDURAL     CONFLICT-RESOLUTION
3.385   PROCEDURAL     PRODUCTION-FIRED RESPOND-NEXT
3.385   PROCEDURAL     CLEAR-BUFFER VISUAL
3.385   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
3.385   PROCEDURAL     CLEAR-BUFFER MANUAL
3.385   MOTOR          PRESS-KEY KEY d
3.385   DECLARATIVE    start-retrieval
3.385   DECLARATIVE    RETRIEVED-CHUNK E
3.385   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL E
3.385   PROCEDURAL     CONFLICT-RESOLUTION
3.535   PROCEDURAL     CONFLICT-RESOLUTION
3.585   PROCEDURAL     CONFLICT-RESOLUTION
3.595   PROCEDURAL     CONFLICT-RESOLUTION
3.685   PROCEDURAL     CONFLICT-RESOLUTION
3.735   PROCEDURAL     PRODUCTION-FIRED RESPOND-FINAL
3.735   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
3.735   PROCEDURAL     CLEAR-BUFFER MANUAL
3.735   MOTOR          PRESS-KEY KEY e
3.735   PROCEDURAL     CONFLICT-RESOLUTION
3.985   PROCEDURAL     CONFLICT-RESOLUTION
4.035   PROCEDURAL     CONFLICT-RESOLUTION
4.135   PROCEDURAL     CONFLICT-RESOLUTION
4.285   PROCEDURAL     CONFLICT-RESOLUTION
4.285   ------         Stopped because no events left to process
```

The first problem in the trace shows up at time 2.435 when respond-next fires but we expect find-prompt to fire. However, stepping to that point will be too late because the real issue we want to investigate is during the conflict resolution action which results in respond-next being selected – we want to see why find-prompt isn't selected at that time. To see the production selection event in the trace (and thus be able to step to it) we will have to set the trace-detail parameter to high. If we make that change, save and then load the model we can now run it and use the Stepper to get to the point where the problem occurs, which is time 2.385, when the conflict resolution action selects respond-next instead of find-prompt.

Stepping to that production selection event we see that in fact both respond-next and find-prompt match at that point in time (the Stepper shows both of them after we step past the production-selected action). So, now the question is why is one chosen over the other? The answer to that has to do with how the procedural module selects among productions when more than one matches. The first determination is by utility values; the production with the higher utility value will be the one chosen. In this case both productions have the same utility which is the default of 0 since we have not changed them. When productions have the same utility how the procedural module decides is determined by the setting of the :er (enable randomness) parameter. If the parameter is set to nil (which is the default value) then the module will use an unspecified but deterministic mechanism to choose one of the two productions. That will result in a specific model always having the same production chosen when that same tie situation occurs, but it does not guarantee that same choice will be made for any other model or even for that same model if it is changed in any way. While that is deterministic and can be useful when starting to work on a model it is not generally a good thing to rely on for a robust model. Instead the recommendation is to set the :er parameter to t which means that whenever there is a tie for the top utility value the model will randomly pick which production to fire (of course as was discussed above even the random processes of the model can be made deterministic by setting the seed parameter). In this model the :er parameter is set to t, thus that is why sometimes it works and sometimes it does not.

Options for how to fix the problem

Now that we know what's wrong with the model we need to make sure that find-prompt always fires instead of respond-next in that situation. There are a few options available, including yet another redesign of our task. We will look at some of the options available before making a choice or determining whether or not to amend the design again.

The first thing we could do is turn off the :er parameter and see which one it favors. If find-prompt is the winner then that would solve the problem. However, that's not really a good choice since it would only work because of an arbitrary mechanism in the procedural module which we cannot control and if we make any other changes to the model it may stop working.

As was done in the sperling model for the unit 3 example we could set explicit utilities on the productions involved. That way we could guarantee that find-prompt is always chosen over respond-next. This would be better than the previous option since we would be in control of how the choice was made. In this situation that seems like a reasonable solution, but when we get to later units and are working with models that are able to learn the utilities we will find that setting fixed initial values to control the operation of the model may not work as well.

We could try to find some state that differs at that time which would allow us to add additional conditions to one or both of those productions to prevent them from both matching at that point. Both productions already have tests using the **imaginal** and **visual** buffers, so those are not likely to provide any differentiation. However, read-prompt requires a chunk in the **visual-location** buffer and respond-next does not. So, we could make that explicit by adding a test that the **visual-location** buffer was empty to respond-next and that should prevent them from both matching at the same time. If we choose to do that we would also want to make that same change to respond-previous to be consistent.

The next alternative is to adjust the earlier productions in the model so that it has a different state than it does now at that critical time when the screen changes so that both productions no longer match. Here there seem to be a variety of options available. One would be to add a **goal** buffer chunk with an explicit state which could be tested, but we've been trying to avoid that as part of the design for the model. Instead of using the **goal** buffer, since we already have a chunk in the **imaginal** buffer, we could add some explicit state marker to that chunk or perhaps set the contents of that chunk's existing slot in such a way as to implicitly indicate the state. That however seems to still go against the design we have for the model and also goes against the distinction between the **goal** and **imaginal** buffers in ACT-R i.e. that **goal** should be used for state information and **imaginal** for problem representation. Another option would be to change the state by changing the actions which the model performs. In particular, we can stop the automatic re-encoding from happening by having the model stop attending to the location of the letter once it has encoded it. That would prevent respond-next and respond-previous from being able to match until after find-prompt fires because there wouldn't be a chunk in the **visual** buffer. In fact if we had done that earlier it may have avoided some of the other problems we encountered.

Now we have three options which seem reasonable: set explicit utilities for the productions, add an additional condition to the respond productions, or have the model stop attending the letter. So, how do we decide which one to use? An important thing to consider in making that decision is why are we creating the model? If we had data for this task that we were trying to fit then that might help us to make the decision based on how the model's response times might differ among the options. Something else to consider would be cognitive plausibility – are we trying to create a model which we think performs the task like a person? If so, then we would want to consider which of the options seems to best correspond to what we think a person does while performing the task. If one has other objectives for building the model, then comparing the options with respect to those objectives would be the thing to do. Essentially, there is not a single "right" model for a task. What is important is that the model one builds satisfies the purposes for which it was written, and that usually involves understanding the details about how the model works and being able to justify the choices made.

Since the objective of this model is demonstrating debugging and modeling techniques related to perceptual and motor module issues, any of those options seems like a justifiable choice. The last one of the three however seems like it would be the best since it uses another perceptual action which may provide additional areas to investigate.

Adding the new action

To make the model stop attending we need to make an explicit request to the vision module indicating that with the "cmd clear" request to the **visual** buffer. In this task the model does not need to keep attending the letter after it has harvested the information from the **visual** buffer and that happens in the encode-letter production. Thus, that is where we want to make the request to stop attending. In addition to making the request we should also add a test to the LHS of the production to make sure the module is free to avoid the possibility of jamming when it gets that request. Here is the updated production with those changes:

```
(p encode-letter
   =imaginal>
     isa         task
     letter      nil
   =visual>
     isa         visual-object
```

```
      value         =letter
    ?visual>
      state         free
  ==>
    +visual>
      cmd           clear
    =imaginal>
      letter        =letter
)
```

With that change and the trace-detail set back to medium here is the trace we get when running it with the seed we had set for the incorrect trial:

```
0.000   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION          visicon-update
0.000   PROCEDURAL      CONFLICT-RESOLUTION
0.050   PROCEDURAL      PRODUCTION-FIRED FIND-LETTER
0.050   PROCEDURAL      CLEAR-BUFFER VISUAL-LOCATION
0.050   PROCEDURAL      CLEAR-BUFFER VISUAL
0.050   PROCEDURAL      CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL      CONFLICT-RESOLUTION
0.135   VISION          Encoding-complete VISUAL-LOCATION0-0 NIL
0.135   VISION          SET-BUFFER-CHUNK VISUAL TEXT0
0.135   PROCEDURAL      CONFLICT-RESOLUTION
0.250   IMAGINAL        SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL      CONFLICT-RESOLUTION
0.300   PROCEDURAL      PRODUCTION-FIRED ENCODE-LETTER
0.300   PROCEDURAL      CLEAR-BUFFER VISUAL
0.300   VISION          CLEAR
0.300   PROCEDURAL      CONFLICT-RESOLUTION
0.350   PROCEDURAL      CONFLICT-RESOLUTION
2.300   NONE            pm-issue-display (vision exp-window Simple task) next
2.300   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
2.300   VISION          visicon-update
2.300   PROCEDURAL      CONFLICT-RESOLUTION
2.350   PROCEDURAL      PRODUCTION-FIRED FIND-PROMPT
2.350   PROCEDURAL      CLEAR-BUFFER VISUAL-LOCATION
2.350   PROCEDURAL      CLEAR-BUFFER VISUAL
2.350   PROCEDURAL      CONFLICT-RESOLUTION
2.435   VISION          Encoding-complete VISUAL-LOCATION1-0 NIL
2.435   VISION          SET-BUFFER-CHUNK VISUAL TEXT1
2.435   PROCEDURAL      CONFLICT-RESOLUTION
2.485   PROCEDURAL      PRODUCTION-FIRED RESPOND-NEXT
2.485   PROCEDURAL      CLEAR-BUFFER VISUAL
2.485   PROCEDURAL      CLEAR-BUFFER RETRIEVAL
2.485   PROCEDURAL      CLEAR-BUFFER MANUAL
2.485   MOTOR           PRESS-KEY KEY d
2.485   DECLARATIVE     start-retrieval
2.485   DECLARATIVE     RETRIEVED-CHUNK E
2.485   DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL E
2.485   PROCEDURAL      CONFLICT-RESOLUTION
2.635   PROCEDURAL      CONFLICT-RESOLUTION
2.685   PROCEDURAL      CONFLICT-RESOLUTION
2.695   PROCEDURAL      CONFLICT-RESOLUTION
2.785   PROCEDURAL      CONFLICT-RESOLUTION
2.835   PROCEDURAL      PRODUCTION-FIRED RESPOND-FINAL
2.835   PROCEDURAL      CLEAR-BUFFER RETRIEVAL
2.835   PROCEDURAL      CLEAR-BUFFER MANUAL
2.835   MOTOR           PRESS-KEY KEY e
2.835   PROCEDURAL      CONFLICT-RESOLUTION
```

```
3.085   PROCEDURAL     CONFLICT-RESOLUTION
3.135   PROCEDURAL     CONFLICT-RESOLUTION
3.235   PROCEDURAL     CONFLICT-RESOLUTION
3.385   PROCEDURAL     CONFLICT-RESOLUTION
3.385   ------         Stopped because no events left to process
```

The model successfully completed the task. So, now it looks like the model is working correctly, but we should remove the seed parameter setting and run a few more tests to make sure. Running some additional tests seems to show that the model is now able to perform the task as expected. Given some of the issues that we encountered however, there is some additional testing that might be worthwhile to perform. Because we had issues with where the letter and prompts were displayed it might be a good idea to change the code which presents those items to make sure that the model can perform the task regardless of where the items are on the screen. We will not work through those tests here, but you should try that out on your own to see what happens. In addition to that you may also want to consider implementing some of the proposed, but not chosen, fixes that were described as we encountered some of the problems to see how those solutions differ in performance, if at all, from the options that were chosen. Finally, you might also want to consider alternative designs for performing the task and for the initial letter representations.

Additional Environment Tools

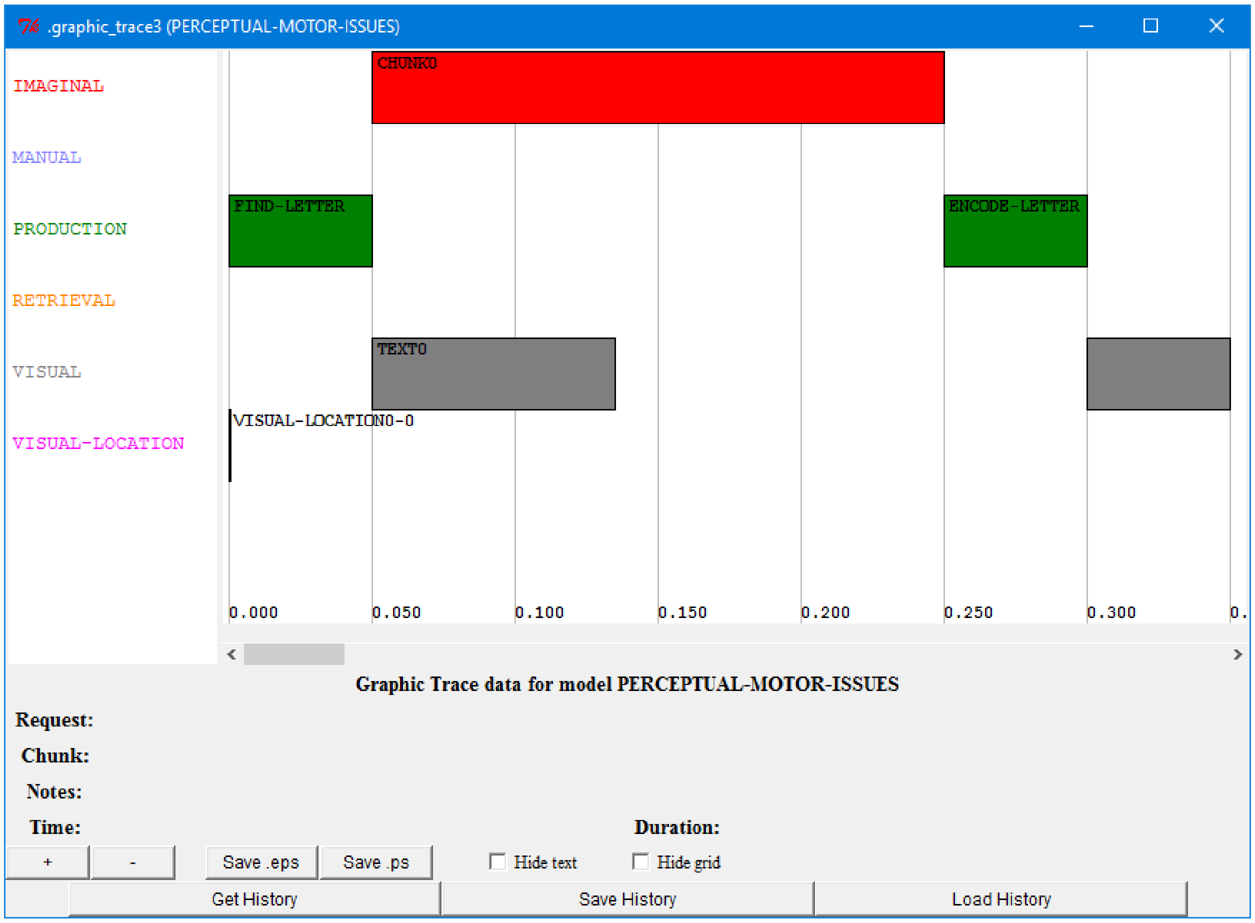
To debug this model we have relied on reading the trace, inspecting the buffer contents and status, and using the Stepper. Those are important skills to learn because they will be useful for almost all ACT-R modeling tasks. However, there are some other tools available in the Environment which we could also have used while working with this model. The two "Recordable Data" sections of the Control Panel can often be useful when working with larger models or models which run for longer periods of time. We will briefly describe how to set up and use some of those tools here and provide some suggestions for how they may be useful. For more details on using those tools you should consult the Environment's manual which is included in the docs directory of the ACT-R distribution.

General Usage

All of the tools which record data from the model need to be enabled before the run begins, and the easiest way to enable them is to just open it first. Once it is open the appropriate data will be recorded as the model runs. For some of the tools there are also parameters which one can set in the model to record the data, but we will not discuss any of those parameters here. For the tools in the "Buffer Based Recordable Data" section in addition to enabling the tool one has to select the specific buffers for which that data is to be recorded. The tool to do that will also open automatically when any of the tools which need it are opened. By default all of the buffers which are used in the productions of the model will be selected, but you can add others or remove ones you don't need by checking or unchecking the box in the selection window. All of the buffer based tools will record data for the same set of selected buffers.

Graphic Traces

Instead of reading through the text based trace one can instead use a graphic representation of the model's activities. The "Graphic Trace" button opens a viewer which will show the activities the model performed for each recorded buffer. The option menu button to the right of the "Graphic Trace" button lets one choose whether the information is displayed horizontally or vertically. Once you have run the model you can get the trace by pressing the "Get History" button at the bottom of the window (which is true for all of the recordable data tools). Here is what that will look like using the horizontal view after running the final version of the model:

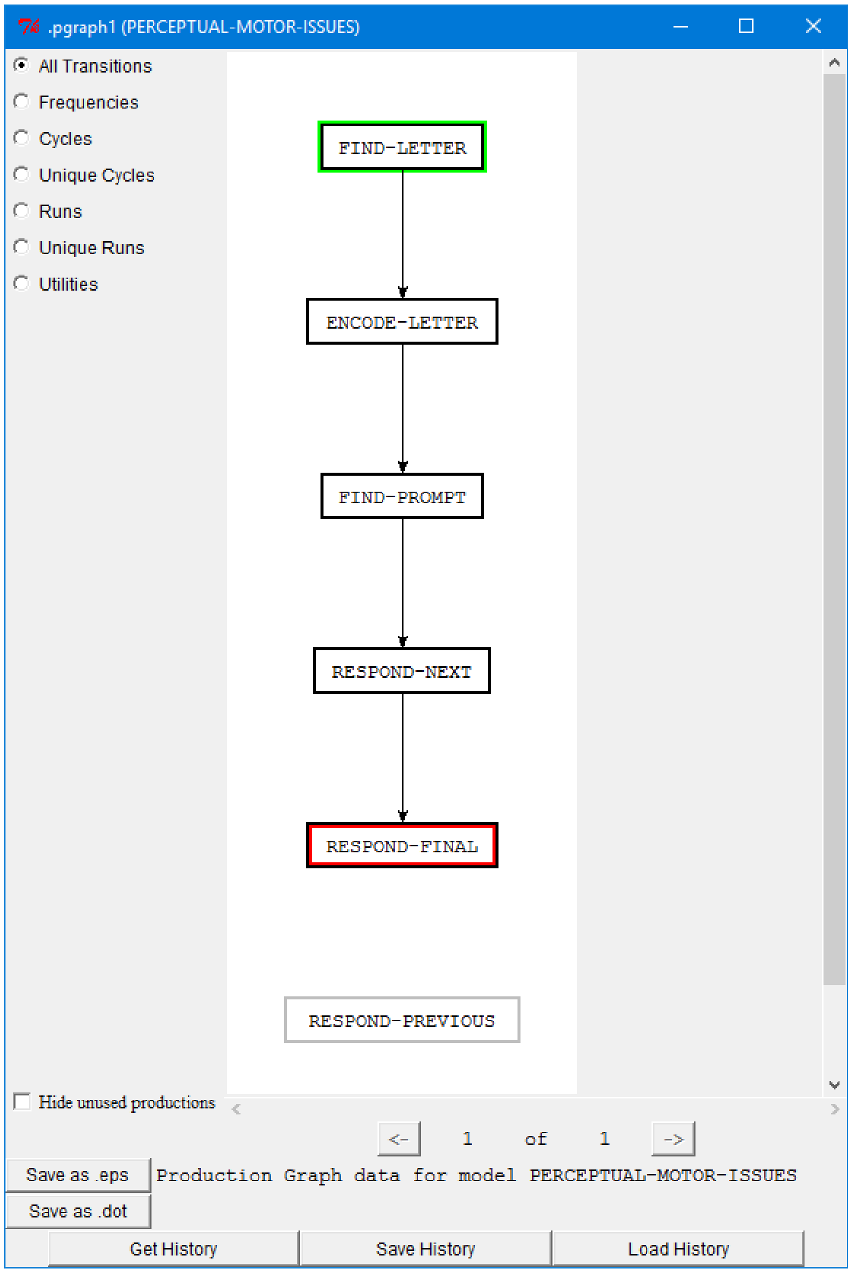


On the left we see the names of all the buffers used in the model (plus a line for productions) and along the bottom we see the time. For each buffer there are boxes displayed which correspond to the actions which occurred related to that buffer with text in the box to indicate what happened. The boxes in the production row show the names of the productions which fired, and for the other buffers they display the name of the chunk in the buffer at that time if there was one. Placing the mouse over a box will show additional information at the bottom of the window. That will show the request made to the buffer if that is why it was busy, additional notes if there are any for the action, the name of the chunk again, and the start and end times for the activity.

For this task, since the model was relatively small there may not have been much benefit to using the graphic trace over the text trace for debugging purposes. For larger models however it may be easier to find problems using the graphic trace because things like dependencies may be easier to see with the graphic representation. For example, it may be easier to see why encode-letter isn't selected until time .250 in the graph than in the text trace because the dependence on the completion of the **imaginal** buffer's action is more obvious.

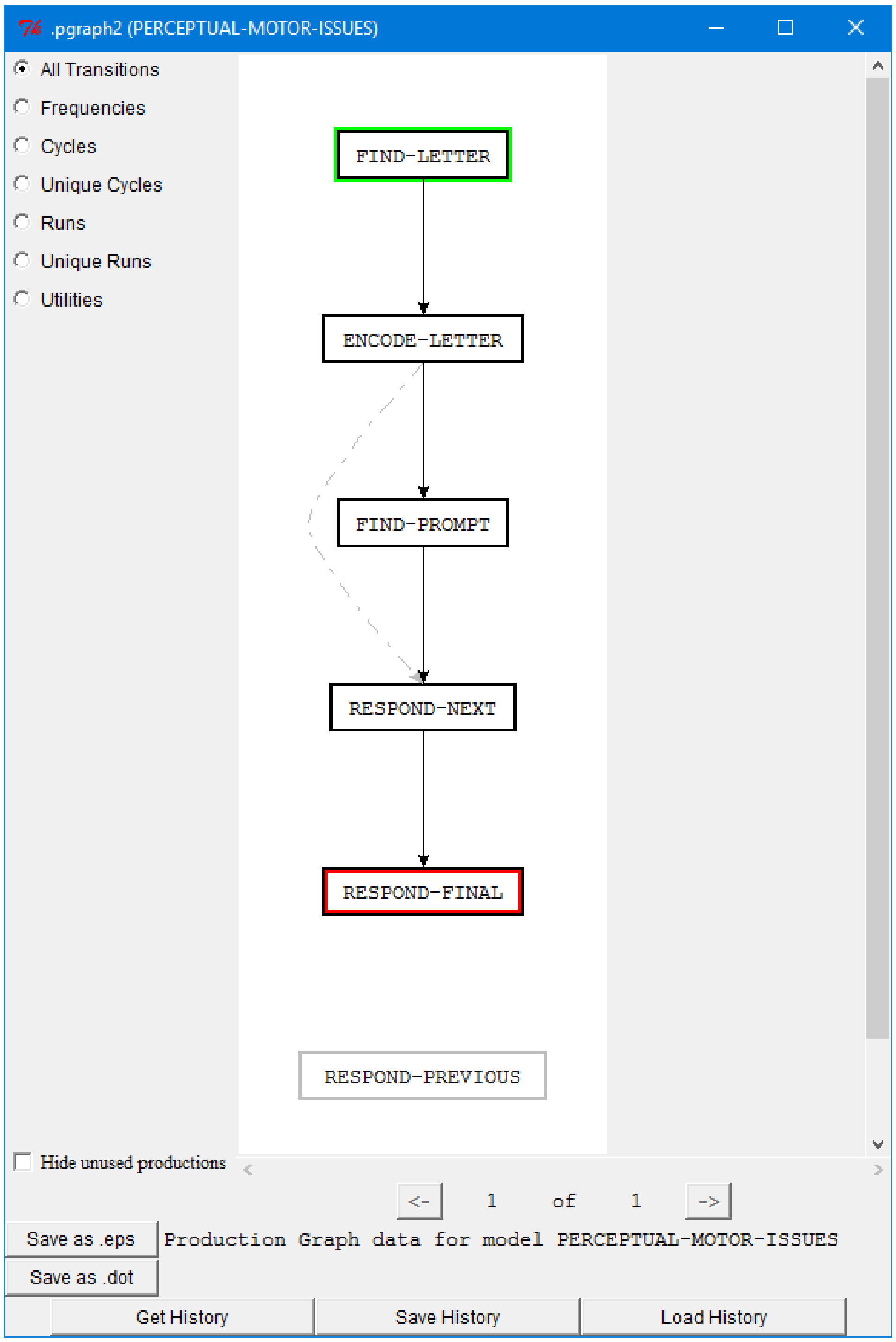
Production Graph

The "Production" tool can be used to show a graph of the production transitions which occur in the model. Select "graph" using the options menu button to the right of the "Production" button and then press the "Production" button to open the tool. Here is what that looks like for the final version of the model using the "All Transitions" display (after resizing the window to show the whole graph at once):



Production Graph data for model PERCEPTUAL-MOTOR-ISSUES

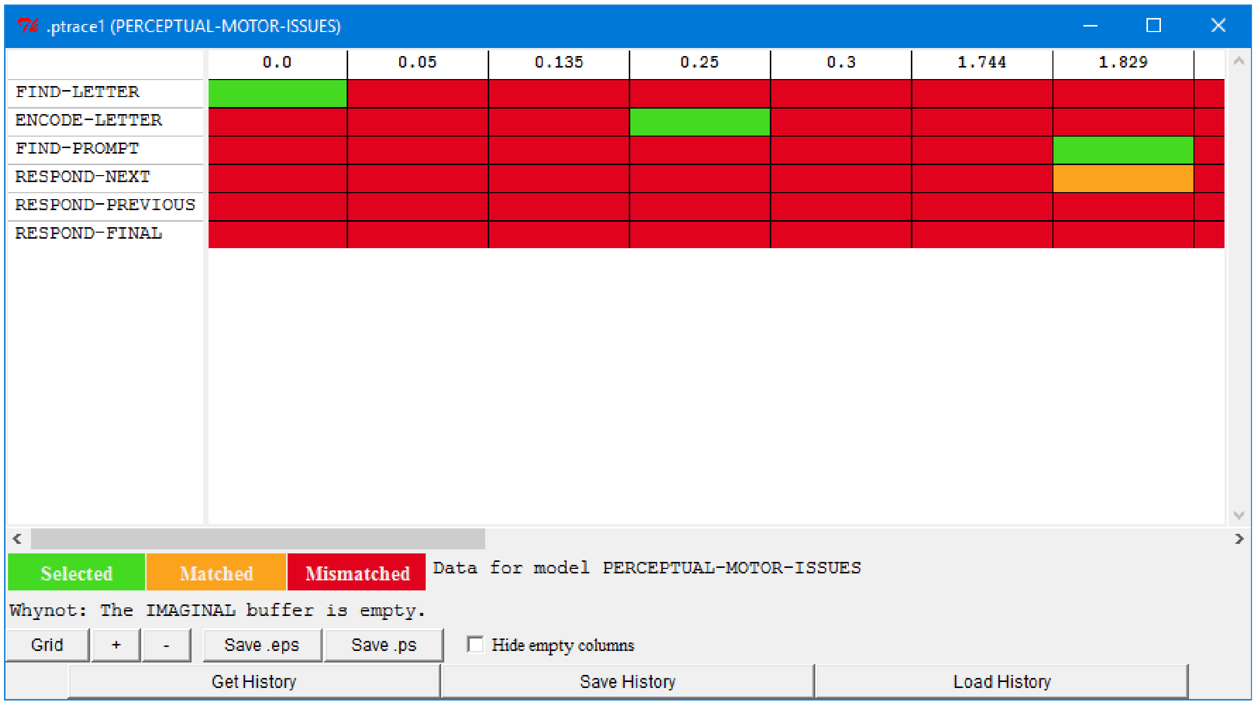
It shows the sequence of productions which occurred in the model from start (green) to end (red) along with productions which were not used (those in gray boxes). This provides an easy way to compare the model's production firings to what we would expect. It can also help with detecting problems along the way because it also shows productions which match but are not selected which would require turning on additional traces to see in the text trace. In particular here is a view of the graph for our model version 6 on a trial where it performed correctly:



The dotted line shows us that the respond-next production could have fired after encode-letter but didn't. That would have let us know that there was a problem without having to run additional tests to find a trial where the model actually responded incorrectly.

Production Grid

The "Production" tool can also be used to show a grid of the production selections which occur in the model. Select "grid" using the options menu button to the right of the "Production" button and then press the "Production" button to open the tool. The tool shows the production selection and firing information in a chart where each column corresponds to a conflict resolution action. Here is the same model run as shown in the graph above:



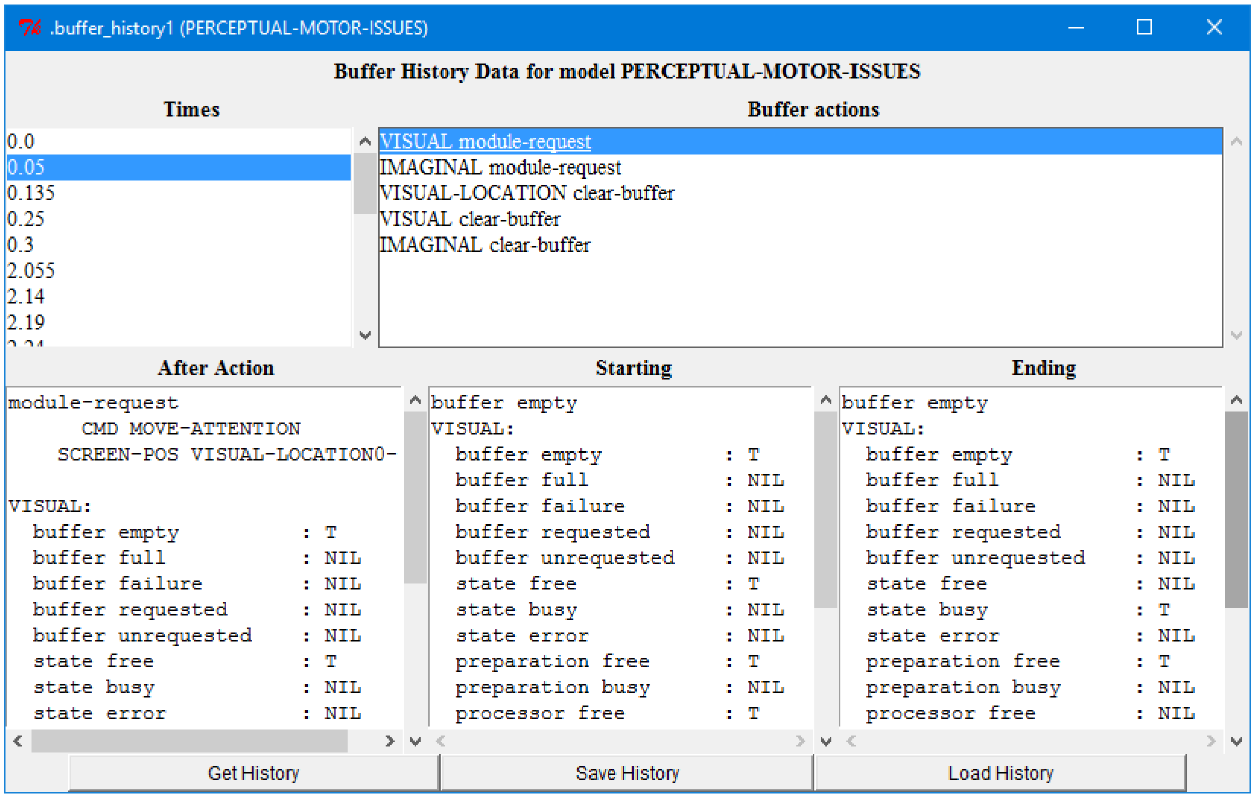
	0.0	0.05	0.135	0.25	0.3	1.744	1.829
FIND-LETTER	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-LETTER	🟥	🟥	🟥	🟩	🟥	🟥	🟥
FIND-PROMPT	🟥	🟥	🟥	🟥	🟥	🟥	🟩
RESPOND-NEXT	🟥	🟥	🟥	🟥	🟥	🟥	🟧
RESPOND-PREVIOUS	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND-FINAL	🟥	🟥	🟥	🟥	🟥	🟥	🟥

```
Data for model PERCEPTUAL-MOTOR-ISSUES
Whynot: The IMAGINAL buffer is empty.
```

The green boxes are the selected productions, red means it did not match, and orange means that it matched but was not selected (those colors can be changed by clicking on the corresponding box in the lower left of the tool). In addition to that, the tool will also display the whynot information for the unselected productions at the bottom when the mouse cursor is placed over the red boxes to show why that production was not selected during that specific conflict resolution event. In the image above the cursor was over the encode-letter box at time 0.0 and we see at the bottom that the whynot information indicates that it didn't match because the **imaginal** buffer was empty. In longer running models having all the whynot information recorded for inspection afterwards can be much easier than stepping through the model to particular times and then requesting the whynot information.

Buffers

The “Buffer History” tool records all of the changes which occur to all of the buffers during a run. Here is the display for a run of the final model in this task:



Buffer History Data for model PERCEPTUAL-MOTOR-ISSUES

Times

```
0.0
0.05
0.135
0.25
0.3
2.055
2.14
2.19
```

Buffer actions

```
VISUAL module-request
IMAGINAL module-request
VISUAL-LOCATION clear-buffer
VISUAL clear-buffer
IMAGINAL clear-buffer
```

After Action

```
module-request
      CMD MOVE-ATTENTION
    SCREEN-POS VISUAL-LOCATION0-

VISUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
```

Starting

```
buffer empty
VISUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  preparation free      : T
  preparation busy      : NIL
  processor free        : T
```

Ending

```
buffer empty
VISUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : NIL
  state busy            : T
  state error           : NIL
  preparation free      : T
  preparation busy      : NIL
  processor free        : NIL
```

In the upper left are all the times at which some buffer change occurred in the model. If you select a time then all of the buffer actions which occurred at that time will be displayed in the upper right. Selecting one of those will then fill in the bottom three displays. The one on the left shows the details of the action made and the buffer status information at that time. The one in the middle shows the chunk which was in the buffer at the start of this time step (or a notice that it was empty) and the buffer status information at that point, and the one on the bottom right shows the contents of the buffer and the status at the end of that time step.

Original Style Warnings

One last thing to look at is what we would have seen if we had left the style warnings enabled with the starting model. That would have resulted in seeing these two warnings:

```
#|Warning: Production FIND-LETTER makes a request to buffer IMAGINAL without a query
in the conditions. |#
#|Warning: Production RESPOND-FINAL makes a request to buffer MANUAL without a query
in the conditions. |#
```

Both of those warnings indicate problems which we fixed while working through the model. Had we seen them in advance we likely would have immediately added state free queries to the productions indicated because that is the obvious thing to do when warned about making a request without querying the module. That would have avoided the motor module jamming issue we encountered, but would not actually have changed the problem with find-letter firing again because the **imaginal** module was also free at the time of that second firing. Thus if we had left the warnings on (which is strongly recommended) it would have saved us some time debugging the model, but just fixing the warnings would not have been enough to make the model run correctly right from the start.

Unit 4: Activation of Chunks and Base-Level Learning

There are two objectives of this unit. The first is to introduce the subsymbolic quantity of activation associated with chunks. The other is to show how those activation values are learned through the history of usage of the chunks.

4.1 Introduction

We have seen **retrieval** requests in productions many times in the tutorial, like this one from the count model in unit 1:

```
(p start
   =goal>
      ISA         count-from
      start       =num1
      count       nil
 ==>
   =goal>
      ISA         count-from
      count       =num1
   +retrieval>
      ISA         number
      number      =num1
   )
```

In this case an attempt is being made to retrieve a chunk with a particular number (bound to **=num1**) in its **number** slot. Up to now we have been working with the system at the symbolic level. If there was a chunk that matched that **retrieval** request it would be placed into the **retrieval** buffer, and if not then the request would fail and the buffer would indicate the failure. The system was deterministic and we did not consider any timing cost associated with that memory retrieval or the possibility that a matching chunk in declarative memory might fail to be retrieved. For the simple tasks we have looked at so far that was sufficient.

Most psychological tasks however are not that simple and issues such as accuracy and latency are measured over time or across different conditions. For modeling these more involved tasks one will typically need to use the subsymbolic components of ACT-R to accurately model and predict human performance. For the remainder of the tutorial, we will be looking at the subsymbolic components that control the performance of the system. To use the subsymbolic components we need to turn them on by setting the :esc parameter (enable subsymbolic computations) to **t**:

```
(sgp :esc t)
```

That setting will be included in all of the models from this point on in the tutorial.

4.2 Activation

Every chunk in ACT-R's declarative memory has associated with it a numerical value called its activation. The activation reflects the degree to which past experiences and current context indicate that chunk will be useful at any particular moment. When a **retrieval** request is made the chunk with the greatest activation among those that match the specification of the request will be the one placed into the **retrieval** buffer. There is one constraint on that however. There is another parameter called the retrieval threshold which sets the minimum activation a chunk can have and still be retrieved. It is set with the :rt parameter:

```
(sgp :rt -0.5)
```

If the chunk with the highest activation among those that match the request has an activation which is less than the retrieval threshold, then no chunk will be placed into the **retrieval** buffer and a buffer failure will be indicated.

The activation A_i of a chunk *i* is computed from three components – the base-level, a context component, and a noise component. We will discuss the context component in the next unit. So, for now the activation equation is:

$$A_i = B_i + \varepsilon_i$$

B_i**:** The base-level activation. This reflects the recency and frequency of practice of the chunk *i*.

ε_i: The noise value. The noise is composed of two components: a permanent noise which is associated with each chunk when it is added to declarative memory and an instantaneous noise computed for each chunk at the time of a **retrieval** request.

We will discuss these components in detail below.

4.3 Base-level Learning

The equation describing learning of base-level activation for a chunk *i* is:

$$B_i = \ln(\sum_{j=1}^{n} t_j^{-d})$$

***n*:** The number of presentations for chunk *i*.

t_j: The time since the *jth* presentation.

d: The decay parameter which is set using the :bll (base-level learning) parameter. This parameter is almost always set to 0.5.

This equation describes a process in which each time an item is presented there is an increase in its base-level activation, which decays away as a power function of the time since that presentation. These decay effects are summed and then passed through a logarithmic transformation.

There are two types of events that are considered as presentations of a chunk. The first is its initial entry into declarative memory. The other is when a chunk merges with a chunk that is already in declarative memory. The next two subsections describe those events in more detail.

4.3.1 Chunks Entering Declarative Memory

When a chunk is initially entered into declarative memory is counted as its first presentation. There are two ways for a chunk to be entered into declarative memory, both of which have been discussed in the previous units. They are:

- Explicitly by the modeler using the **add-dm** command. These chunks are entered at the time the call is executed, which is time 0 for a call in the body of the model definition.

- When the chunk is cleared from a buffer. We have seen this happen in many of the previous models as visual locations, visual objects, and goal chunks are cleared from their buffers they can then be found among the chunks in declarative memory.

4.3.2 Chunk Merging

Something we have not seen previously is what happens when the chunk cleared from a buffer is an identical match to a chunk which is already in declarative memory. If a chunk has the same set of slots and values as a chunk which already exists in the model's declarative memory then instead of being added to declarative memory that chunk goes through a process we refer to as merging with the existing chunk in declarative memory. Instead of adding the new chunk to declarative memory the preexisting chunk in declarative memory is credited with a presentation, and the name of the chunk that was cleared from the buffer now references the chunk that was already in declarative memory i.e. there is one chunk which now has two (or possibly more) names.

4.4 Optimized Learning

Because of the need to separately calculate the effect of each presentation, the learning rule is computationally expensive and for some models the real time cost of computation is too great to be able to actually run the model in a reasonable amount time. To reduce the computational cost there is an approximation that one can use when the presentations are approximately uniformly distributed over the time since the item was created. This approximation can be enabled by turning on the optimized learning parameter - :ol. In fact, its default setting is on (the value **t**). When optimized learning is enabled, the following equation applies:

$$B_i = \ln\left(\frac{n}{1-d}\right) - d * \ln(L)$$

***n*:** The number of presentations of chunk *i*.

***L*:** The lifetime of chunk *i* (the time since its creation).

***d*:** The decay parameter.

4.5 Noise

The noise component of the activation equation contains two sources of noise. There is a permanent noise which can be associated with a chunk and an instantaneous noise value which will be recomputed at each retrieval attempt. Both noise values are generated according to a logistic distribution characterized by a parameter *s*. The mean of the logistic distribution is 0 and the variance, σ^2, is related to the *s* value by this equation:

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

The permanent noise *s* value is set with the :pas parameter and the instantaneous noise *s* value is set with the :ans parameter. Typically, we are only concerned with the instantaneous noise (the variance from trial to trial) and leave the permanent noise turned off (a value of **nil**).

4.6 Probability of Recall

If we make a **retrieval** request and there is a matching chunk in declarative memory, that chunk will only be retrieved if its activation exceeds the retrieval threshold, τ. The probability of this happening depends on the expected activation of the chunk (its activation without the instantaneous noise), A_i, and the amount of instantaneous noise in the system based on its *s* parameter:

$$recallprobability_i = \frac{1}{1 + e^{\frac{\tau - A_i}{s}}}$$

Inspection of that formula shows that, as A_i tends higher, the probability of recall approaches 1, whereas, as τ tends higher, the probability decreases. In fact, when $\tau = A_i$, the probability of recall is .5. The *s* parameter controls the sensitivity of recall to changes in activation. If *s* is close to 0, the transition from near 0% recall to near 100% will be abrupt, whereas when *s* is larger, the transition will be a slow sigmoidal curve. It is important to note however that this is only a description of what can happen with the retrieval of a chunk. When a **retrieval** request is made the chunk's activation plus the instantaneous noise will either be above the threshold or not.

4.7 Retrieval Latency

The activation of a chunk also determines how quickly it can be retrieved. When a **retrieval** request is made, the time it takes until the chunk that is retrieved is available in the **retrieval** buffer is given by this equation:

$$Time = Fe^{-A}$$

A: The activation of the chunk which is retrieved.

F: The latency factor (set using the :lf parameter).

If no chunk matches the **retrieval** request, or no chunk has an activation which is greater than the retrieval threshold then a failure will occur. The time it takes for the failure to be signaled is:

$$Time = Fe^{-\tau}$$

τ: The retrieval threshold.

F: The latency factor.

4.8 The Paired-Associate Example

Now that we have described how activation works, we will look at an example model which shows the effect of base-level learning. Anderson (1981) reported an experiment in which subjects studied and recalled a list of 20 paired associates for 8 trials. The paired associates consisted of 20 nouns like "house" associated with the digits 0 - 9. Each digit was used as a response twice. Below is the mean percent correct and mean latency to type the correct digit for each of the trials. Note subjects got 0% correct on the first trial because they were just studying them for the first time and the mean latency is 0 only because there were no correct responses.

Trial	Accuracy	Latency
1	.000	0.000
2	.526	2.156
3	.667	1.967
4	.798	1.762
5	.887	1.680

6	.924	1.552
7	.958	1.467
8	.954	1.402

The paired-model.lisp file in the unit4 directory contains a model for this task and the code to run the experiment can be found in the paired file in the Lisp and Python directories. The experiment code is written to allow one to run a general form of the experiment. Both the number of pairs to present and the number of trials to run can be specified. You can run the model through n trials of m paired associates (m no greater than 20) with the paired-task function in the Lisp version and the task function in the paired module of the Python version:

```
? (paired-task m n)

>>> paired.task(m,n)
```

If you would like to do the task as a person you can provide a true value for the optional third parameter. To run yourself through 3 trials with 2 pairs that would look like this:

```
? (paired-task 2 3 t)

>>> paired.task(2,3,True)
```

For each of the m words you will see the stimulus for 5 seconds during which you have the opportunity to make your response. Then you will see the associated number for 5 seconds. The simplest form of the experiment is one in which a single pair is presented twice. To run the model through that task use the appropriate one of these function calls:

```
? (paired-task 1 2)

>>> paired.task(1,2)
```

Here is the trace of the model doing such a task. The first time the model has an opportunity to learn the pair and the second time it has a chance to recall the response from that learned pair:

```
 0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL NIL
 0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
 0.000   VISION         visicon-update
 0.000   PROCEDURAL     CONFLICT-RESOLUTION
 0.050   PROCEDURAL     PRODUCTION-FIRED ATTEND-PROBE
 0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
 0.050   PROCEDURAL     CLEAR-BUFFER VISUAL
 0.050   PROCEDURAL     CONFLICT-RESOLUTION
 0.135   VISION         Encoding-complete VISUAL-LOCATION0-0 NIL
 0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
 0.135   PROCEDURAL     CONFLICT-RESOLUTION
 0.185   PROCEDURAL     PRODUCTION-FIRED READ-PROBE
 0.185   PROCEDURAL     CLEAR-BUFFER VISUAL
 0.185   PROCEDURAL     CLEAR-BUFFER IMAGINAL
 0.185   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
 0.185   DECLARATIVE    start-retrieval
 0.185   PROCEDURAL     CONFLICT-RESOLUTION
 0.385   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
 0.385   PROCEDURAL     CONFLICT-RESOLUTION
 3.141   DECLARATIVE    RETRIEVAL-FAILURE
 3.141   PROCEDURAL     CONFLICT-RESOLUTION
 3.191   PROCEDURAL     PRODUCTION-FIRED CANNOT-RECALL
 3.191   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
 3.191   PROCEDURAL     CLEAR-BUFFER VISUAL
 3.191   VISION         CLEAR
 3.191   PROCEDURAL     CONFLICT-RESOLUTION
 3.241   PROCEDURAL     CONFLICT-RESOLUTION
 5.000   ------         Stopped because time limit reached
 5.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1 NIL
 5.000   VISION         visicon-update
 5.000   PROCEDURAL     CONFLICT-RESOLUTION
 5.050   PROCEDURAL     PRODUCTION-FIRED DETECT-STUDY-ITEM
 5.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
 5.050   PROCEDURAL     CLEAR-BUFFER VISUAL
 5.050   PROCEDURAL     CONFLICT-RESOLUTION
 5.135   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
 5.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
 5.135   PROCEDURAL     CONFLICT-RESOLUTION
 5.185   PROCEDURAL     PRODUCTION-FIRED ASSOCIATE
 5.185   PROCEDURAL     CLEAR-BUFFER IMAGINAL
 5.185   PROCEDURAL     CLEAR-BUFFER VISUAL
 5.185   VISION         CLEAR
 5.185   PROCEDURAL     CONFLICT-RESOLUTION
 5.235   PROCEDURAL     CONFLICT-RESOLUTION
10.000   ------         Stopped because time limit reached
10.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2 NIL
10.000   VISION         visicon-update
10.000   PROCEDURAL     CONFLICT-RESOLUTION
10.050   PROCEDURAL     PRODUCTION-FIRED ATTEND-PROBE
10.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
10.050   PROCEDURAL     CLEAR-BUFFER VISUAL
10.050   PROCEDURAL     CONFLICT-RESOLUTION
10.135   VISION         Encoding-complete VISUAL-LOCATION2-0 NIL
10.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
10.135   PROCEDURAL     CONFLICT-RESOLUTION
10.185   PROCEDURAL     PRODUCTION-FIRED READ-PROBE
10.185   PROCEDURAL     CLEAR-BUFFER VISUAL
10.185   PROCEDURAL     CLEAR-BUFFER IMAGINAL
10.185   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
10.185   DECLARATIVE    start-retrieval
10.185   PROCEDURAL     CONFLICT-RESOLUTION
10.385   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK1
10.385   PROCEDURAL     CONFLICT-RESOLUTION
11.145   DECLARATIVE    RETRIEVED-CHUNK CHUNK0-0
11.145   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL CHUNK0-0
11.145   PROCEDURAL     CONFLICT-RESOLUTION
11.195   PROCEDURAL     PRODUCTION-FIRED RECALL
11.195   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
```

```
11.195   PROCEDURAL     CLEAR-BUFFER MANUAL
11.195   PROCEDURAL     CLEAR-BUFFER VISUAL
11.195   MOTOR          PRESS-KEY KEY 9
11.195   VISION         CLEAR
11.195   PROCEDURAL     CONFLICT-RESOLUTION
11.245   PROCEDURAL     CONFLICT-RESOLUTION
11.445   PROCEDURAL     CONFLICT-RESOLUTION
11.495   PROCEDURAL     CONFLICT-RESOLUTION
11.595   PROCEDURAL     CONFLICT-RESOLUTION
11.745   PROCEDURAL     CONFLICT-RESOLUTION
15.000   ------         Stopped because time limit reached
15.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3 NIL
15.000   VISION         visicon-update
15.000   PROCEDURAL     CONFLICT-RESOLUTION
15.050   PROCEDURAL     PRODUCTION-FIRED DETECT-STUDY-ITEM
15.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
15.050   PROCEDURAL     CLEAR-BUFFER VISUAL
15.050   PROCEDURAL     CONFLICT-RESOLUTION
15.135   VISION         Encoding-complete VISUAL-LOCATION3-0 NIL
15.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT3
15.135   PROCEDURAL     CONFLICT-RESOLUTION
15.185   PROCEDURAL     PRODUCTION-FIRED ASSOCIATE
15.185   PROCEDURAL     CLEAR-BUFFER IMAGINAL
15.185   PROCEDURAL     CLEAR-BUFFER VISUAL
15.185   VISION         CLEAR
15.185   PROCEDURAL     CONFLICT-RESOLUTION
15.235   PROCEDURAL     CONFLICT-RESOLUTION
20.000   ------         Stopped because time limit reached
```

The basic structure of the screen processing productions should be familiar by now. The one thing to note is that because this model must wait for stimuli to appear on screen it takes advantage of the buffer stuffing mechanism in the **visual-location** buffer so that it can wait for the change instead of continuously checking. The way it does that is by having the first production that will match, for either the probe or the associated number, have a **visual-location** buffer test on its LHS and no productions which make requests for visual-locations. Thus, those productions will only match once buffer stuffing places a chunk into the **visual-location** buffer. Here are the **attend-probe** and **detect-study-item** productions for reference:

```
(p attend-probe
    =goal>
      isa      goal
      state    start
    =visual-location>
    ?visual>
     state     free
   ==>
    +visual>
      cmd      move-attention
      screen-pos =visual-location
    =goal>
      state    attending-probe
)
```

```
(p detect-study-item
    =goal>
      isa      goal
      state    read-study-item
    =visual-location>
    ?visual>
      state    free
   ==>
    +visual>
      cmd      move-attention
      screen-pos =visual-location
    =goal>
      state    attending-target
)
```

Because the buffer is cleared automatically by strict harvesting and no later productions issue a request for a visual-location these productions must wait for buffer stuffing to put a chunk into the **visual-location** buffer before they can match. Since none of the other productions match in the mean time the model will just wait for the screen to change before doing anything else.

Now we will focus on the productions which are responsible for forming the association and retrieving the chunk. When the model attends to the probe with the **read-probe** production two actions are taken (in addition to the updating of the **goal** state):

```
(p read-probe
    =goal>
      isa      goal
      state    attending-probe
    =visual>
      isa      visual-object
      value    =val
    ?imaginal>
      state    free
   ==>
    +imaginal>
      isa      pair
      probe    =val
    +retrieval>
      isa      pair
      probe    =val
    =goal>
      state    testing
)
```

It makes a request to the **imaginal** buffer to create a chunk which will hold the value read from the screen in the probe slot. It also makes a request to the **retrieval** buffer to retrieve a chunk from declarative memory which has that same value in the probe slot.

We will come back to the **retrieval** request shortly. For now we will focus on the creation of the chunk for representing the pair of items in the **imaginal** buffer.

The **associate** production fires after the model reads the number which is associated with the probe:

```
(p associate
    =goal>
      isa      goal
      state    attending-target
    =visual>
      isa      visual-object
      value    =val
   =imaginal>
      isa      pair
      probe    =probe
    ?visual>
      state    free
  ==>
   =imaginal>
      answer   =val
   -imaginal>
   =goal>
      state    start
   +visual>
      cmd      clear
)
```

This production sets the answer slot of the chunk in the **imaginal** buffer to the answer which was read from the screen. It also clears that chunk from the buffer so that it is entered into declarative memory. That will result in a chunk like this being added to the model's declarative memory:

```
CHUNK0-0
   PROBE  "zinc"
   ANSWER  "9"
```

This chunk serves as the memory of this trial. An important thing to note is that the chunk in the buffer is not added to the model's declarative memory until that buffer is cleared. Often that happens when the model later harvests that chunk from the buffer, but in this case the model does not harvest the chunk later so it is explicitly cleared in the last production which modifies it. One could imagine adding additional productions which would rehearse that information clearing the buffer as it does so, but for the demonstration model that is not done.

This production also makes a request to the **visual** buffer to stop attending to the item. That is done so that the model does not perform the automatic re-encoding when the screen is updated.

Now, consider the **retrieval** request in the read-probe production again:

```
    +retrieval>
      isa      pair
      probe    =val
```

The declarative memory module will attempt to retrieve a chunk with the requested probe value. Depending on whether a chunk can be retrieved, one of two production rules may apply corresponding to either the successful retrieval of such a chunk or the failure to retrieve a matching chunk:

```
(p recall
    =goal>
      isa      goal
      state    testing
    =retrieval>
      isa      pair
      answer   =ans
    ?manual>
      state    free
    ?visual>
      state    free
   ==>
    +manual>
      cmd      press-key
      key      =ans
    =goal>
      state    read-study-item
    +visual>
      cmd      clear
)

(p cannot-recall
    =goal>
      isa      goal
      state    testing
    ?retrieval>
```

```
      buffer   failure
    ?visual>
      state    free
   ==>
    =goal>
     state    read-study-item
    +visual>
     cmd      clear
)
```

The probability of the recall production firing and the mean latency for the recall will be determined by the activation of the corresponding chunk. The probability will increase with repeated presentations and successful retrievals, while the latency will decrease. That is because each repeated presentation will result in creating a new chunk in the **imaginal** buffer which will merge with the existing chunk for that trial in declarative memory thus increasing its activation. Similarly, when it successfully retrieves a chunk for a trial and the recall production fires that chunk in the **retrieval** buffer will be cleared by the strict harvesting mechanism and then merge with the chunk in declarative memory again increasing its activation.

This model gives a pretty good fit to the data as illustrated below in a run of 100 simulated subjects using the paired-experiment function in Lisp or the experiment function from the paired module in Python (because of stochasticity the results are more reliable if there are more runs and to generate that many runs in a reasonable amount of time one must turn off the trace and remove the seed parameter to allow for differences from run to run):

```
? (paired-experiment 100)

>>> paired.experiment(100)
```

```
Latency:
CORRELATION:  0.997
MEAN DEVIATION:  0.090
Trial    1       2       3       4       5       6       7       8
        0.000   2.173   1.886   1.682   1.549   1.451   1.350   1.301

Accuracy:
CORRELATION:  0.992
MEAN DEVIATION:  0.049
Trial    1       2       3       4       5       6       7       8
        0.000   0.553   0.773   0.879   0.910   0.929   0.949   0.954
```

4.9 Parameter estimation

To get the model to fit the data requires not only writing a plausible set of productions which can accomplish the task, but also setting the ACT-R parameters that control the behavior as described in the equations governing the operation of declarative memory. Running the model without enabling the base-level learning component of declarative memory produces results like this:

```
Latency:
CORRELATION:  0.925
MEAN DEVIATION:  0.280
Trial    1       2       3       4       5       6       7       8
        0.000   1.556   1.551   1.541   1.550   1.545   1.546   1.546

Accuracy:
CORRELATION:  0.884
MEAN DEVIATION:  0.223
Trial    1       2       3       4       5       6       7       8
        0.000   1.000   1.000   1.000   1.000   1.000   1.000   1.000
```

That shows perfect recall after one presentation and essentially no difference in the time to respond across trials. Just turning on base-level learning by specifying the :bll parameter with the recommended value of .5 results in these results:

```
Latency:
CORRELATION:  0.000
MEAN DEVIATION:  1.619
Trial    1       2       3       4       5       6       7       8
        0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000


Accuracy:
CORRELATION:  0.000
MEAN DEVIATION:  0.777
Trial    1       2       3       4       5       6       7       8
        0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

That shows a complete failure to retrieve any of the facts which would happen because they have an activation below the retrieval threshold (which defaults to 0). Lowering the retrieval threshold so that they can be retrieved results in something like this:

```
Latency:
CORRELATION:  0.939
MEAN DEVIATION:  1.522

Trial    1       2       3       4       5       6       7       8
        0.000   3.413   3.841   4.071   3.693   2.968   2.567   2.321

Accuracy:
CORRELATION:  0.821
MEAN DEVIATION:  0.296
```

```
Trial    1        2        3        4        5        6        7        8
       0.000    0.050    0.080    0.470    1.000    1.000    1.000    1.000
```

That shows some of the general trends, but does not fit the data well. The behavior of this model and the one that you will write for the assignment of this unit really depends on the settings of four parameters. Here are those parameters and their settings in this model. The retrieval threshold is set at -2. This determines how active a chunk has to be to be retrieved 50% of the time. The instantaneous activation noise is set at 0.5. This determines how quickly probability of retrieval changes as we move past the threshold. The latency factor is set at 0.4. This determines the magnitude of the activation effects on latency. Finally, the decay rate for base-level learning is set to the value 0.5 which is where we recommend it be set for most tasks that involve the base-level learning mechanism.

How to determine those values can be a tricky process because the equations are all related and thus they cannot be independently manipulated for a best fit. Typically some sort of searching is required, and there are many ways to accomplish that. For the tutorial models there will typically be only one or two parameters that you will need to adjust and we recommend that you work through the process "by hand" adjusting the parameters individually to see the effect that they have on the model. There are other ways of determining parameters that can be used, but we will not be covering any such mechanisms in the tutorial.

4.10 The Activation trace

A parameter named :act is also set in the sgp call in this model. This is the activation trace parameter. If it is turned on, it causes the declarative memory system to print the details of the activation computations that occur during a **retrieval** request in the trace. If you set it to **t** and reload the model and run for two trials of one pair (like the trace above) you will find these additional details where the retrieval requests are handled by the declarative module:

```
 0.185   DECLARATIVE    start-retrieval
No matching chunk found retrieval failure
...
10.185   DECLARATIVE    start-retrieval
Chunk CHUNK0-0 matches
Computing activation for chunk CHUNK0-0
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (5.185)
  creation time: 5.185 decay: 0.5  Optimized-learning: T
base-level value: -0.11157179
Total base-level: -0.11157179
Adding transient noise 0.29328185
Adding permanent noise 0.0
Chunk CHUNK0-0 has an activation of: 0.18171006
Chunk CHUNK0-0 has the current best activation 0.18171006
Chunk CHUNK0-0 with activation 0.18171006 is the best
...
```

You may find this detailed accounting of the activation computation useful in debugging your models and in understanding how the system computes activation values.

4.11 The :ncnar Parameter

There is one final parameter being set in the model which you have not seen before - :ncnar (normalize chunk names after run). This parameter does not affect the model's performance on the tasks, but it does affect the actual time it takes to run the simulation. The details on what exactly it does can be found in the code description text for this unit. The reason it is turned off (set to **nil**) in the models for this unit is to decrease the time it takes to run the simulations.

4.12 Unit Exercise: Alpha-Arithmetic

The following data were obtained by N. J. Zbrodoff on judging alphabetic arithmetic problems. Participants were presented with an equation like A + 2 = C and had to respond yes or no whether the equation was correct based on counting in the alphabet – the preceding equation is correct, but B + 3 = F is not.

She manipulated whether the addend was 2, 3, or 4 and whether the problem was true or false. She had 2 versions of each of the 6 kinds of problems (3 addends x 2 responses) each with a different letter (a through f). She then manipulated the frequency with which problems were studied in sets of 24 trials:

- In the Control condition, each of the 2, 3, and 4 addend problems occurred twice.
- In the Standard condition, the 2 addend problems occurred three times, the 3 addend problems twice, and the 4 addend problems once.
- In the Reverse condition, the 2 addend problems occurred once, the 3 addend problems twice, and the 4 addend problems three times.

Each participant saw problems based on one of the three conditions. There were 8 repetitions of a set of 24 problems in a block (192 problems), and there were 3 blocks for 576 problems in all. The data presented below are in seconds to correctly judge the problems true or false based on the block and the addend. They are aggregated over both true and false responses:

`Control Group (all problems equally frequently)`

	Two	Three	Four
Block 1	1.840	2.460	2.820
Block 2	1.210	1.450	1.420
Block 3	1.140	1.210	1.170

`Standard Group (smaller problems more frequent)`

	Two	Three	Four
Block 1	1.840	2.650	3.550

```
Block 2   1.060   1.450   1.920
Block 3   0.910   1.080   1.480

Reverse Group (larger problems more frequent)
           Two    Three    Four
Block 1   2.250   2.530   2.440
Block 2   1.470   1.460   1.100
Block 3   1.240   1.120   0.870
```

The interesting phenomenon concerns the interaction between the effect of the addend and amount of practice. Presumably, the addend effect originally occurs because subjects have to engage in counting, but latter they come to rely mostly on retrieval of answers they have stored from previous computations.

The task for this unit is to develop a model of the control group data. Functions to run the experiment can be found in the appropriate zbrodoff file and most of a model that can perform the task is provided in the zbrodoff-model.lisp file with the unit materials. The model as given does the task by counting through the alphabet and numbers “in its head” to arrive at an answer which it compares to the initial equation to determine how to respond. The timing for this model to complete the task is determined by the perceptual and motor actions which it performs: reading the display, subvocalizing the items as it counts through the alphabet (similar to the speak action we saw earlier in the tutorial for the speech module), and pressing the response key. Here is the performance of this model on the task when run through the whole experiment once:

```
CORRELATION:  0.295
MEAN DEVIATION:  1.824

              2 (64)      3 (64)      4 (64)
Block  1  2.598 (64)  3.401 (64)  3.989 (64)
Block  2  2.590 (64)  3.389 (64)  3.984 (64)
Block  3  2.595 (64)  3.400 (64)  3.990 (64)
```

It is always correct (the 64 on the top row indicates how many presentations per cell and the number correct is then displayed for each cell) but it does not get any faster from block to block because it always uses the counting strategy. Your first task is to extend the model so that it attempts to remember previous instances of the trials. If it can remember the answer it does not have to resort to the counting strategy and can respond much faster.

The model encodes each trail in a chunk which has the result of its counting for the trial. A completed problem for a trial where the stimulus was “A+2 = C” would look like this:

```
CHUNK2-0
   RESULT  C
   ARG1  A
```

```
    ARG2  TWO
```

The result slot contains the result of counting 2 letters from A.

To do the counting, the model uses slots of the **goal** buffer to hold the target answer and the numbers and letters as it counts, and it encodes the result of the final count in the **imaginal** buffer. It only counts as far as it needs to and then compares the counted result to the presented answer to respond. Thus the same chunk will result from a trial where the stimulus presented is "A+2 = D" because it only counts A plus 2 and then compares the result of that, C, to the letter presented to determine if the problem is correct or not (since C is not D this problem would be incorrect). The assumption is that the person is actually learning the letter counting facts and not just memorizing the stimulus-response pairings for the task. The model will learn one chunk for each of the additions which it encounters, which will be a total of six after it completes a set of trials.

A strong recommendation for adding the retrieval strategy to the model is to continue to use the existing encoding productions before the retrieval, the existing response productions (**final-answer-yes** and **final-answer-no**) after a successful retrieval, and the given counting productions if it fails to retrieve. It may be necessary to modify productions at the end of the encoding process and/or the beginning of the counting process to add the retrieval process into the model, but the response productions should **not** be modified in any way and you should **not** add any additional productions for responding. Using the given response productions is important because they already handle the important steps necessary for the model to repeatedly perform this task successfully: they create a new goal chunk which will make sure the model is ready for the next trial, they make the correct response based on the comparison of the value read from the screen and the correct value of the sum which has been encoded in the **imaginal** buffer, and that **imaginal** buffer chunk is cleared so that it can enter declarative memory and strengthen the knowledge for that fact.

After your model is able to utilize a retrieval strategy along with the counting strategy given, your next step is to adjust the parameters so that the model's performance better fits the experimental data. The results should look something like this after you have the retrieval strategy working with the parameters as set in the starting model:

```
CORRELATION:  0.983
MEAN DEVIATION:  0.533

              2 (64)      3 (64)      4 (64)
Block  1  1.405 (64)  1.556 (64)  1.601 (64)
Block  2  1.175 (64)  1.273 (64)  1.250 (64)
Block  3  1.162 (64)  1.260 (64)  1.214 (64)
```

The model is responding correctly on all trials, the correlation is good, but the deviation is quite high because the model is too fast overall. The model's performance will depend on the same four parameters as the paired associate model: latency factor, activation noise, base-level decay rate, and retrieval threshold. In the model you are given, the first three are set to the same values as in the paired associate model and represent reasonable values for this task. The retrieval threshold (the :rt parameter) is set to its default value of 0. This is the parameter you should adjust first to improve the fit to the data. Here is our fit to the data adjusting only the retrieval threshold:

```
CORRELATION:  0.991
MEAN DEVIATION:  0.101

              2 (64)      3 (64)      4 (64)
Block  1  1.879 (64)  2.497 (64)  2.592 (64)
Block  2  1.263 (64)  1.482 (64)  1.420 (64)
Block  3  1.190 (64)  1.381 (64)  1.200 (64)
```

If you would like to try to fit the data even better then you could also adjust the latency factor and activation noise parameters as well. The base-level decay rate parameter should be left at the value .5 (that is a recommended value which should not be adjusted in most models). Here is our best fit with adjusting all three parameters:

```
CORRELATION:  0.996
MEAN DEVIATION:  0.065

              2 (64)      3 (64)      4 (64)
Block  1  1.970 (64)  2.446 (64)  2.775 (64)
Block  2  1.263 (64)  1.419 (64)  1.422 (64)
Block  3  1.176 (64)  1.291 (64)  1.258 (64)
```

This experiment is more complicated than the ones that you have seen previously. It runs continuously for many trials and the learning that occurs across trials is important. Thus the model cannot treat each trial as an independent event and be reset before each one as has been done for the previous units. While writing your model and testing the fit to the data you will probably want to test it on smaller runs than the whole task. There are five functions provided for running the experiment and subcomponents of the whole experiment.

The function to present a single problem to the model is called **zbrodoff-problem** in the Lisp version and **problem** in the zbrodoff module of the Python version. It takes four required parameters which are all single character strings and an optional fifth parameter. The first three parameters are the elements of the equation to present. The fourth is the correct key which should be pressed for the trial, where k is the correct key for a true problem and d for a false problem. The optional parameter indicates whether or not to show the task display. If the optional parameter is not provided then the window will not be shown (a virtual window will be used) and if it is a true value then it will show the task window. These examples would present the “a + 2 = c” problem (which is true) to the model with a window that is visible:

```
? (zbrodoff-problem "a" "2" "c" "k" t)
```

```
>>> zbrodoff.problem('a','2','c','k',True)
```

Here are the twelve different problems which are used in this experiment:

```
true:  a+2=c, d+2=f, b+3=3, e+3=h, c+4=g, f+4=j
false: a+2=d, d+2=g, b+3=f, e+3=i, c+4=h, f+4=k
```

The single problem function should be used until you are certain that your model is able to successfully use a retrieval strategy along with counting. To do that you will want to present the model with the same trial again and again and make sure that at some point it can retrieve the correct fact and respond correctly. You will also want to test it with both true and false facts to make sure it can retrieve the right information and respond correctly in both cases. Finally, you should check the model's declarative memory to make sure that it is only creating the correct facts – it should not be learning chunks which represent the wrong addition.

Once you are confident that your model is learning the correct chunks and can use both retrieval and counting to respond correctly you can use the functions to present a set or block of items: **zbrodoff-set** and **zbrodoff-block** in Lisp and **set** and **block** from the zbrodoff module in Python. Those will run the model over multiple trials and print the results. Each takes one optional parameter to control whether the task display is shown, and if it is not provided they will not show the display. The set function runs the model through 24 trials of the task presenting each problem twice in a randomly generated order. The block function runs through 192 trials, which is 8 repetitions of the 24 trial set. Here are examples of running a set of trials in Lisp and a block in Python:

```
? (zbrodoff-set)

              2 ( 8)      3 ( 8)      4 ( 8)
Block  1  2.307 ( 8)  2.807 ( 8)  3.295 ( 8)
```

```
>>> zbrodoff.block()

              2 (64)      3 (64)      4 (64)
Block  1  2.031 (64)  2.258 (64)  2.659 (64)
```

After making sure the model can successfully complete a set and block of trials then you will want to test it with the function which resets the model and then runs one pass through the whole experiment: **zbrodoff-experiment** in Lisp and **experiment** in the zbrodoff module in Python. It has two optional parameters. The first determines whether the task window is visible or not, and the second determines whether the results are printed. The defaults are to not show the window and to show the results, which is probably how you want to run it:

```
? (zbrodoff-experiment)

              2 (64)      3 (64)      4 (64)
Block  1  1.953 (64)  2.285 (64)  2.730 (64)
Block  2  1.226 (64)  1.184 (64)  1.403 (64)
Block  3  1.123 (64)  1.074 (64)  1.085 (64)
```

```
>>> zbrodoff.experiment()

              2 (64)      3 (64)      4 (64)
Block  1  2.298 (64)  2.812 (64)  3.300 (64)
Block  2  2.298 (64)  2.803 (64)  3.297 (64)
Block  3  2.287 (64)  2.798 (64)  3.290 (64)
```

If the model is able to successfully complete the experiment you can move on to the function that runs the experiment multiple times, averages the results, and compares the results to the human data: **zbrodoff-compare** in Lisp and **compare** in the zbrodoff module in Python. Those functions take one parameter indicating the number of times to run the full experiment. It may take a while to run, especially if you request a lot of trials. Here are some examples:

```
? (zbrodoff-compare 10)
CORRELATION:  0.288
MEAN DEVIATION:  1.309

              2 (64)      3 (64)      4 (64)
Block  1  2.296 (64)  2.799 (64)  3.296 (64)
Block  2  2.297 (64)  2.797 (64)  3.293 (64)
Block  3  2.298 (64)  2.796 (64)  3.295 (64)

>>> zbrodoff.compare(10)
CORRELATION:  0.288
MEAN DEVIATION:  1.311

              2 (64)      3 (64)      4 (64)
Block  1  2.301 (64)  2.798 (64)  3.300 (64)
Block  2  2.299 (64)  2.798 (64)  3.297 (64)
Block  3  2.294 (64)  2.796 (64)  3.301 (64)
```

An important thing to note is that among those functions the only ones that reset the model are the ones that run the full experiment. So if you are using the other functions while testing the model keep in mind that unless you explicitly reset the model (by pressing the “Reset” button on the Control Panel, reloading the model, or calling the ACT-R **reset** function from the ACT-R prompt or the **actr.reset** function in Python) then the model will still have all the chunks which it has learned since the last time it was reset (or loaded) in its declarative memory.

As you look at the starting model, you will see one additional setting at the end of the model definition which you have not seen before:

```
 (set-all-base-levels 100000 -1000)
```

This sets the base-level activation of all the chunks in declarative memory that exist when it is called (which are the number and letter chunks provided) to very large values by setting the parameters **n** and **L** of the optimized base-level equation for each one. The first parameter, 100000, specifies **n** and the second parameter, -1000, specifies the creation time of the chunk. This ensures that the initial chunks which encode the sequencing of numbers and letters maintain a very high base-level activation and do not fall below the retrieval threshold over the course of the task. The assumption is that counting and the order of the alphabet are very well learned tasks for the model and the human participants in the experiment and that knowledge does not have any significant effect of learning or decay during the course of the experiment.

Because this experiment involves a lot of trials and you need to run several experiments to get the average results of the model there are some additional things that can be done to improve the performance of running the experiment i.e. the real time it takes to run the model through the experiment not the simulated time the model reports for doing the task. Probably the most important will be to turn off the model's trace by setting the :v parameter to **nil**. The starting model has that setting, but while you are testing and debugging your addition of a retrieval process you will probably want to turn it back on by setting it to **t** so that you can see what is happening in your model. Something else which you will want to do when running the whole experiment is to close any of the Recordable data tools which you may have opened in the ACT-R Environment because there is a cost to recording the underlying data needed for those tools. You will also want to turn off the running indicator if you have enabled it because there is also a cost to update that display. The final thing which you may want to do is to use the Lisp version of the experiment from the ACT-R prompt instead of the version connected from Python because there is some additional cost to running the Python version of the experiments and for this task that might be noticeable over many runs.

References

Anderson, J.R. (1981). Interference: The relationship between response latency and response accuracy. *Journal of Experimental Psychology: Human Learning and Memory, 7*, 326-343.

Zbrodoff, N. J. (1995). Why is 9 + 7 harder than 2 + 3? Strength and interference as explanations of the problem-size effect. *Memory & Cognition, 23* (6), 689-700.

Unit 4 Code Description

There are only a couple of new commands used in the models for this unit and one of them, set-all-base-levels, was discussed in the unit text. So there is not really much of anything new to discuss about the functions used to run the models, but there is a different approach to running the models used for the zbrodoff experiment compared to the paired associate task and all of the previous units' tasks. We will start by describing the differences between those approaches, and then look at the code which implements them with the focus more on the overall structure of the experiment implementations and model interaction than the details of every function call.

So far you have seen what can be described as an iterative or "trial at a time" approach to writing the experiments for models. The experiments run by executing some setup code for a trial, running the model to completion on that trial, recording a result and then repeating that process for the next trial until all the trials are done. That style is a commonly used approach, but it has some drawbacks which you may have encountered. For instance, when running an experiment it is not possible to stop it using the Stop button in the Stepper tool of the ACT-R Environment because the Stop button only stops the current "run" of ACT-R, but it cannot affect the code which is making those calls to run things. Instead you have to terminate the execution of the task somehow, which can be difficult if it wasn't written with a way to do so.[1] For models with few trials or that get reset on each trial that may not be a significant issue, but for large experiments or models that need to learn from trial to trial that can make things difficult to work with, particularly if there is a problem with the model on a later trial that forces one to abandon a very long run by terminating the experiment code which likely doesn't have any way to continue where the model left off for debugging or investigating the problem.

An alternative way to write the experiments is with a more event-driven approach. The system which runs the ACT-R models is a general discrete-event simulation system which can be used to run other things as well, like an experiment for a model (or a person if precision timing information isn't required). Calling one of the ACT-R running commands causes all of the events which have been created to be executed in either a simulated time or real time sequence. Up to now those events have been mostly generated by the model, e.g. production firing, memory retrieval, and key presses, or by ACT-R commands like goal-focus. However, as was seen in the sperling task, arbitrary events can also be scheduled to execute at particular simulated times. One can also use the interface events generated by the model (like output-key) to do things other than just record the response. By scheduling events to occur at appropriate times and putting some of the control into the functions that handle the model's actions the experiment can run "with" the model instead of "around" it. Such an experiment only needs to call the run function one time to complete the whole experiment instead of once (or more) per trial.

Having the model running in an event-driven experiment with a single call to run typically allows for more interactive control of the task as a whole. The Stepper tool in the

[1] Hitting control-C or some other interrupt key in a Lisp running ACT-R may stop things, but if it's the ACT-R system code that is interrupted instead of the task code then it may cause problems for running ACT-R and require exiting and restarting.

Environment will also pause on user scheduled events and the Stop button will stop the whole experiment if it is being driven by the events that are run. That allows one to see exactly what is happening at specific points in the experiment without having to abort the experiment function. Additionally, to continue after stopping all one needs to do is then call run again to have the model and the experiment continue from where they left off since the events are still scheduled to occur at the appropriate times. It can also make writing the model itself easier because one doesn't have to make sure that the model "knows" when to stop for the task code to update and can just focus on having it respond to the events that occur as they occur instead of as a sequence of separate interactions.

The two models for this unit use those two different approaches. The paired associate task is written using the iterative approach, and the zbrodoff task is written as an event-driven experiment.

Paired associate task

Lisp

We start by loading the corresponding model for the task.

```
(load-act-r-model "ACT-R:tutorial;unit4;paired-model.lisp")
```

Then we define some global variables to hold the key pressed, the time of the response, the possible stimuli to use for the experiment, and the data from the original experiment for comparison.

```
(defvar *response* nil)
(defvar *response-time* nil)

(defvar *pairs* '(("bank" "0") ("card" "1") ("dart" "2") ("face" "3") ("game" "4")
                  ("hand" "5") ("jack" "6") ("king" "7") ("lamb" "8") ("mask" "9")
                  ("neck" "0") ("pipe" "1") ("quip" "2") ("rope" "3") ("sock" "4")
                  ("tent" "5") ("vent" "6") ("wall" "7") ("xray" "8") ("zinc" "9")))

(defvar *paired-latencies* '(0.0 2.158 1.967 1.762 1.680 1.552 1.467 1.402))
(defvar *paired-probability* '(0.000 .526 .667 .798 .887 .924 .958 .954))
```

The paired-task function takes two required parameters which are the number of pairs to present in a trial and the number of trials to run. It also takes an optional parameter which can be specified as true to run a person through the task instead of the model.

```
(defun paired-task (size trials &optional human)
```

It monitors output-key with the respond-to-key-press function to record the responses.

```
  (add-act-r-command "paired-response" 'respond-to-key-press
                     "Paired associate task key press response monitor")
  (monitor-act-r-command "output-key" "paired-response")
```

It calls do-experiment to run the actual experiment, removes the monitoring functions when it's done, and returns the result of the do-experiment call.

```
(prog1

    (do-experiment size trials human)

  (remove-act-r-command-monitor "output-key" "paired-response")
  (remove-act-r-command "paired-response")))
```

The respond-to-key-press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press (or nil if the key was pressed by a person interacting with the window) and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get-time function, and we can pass the model parameter to get-time to get the appropriate time since a model name is a true value which results in the ACT-R time being returned whereas if it is a person making the response the model value will be nil which means get-time will use the real clock instead of the ACT-R clock.

```
(defun respond-to-key-press (model key)

  (setf *response-time* (get-time model))
  (setf *response* key))
```

The do-experiment function takes three parameters which are the number of pairs to present in a trial, the number of trials to run, and whether it is a person or model doing the task. It runs the experiment for the size and number of trials requested and returns a list of lists. There is one sublist for each of the trials and they are in the order of presentation. Each of the sublists contains the percentage of answers correct and the average response time for the correct answers in the corresponding trial.

```
(defun do-experiment (size trials human)
```

First, if a person is doing the task make sure that there is a window available, otherwise print a warning and stop.

```
  (if (and human (not (visible-virtuals-available?)))
        (print-warning "Cannot run the task as a person without a visible window
available.")
```

To do the task, reset the ACT-R system and model to initial state.

```
    (progn

      (reset)
```

Create variables to hold the list of results, an indication of whether the model is doing the task, and the current task window which is opened immediately with a visible window for a person and virtual for the model.

```
      (let* ((result nil)
             (model (not human))
```

```
    (window (open-exp-window "Paired-Associate Experiment" :visible human)))
```

If the model is doing the task tell it to interact with that window.

```
(when model
  (install-device window))
```

Loop over the number of trials indicated.

```
(dotimes (i trials)
```

Create variables to hold the count and times for correct responses during this trial.

```
  (let ((score 0.0)
        (time 0.0))
```

Loop over the required number of pairs chosen randomly from *pairs*.

```
    (dolist (x (permute-list (subseq *pairs* (- 20 size))))
```

Clear the window and display the prompt.

```
      (clear-exp-window window)
      (add-text-to-exp-window window (first x) :x 150 :y 150)
```

Clear the *response* variable and record the current time.

```
      (setf *response* nil)
      (let ((start (get-time model)))
```

If it’s the model run it for exactly 5 seconds, or if it’s a person loop until 5000 milliseconds have passed making sure to call process-events in the loop.

```
        (if model
            (run-full-time 5)
          (while (< (- (get-time nil) start) 5000)
            (process-events)))
```

If the response is correct update the score and time values.

```
        (when (equal *response* (second x))
          (incf score 1.0)
          (incf time (- *response-time* start)))
```

Clear the window, display the associated number, and record the current time.

```
                (clear-exp-window window)
                (add-text-to-exp-window window (second x) :x 150 :y 150)
                (setf start (get-time model))
```

If it's the model run it for exactly 5 seconds, or if it's a person loop until 5000 milliseconds have passed making sure to call process-events in the loop.

```
                (if model
                    (run-full-time 5)
                  (while (< (- (get-time nil) start) 5000)
                    (process-events)))))
```

Save the proportion correct and the average time for a correct response.

```
             (push (list (/ score size)
                         (if (> score 0) (/ time score 1000.0) 0))
                   result)))
```

Return the list of trial results in order (reversed since the values were pushed onto the front as the trials progressed).

```
         (reverse result)))))
```

The paired-experiment function takes one parameter which is the number of times to repeat the full experiment (20 pairs and 8 trials). The experiment is run that many times and the results are averaged and compared to the experimental results.

```
(defun paired-experiment (n)
  (let ((data nil))
    (dotimes (i n)
      (if (null data)
          (setf data (paired-task 20 8))
        (setf data (mapcar (lambda (x y)
                             (list (+ (first x) (first y))
                                   (+ (second x) (second y))))
                     data (paired-task 20 8)))))
    (output-data data n)))
```

The output-data function takes two parameters. The first is a list of cumulative data from running multiple iterations of the experiment and the second parameter indicates how many repetitions were added into that cumulative data. It averages that data and then calls print-results to display the comparison and table for both the latency and accuracy data.

```
(defun output-data (data n)
  (print-results (mapcar (lambda (x) (/ (second x) n)) data)
                 *paired-latencies* "Latency")
  (print-results (mapcar (lambda (x) (/ (first x) n)) data)
                 *paired-probability* "Accuracy"))
```

The print-results function takes three parameters. The first is the list of data from running the experiment. The second is the experimental results for that data, and the third is the label to print when displaying that data. The new data is compared to the experimental results and printed along with a table of the new data.

```
(defun print-results (predicted data label)
 (format t "~%~%~A:~%" label)
  (correlation predicted data)
  (mean-deviation predicted data)
  (format t "Trial   1      2      3      4      5      6      7      8~%")
  (format t "     ~{~8,3f~}~%" predicted))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

Then it loads the corresponding model for the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit4;paired-model.lisp")
```

Define some global variables to hold the key pressed, the time of the response, the possible stimuli to use for the experiment, and the data from the original experiment for comparison.

```
response = False
response_time = False

pairs = list(zip(['bank','card','dart','face','game','hand','jack','king','lamb','mask',
                   'neck','pipe','quip','rope','sock','tent','vent','wall','xray','zinc']
,
                 ['0','1','2','3','4','5','6','7','8','9',
                  '0','1','2','3','4','5','6','7','8','9']))

latencies = [0.0, 2.158, 1.967, 1.762, 1.680, 1.552, 1.467, 1.402]
probabilities = [0.0, .526, .667, .798, .887, .924, .958, .954]
```

The task function takes two required parameters which are the number of pairs to present in a trial and the number of trials to run. It also takes an optional parameter which can be specified as true to run a person through the task instead of the model.

```
def task (size,trials,human=False):
```

It monitors output-key with the respond_to_key_press function to record the responses.

```
    actr.add_command("paired-response",respond_to_key_press,
                     "Paired associate task key press response monitor")
    actr.monitor_command("output-key","paired-response")
```

It calls do_experiment to run the actual experiment, removes the monitoring functions when it's done, and returns the result of the do_experiment call.

```
    result = do_experiment(size,trials,human)

    actr.remove_command_monitor("output-key","paired-response")
    actr.remove_command("paired-response")

    return result
```

The respond_to_key_press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press (or None if the key was pressed by a person interacting with the window) and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get_time function, and we can pass the model parameter to get_time to get the appropriate time since a model name is a true value which results in the ACT-R time being returned whereas if it is a person making the response the model value will be None which means get_time will use the real clock instead of the ACT-R clock.

```
def respond_to_key_press (model,key):
    global response,response_time

    response_time = actr.get_time(model)

    response = key
```

The do_experiment function takes three parameters which are the number of pairs to present in a trial, the number of trials to run, and whether it is a person or model doing the task. It runs the experiment for the size and number of trials requested and returns a list of tuples. There is one tuple for each of the trials and they are in the order of presentation. Each of the tuples contains the percentage of answers correct and the average response time for the correct answers in the corresponding trial.

```
def do_experiment(size, trials, human):
```

First, if a person is doing the task make sure that there is a window available, otherwise print a warning and stop.

```
    if human and not(actr.visible_virtuals_available()):
        actr.print_warning("Cannot run the task as a person without a visible window.")
    else:
```

To do the task, reset the ACT-R system and model to initial state.

```
      actr.reset()
```

Create variables to hold the list of results, an indication of whether the model is doing the task, and the current task window which is opened immediately with a visible window for a person and virtual for the model.

```
      result = []
      model = not(human)
      window = actr.open_exp_window("Paired-Associate Experiment", visible=human)
```

If the model is doing the task tell it to interact with that window.

```
if model:
    actr.install_device(window)
```

Loop over the number of trials indicated.

```
for i in range(trials):
```

Create variables to hold the count and times for correct responses.

```
    score = 0
    time = 0
```

Loop over the required number of randomly chosen pairs.

```
    for prompt,associate in actr.permute_list(pairs[20 - size:]):
```

Clear the window and display the prompt.

```
        actr.clear_exp_window(window)
        actr.add_text_to_exp_window (window, prompt, x=150 , y=150)
```

Clear the response variable and record the current time.

```
        global response
        response = ''
        start = actr.get_time(model)
```

If it's the model run it for exactly 5 seconds, or if it's a person loop until 5000 milliseconds have passed making sure to call process_events.

```
        if model:
            actr.run_full_time(5)
        else:
            while (actr.get_time(False) - start) < 5000:
                actr.process_events()
```

If the response is correct update the score and time values.

```
        if response == associate:
            score += 1
            time += response_time - start
```

Clear the window, display the associated number, and record the current time.

```
                    actr.clear_exp_window(window)
                    actr.add_text_to_exp_window (window, associate, x=150 , y=150)
                    start = actr.get_time(model)
```

If it's the model run it for exactly 5 seconds, or if it's a person loop until 5000 milliseconds have passed making sure to call process_events.

```
                    if model:
                        actr.run_full_time(5)
                    else:
                        while (actr.get_time(False) - start) < 5000:
                            actr.process_events()
```

Save the proportion correct and the average time for a correct response.

```
                if score > 0:
                    average_time = time / score / 1000.0
                else:
                    average_time = 0

                result.append((score/size,average_time))
```

Return the list of trial results.

```
        return result
```

The experiment function takes one parameter which is the number of times to repeat the full experiment (20 pairs and 8 trials). The experiment is run that many times and the results are collected and passed to output_data for comparison to the experimental results.

```
def experiment(n):

    for i in range(n):

        if i == 0:
            data = task(20,8)
        else:
            data = list(map(lambda x,y: (x[0] + y[0],x[1] + y[1]),
                               data,task(20,8)))

    output_data(data,n)
```

The output_data function takes two parameters. The first is a list of cumulative data from running multiple iterations of the experiment and the second parameter indicates how many repetitions were added into that cumulative data. It averages that data and then calls print_results to display the comparison and table for both the latency and accuracy data.

```
def output_data(data,n):

    print_results(list(map(lambda x: x[1]/n,data)),latencies,"Latency")
    print_results(list(map(lambda x: x[0]/n,data)),probabilities,"Accuracy")
```

The print_results function takes three parameters. The first is the list of data from running the experiment. The second is the experimental results for that data, and the third is the label to print when displaying that data. The new data is compared to the experimental results and printed along with a table of the new data.

```
def print_results(predicted,data,label):

    print()
    print(label)
    actr.correlation(predicted,data)
    actr.mean_deviation(predicted,data)
    print("Trial    1       2       3       4       5       6       7       8")
    print("     ",end='')
    for i in predicted:
        print('%8.3f' % i,end='')
    print()
```

New Commands

Run-full-time/run_full_time – this function takes one required parameter which is the time to run a model in seconds and an optional parameter to indicate whether to run the model in step with real time. The model will run until the requested amount of time passes whether or not there is something for the model to do i.e. it guarantees that the model will be advanced by the requested amount of time. If the optional parameter is provided and is a true value, then the model is advanced in step with real time instead of being allowed to run as fast as possible in its own simulated time, and if a number is provided instead of True then that sets the scale to use for advancing model time relative to real time.

print-warning and **print_warning** can be used for outputting information in the ACT-R warning trace. The Python function takes a single parameter which is a string and prints that as an ACT-R warning message. The Lisp version is a little more general and works similar to the Lisp format command. It takes a format string and then any number of parameters after that to be used in the format string.

Zbrodoff task

The **zbrodoff** task is written using the event-driven approach and to run the model through the task the code will only need to call run once regardless of how many trials are to be run. This task also differs from all of the previous ones because the data to be collected has to be organized by block and the addend condition. There are many ways one could approach that, and for this task we are going to keep the collection simple and just record all of the trial data as it happens and then process that collection of data at the end to sort it into the appropriate cases. Alternatively, we could record it into an array or other data structure as it happens, but this choice was made to keep the components that are relevant to the event-driven approach easier to follow since that is the important aspect of this task for modeling purposes.

Lisp

It starts by loading the initial model for the task which counts through the alphabet to solve all of the problems.

```
(load-act-r-model "ACT-R:tutorial;unit4;zbrodoff-model.lisp")
```

Then it defines some global variables to hold the trials to be presented, the results that have been collected, and the data from the original experiment.

```
(defvar *trials*)
(defvar *results*)

(defvar *zbrodoff-control-data* '(1.84 2.46 2.82 1.21 1.45 1.42 1.14 1.21 1.17))
```

Because the functions to run this task use an optional parameter to indicate whether or not to show the task window, to keep things easier to use, this experiment uses a global variable to indicate whether it is a person or model doing the task. If it is set to **nil** then it runs a person and if it is set to **t** it runs the model.

```
(defparameter *run-model* t)
```

Instead of using lists to represent the information in a trial and to describe a response we create a custom structure to hold that data in a more descriptive format. A trial consists of a block number, the number addend which indicates the condition, the text to display on the screen, the correct answer, whether the window should be visible, whether the answer was correct, the time the trial started, and the response time for the trial.

```
(defstruct trial block addend text answer visible correct start time)
```

The construct-trial function takes a block number, a list of the items which describe the problem to present, and an optional parameter to indicate whether or not the window should be visible. It creates and returns a trial structure containing the appropriate information.

```
(defun construct-trial (block problem &optional visible)
  (destructuring-bind (addend1 addend2 sum answer) problem
    (make-trial :block block
                :addend addend2
                :text (format nil "~a + ~a = ~a" addend1 addend2 sum)
                :answer answer
                :visible visible)))
```

The present-trial function takes one parameter which is a trial structure and an optional parameter which indicates whether or not to open a new window for this trial (since this task is running continuously it will run faster if it uses the same window repeatedly, but because the same code is used to run it for a variety of different situations it needs to know when to start over with a new display).

```
(defun present-trial (trial &optional (new-window t))
```

If a new window is indicated then it opens one setting the visible status of the window based on the setting in the trial structure provided, and if it is running the model it installs that window device. If a new window is not indicated then it clears the current experiment window.

```
(if new-window
    (let ((w (open-exp-window "Alpha-arithmetic Experiment"
                              :visible (trial-visible trial))))
      (when *run-model*
        (install-device w)))
  (clear-exp-window))
```

It then adds the text for this trial to the window and records the current time in the trial structure.

```
(add-text-to-exp-window nil (trial-text trial) :x 100 :y 150)

(setf (trial-start trial) (get-time *run-model*)))
```

The respond-to-key-press function will be set up to monitor the output-key actions, and thus will be called with two parameters when a key is pressed: the name of the model who pressed the key (or nil if it is a person) and the string naming the key that was pressed. Unlike the previous tasks, since this one is event-driven we will do more than just record the key and time in this function.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))
```

Remove the current trial from those being presented.

```
  (let ((trial (pop *trials*)))
```

Set the response time and correctness of the response in that trial structure.

```
    (setf (trial-time trial) (/ (- (get-time *run-model*) (trial-start trial)) 1000.0))
    (setf (trial-correct trial) (string-equal (trial-answer trial) key))
```

Store that trial on the list of results.

```
    (push trial *results*))
```

If there are anymore trials to present then present the first one on the list and indicate that a new window is not needed. This is what makes this code event-driven – the event of pressing a key directly causes the presentation of the next trial.

```
  (when *trials*
    (present-trial (first *trials*) nil)))
```

The collect-responses function takes one parameter which is the number of trials that are to be run.

```
(defun collect-responses (count)
```

The global list of results is cleared.

```
  (setf *results* nil)
```

The respond-to-key-press function is setup to monitor the output-key command.

```
  (add-act-r-command "zbrodoff-response" 'respond-to-key-press
                     "Zbrodoff task key press response monitor")
  (monitor-act-r-command "output-key" "zbrodoff-response")
```

The first trial is presented in a new window.

```
  (present-trial (first *trials*))
```

If the model is performing the task then it is run for up to 10 seconds times the number of trials that need to be collected (it is assumed that the model will respond in 10 seconds or less per trial on average).

```
  (if *run-model*
      (run (* 10 count))
```

If a person is performing the task and there is a visible window available wait for the appropriate number of responses to be recorded calling process-events in the loop.

```
    (if (visible-virtuals-available?)
        (while (< (length *results*) count)
          (process-events))))
```

Remove the output-key monitoring.

```
  (remove-act-r-command-monitor "output-key" "zbrodoff-response")
  (remove-act-r-command "zbrodoff-response"))
```

The zbrodoff-problem function takes four required parameters and one optional parameter. The four required parameters are the strings that represent the equation to present, for example “A” “2” and “C”, and the string indicating the correct response – either “k” for correct or “d” for incorrect. The optional parameter controls whether the trial is shown in a visible window or not and defaults to the negation of whether or not the model is doing the task i.e. by default if the model is doing the task visible is **nil** in which case the window will be virtual and if a person is doing the task the window will be shown. Providing a value of **t** for the (optional) fifth parameter will cause the window to be displayed while the model is doing the task.

```
(defun zbrodoff-problem (addend1 addend2 sum answer &optional (visible (not *run-model*)))
```

This is only one trial, so set the list of trials to a list of only that one trial.

```
  (setf *trials* (list (construct-trial 1 (list addend1 addend2 sum answer) visible)))
```

Call collect-responses to perform the task and then analyze the results.

```
  (collect-responses 1)
  (analyze-results))
```

The zbrodoff-set function takes one optional parameter which controls whether the window is shown or not, the same as the zbrodoff-problem function. It runs once through a random permutation of the set of equations where a set is two instances of each of the equations with the addends 2, 3, and 4 in each of the true and false conditions, which is a total of 24 problems.

```
(defun zbrodoff-set (&optional (visible (not *run-model*)))
  (setf *trials* (create-set 1 visible))
  (collect-responses 24)
  (analyze-results))
```

The zbrodoff-block function takes one optional parameter like the problem and set functions. It runs one block of the experiment, which is eight repetitions of the set of equations, or a total of 192 problems.

```
(defun zbrodoff-block (&optional (visible (not *run-model*)))
  (setf *trials* nil)
  (dotimes (i 8)
    (setf *trials* (append *trials* (create-set 1 visible))))
  (collect-responses 192)
  (analyze-results))
```

The zbrodoff-experiment function runs the whole experiment once, which is three full blocks, or a total of 576 trials. It takes two optional parameters. The first is to control whether the window is shown or not as with the previous functions. The other controls whether the analysis is printed. The default is to have the analysis printed. Note that this function also calls reset to return the model to its initial condition. It is the only function in the experiment to do so. The other functions allow the model to maintain the information it has gained (which is the new chunks and the history of their use).

```
(defun zbrodoff-experiment (&optional (visible (not *run-model*)) (show t))
  (reset)
  (setf *trials* nil)
  (dotimes (j 3)
    (dotimes (i 8)
      (setf *trials* (append *trials* (create-set (+ j 1) visible)))))
  (collect-responses 576)
  (analyze-results show))
```

The zbrodoff-compare function takes one parameter, which is the number of times to run the whole experiment. It runs that many times through the experiment collecting the data which is then averaged and compared to the original experiment's results.

```
(defun zbrodoff-compare (n)
  (let ((results nil))
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run, and collect the results in a list.

```
(dotimes (i n)
  (push (zbrodoff-experiment nil nil) results))
```

Compute the averages of the response times and the number of correct answers.

```
(let ((rts (mapcar (lambda (x) (/ x (length results)))
             (apply 'mapcar '+ (mapcar 'first results))))
      (counts (mapcar (lambda (x) (truncate x (length results)))
                (apply 'mapcar '+ (mapcar 'second results)))))
```

Display the data fit between the current run and the experimental data and print the table of the results.

```
  (correlation rts *zbrodoff-control-data*)
  (mean-deviation rts *zbrodoff-control-data*)

  (print-analysis rts counts '(1 2 3) '("2" "3" "4") '(64 64 64)))))
```

The analyze-results function takes one optional parameter which controls whether or not the print the table of results. It computes the average response time and number of correct responses in the *results* global variable as a function of the number of blocks presented and the numerical addend and returns a list of those two results.

```
(defun analyze-results (&optional (show t))
  (let* ((blocks (sort (remove-duplicates (mapcar 'trial-block *results*)) '<))
         (addends (sort (remove-duplicates (mapcar 'trial-addend *results*)
                                           :test 'string-equal)
                        'string<))
         (counts nil)
         (rts nil)
         (total-counts (mapcar (lambda (x)
                                 (/ (count x *results*
                                           :key 'trial-addend
                                           :test 'string=)
                                    (length blocks)))
                         addends)))

    (dolist (x blocks)
      (dolist (y addends)
        (let ((data (mapcar 'trial-time
                      (remove-if-not (lambda (z)
                                       (and (response-correct z)
                                            (string= y (response-addend z))
                                            (= x (response-block z))))
                                     *results*))))
          (push (length data) counts)
          (push (/ (apply '+ data) (max 1 (length data))) rts))))


    (when show
      (print-analysis (reverse rts) (reverse counts) blocks addends total-counts))

    (list (reverse rts) (reverse counts))))
```

The print-analysis function takes five parameters that describe a set of data for the task. The data is a list of response times and a list of corresponding correct answers. Then there are two lists that indicate the blocks and addend conditions represented in the data and finally a list of the total number of correct trials in each addend condition. Those data are then displayed in a table.

```
(defun print-analysis (rts counts blocks addends totals)
  (format t "~%        ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend totals)))
  (dotimes (block (length blocks))
    (format t "~%Block ~2d" (nth block blocks))
    (dotimes (addend (length addends))
      (format t " ~6,3f (~2d)" (nth (+ addend (* block (length addends))) rts)
        (nth (+ addend (* block (length addends))) counts))))
  (terpri))
```

The *data-set* variable is created to hold lists that represent all the problems to present in one set of the task.

```
(defvar *data-set* '(("a" "2" "c" "k")("d" "2" "f" "k")
                     ("b" "3" "e" "k")("e" "3" "h" "k")
                     ("c" "4" "g" "k")("f" "4" "j" "k")
                     ("a" "2" "d" "d")("d" "2" "g" "d")
                     ("b" "3" "f" "d")("e" "3" "i" "d")
                     ("c" "4" "h" "d")("f" "4" "k" "d")
                     ("a" "2" "c" "k")("d" "2" "f" "k")
                     ("b" "3" "e" "k")("e" "3" "h" "k")
                     ("c" "4" "g" "k")("f" "4" "j" "k")
                     ("a" "2" "d" "d")("d" "2" "g" "d")
                     ("b" "3" "f" "d")("e" "3" "i" "d")
                     ("c" "4" "h" "d")("f" "4" "k" "d")))
```

The create-set function takes two parameters. The first indicates a number to specify which block of the experiment is being performed and the second indicates whether or not the trials should be shown in a visible window. It returns a randomly ordered list of 24 trial structures that make up one set of the control condition of the experiment.

```
(defun create-set (block visible)
  (mapcar (lambda (x)
            (construct-trial block x visible))
    (permute-list *data-set*)))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

It loads the initial model for the task which counts through the alphabet to solve all of the problems.

```
actr.load_act_r_model("ACT-R:tutorial;unit4;zbrodoff-model.lisp")
```

Then it defines some global variables to hold the trials to be presented, the results that have been collected, and the data from the original experiment.

```
trials = []
results = []

control_data = [1.84, 2.46, 2.82, 1.21, 1.45, 1.42, 1.14, 1.21, 1.17]
```

Because the functions to run this task use an optional parameter to indicate whether or not to show the task window, to keep things easier to use, this experiment uses a global variable to indicate whether it is a person or model doing the task. If it is set to **False** then it runs a person and if it is set to **True** it runs the model.

```
run_model = True
```

Instead of using lists to represent the information in a trial and to describe a response we create a custom class to hold that data. A trial consists of a block number, the number addend which indicates the condition, the text to display on the screen, the correct answer, whether the window should be visible, whether the answer was correct, the time the trial started, and the response time for the trial.

```
class trial():
    def __init__(self,block,addend1,addend2,sum,answer,visible=None):
        self.block = block
        self.addend2 = addend2
        self.text = addend1 + " + " + addend2 + " = " + sum
        self.answer = answer.lower()
        if visible == None:
            self.visible = not(run_model)
        else:
            self.visible = visible
        self.correct = False
```

The present_trial function takes one parameter which is a trial object and an optional parameter which indicates whether or not to open a new window for this trial (since this task is running continuously it will run faster if it uses the same window repeatedly, but because the same code is used to run it for a variety of different situations it needs to know when to start over with a new display).

```
def present_trial(trial, new_window = True):
```

If a new window is indicated then it opens one setting the visible status of the window based on the setting in the trial structure provided, and if it is running the model it installs that window device. If a new window is not indicated then it clears the current experiment window.

```
    if new_window:
        w = actr.open_exp_window("Alpha-arithmetic Experiment", visible=trial.visible)
        if run_model:
          actr.install_device(w)
    else:
        actr.clear_exp_window()
```

It then adds the text for this trial to the window and records the current time in the trial structure.

```
actr.add_text_to_exp_window(None,trial.text, x=100, y=150)

trial.start = actr.get_time(run_model)
```

The respond_to_key_press function will be set up to monitor the output-key actions, and thus will be called with two parameters when a key is pressed: the name of the model who pressed the key (or None if it is a person) and the string naming the key that was pressed. Unlike the previous tasks, since this one is event-driven we will do more than just record the key and time in this function.

```
def respond_to_key_press (model,key):
    global trials,results
```

Set the response time and correctness of the response in the first trial.

```
    trials[0].time = (actr.get_time(run_model) - trials[0].start) / 1000.0

    if key.lower() == trials[0].answer :
        trials[0].correct = True
```

Store that trial on the list of results.

```
    results.append(trials[0])
```

Remove the current trial from those being presented.

```
    trials = trials[1:]
```

If there are anymore trials to present then present the first one on the list and indicate that a new window is not needed. This is what makes this code event-driven – the event of pressing a key directly causes the presentation of the next trial.

```
    if len(trials) > 0 :
        present_trial(trials[0],False)
```

The collect_responses function takes one parameter which is the number of trials that are to be run.

```
def collect_responses(count):
```

The global list of results is cleared.

```
    global results

    results = []
```

The respond_to_key_press function is setup to monitor the output-key command.

```
actr.add_command("zbrodoff-response", respond_to_key_press,
                 "Zbrodoff task key press response monitor")
actr.monitor_command("output-key","zbrodoff-response")
```

The first trial is presented in a new window.

```
present_trial(trials[0])
```

If the model is performing the task and there is a visible then it is run for up to 10 seconds times the number of trials that need to be collected (it is assumed that the model will respond in 10 seconds or less per trial on average).

```
if run_model :
    actr.run(10 * count)
```

If a person is performing the task and there is a visible window available then wait for the appropriate number of responses to be recorded calling process_events in the loop.

```
else:
    if actr.visible_virtuals_available():
        while len(results) < count:
            actr.process_events()
```

Remove the output-key monitoring.

```
actr.remove_command_monitor("output-key","zbrodoff-response")
actr.remove_command("zbrodoff-response")
```

The problem function takes four required parameters and one optional parameter. The four required parameters are the strings that represent the equation to present, for example ‘A’ ‘2’ and ‘C’, and the string indicating the correct response – either ‘k’ for correct or ‘d’ for incorrect. The optional parameter controls whether the trial is shown in a visible window or not and with the default value of **None** if the model is doing the task the window will be virtual and if a person is doing the task the window will be shown. Providing a value of **True** for the (optional) fifth parameter will cause the window to be displayed while the model is doing the task.

```
def problem(addend1,addend2,sum,answer,visible=None):
```

This is only one trial, so set the list of trials to a list of only that one trial.

```
    global trials
    trials = [trial(1,addend1,addend2,sum,answer,visible)]
```

Call collect_responses to perform the task and then analyze the results.

```
    collect_responses(1)
    return analyze_results()
```

The set function takes one optional parameter which controls whether the window is shown or not, the same as the problem function. It runs once through a random permutation of the set of equations where a set is two instances of each of the equations with the addends 2, 3, and 4 in each of the true and false conditions, which is a total of 24 problems.

```
def set(visible=None):

    global trials
    trials = create_set(1,visible)

    collect_responses(24)
    return analyze_results()
```

The block function takes one optional parameter like the problem and set functions. It runs one block of the experiment, which is eight repetitions of the set of equations, or a total of 192 problems.

```
def block(visible=None):
    global trials
    trials = []

    for i in range(8):
        trials = trials + create_set(1,visible)

    collect_responses(192)
    return analyze_results()
```

The experiment function runs the whole experiment once, which is three full blocks, or a total of 576 trials. It takes two optional parameters. The first is to control whether the window is shown or not as with the previous functions. The other controls whether the analysis is printed. The default is to have the analysis printed. Note that this function also calls reset to return the model to its initial condition. It is the only function in the experiment to do so. The other functions allow the model to maintain the information it has gained (which is the new chunks and the history of their use).

```
def experiment(visible=None,show=True):

    actr.reset()

    global trials
    trials = []

    for j in range(3):
        for i in range(8):
            trials = trials + create_set(j+1,visible)

    collect_responses(576)
    return analyze_results(show)
```

The compare function takes one parameter, which is the number of times to run the whole experiment. It runs that many times through the experiment collecting the data which is averaged and compared to the original experiment's results.

```
def compare(n):
```

Initialize some lists to hold the average data.

```
rts = [0,0,0,0,0,0,0,0,0]
counts = [0,0,0,0,0,0,0,0,0]
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run, and collect the results in the data lists.

```
for i in range(n):
    r,c = experiment(False,False)
    rts = list(map(lambda x,y: x + y,rts,r))
    counts = list(map(lambda x,y: x + y,counts,c))
```

Compute the averages of the response times and the counts.

```
rts = list(map(lambda x: x/n,rts))
counts = list(map(lambda x: x/n,counts))
```

Display the data fit between the current run and the experimental data and print the table of the results.

```
actr.correlation(rts,control_data)
actr.mean_deviation(rts,control_data)

print_analysis(rts,counts,[1,2,3],['2','3','4'], [192,192,192])
```

The analyze_results function takes one optional parameter which controls whether or not to print the table of results. It computes the average response time and number of correct responses in the results global variable as a function of the number of blocks presented and the numerical addend and returns a list of those two results.

```
def analyze_results(show=True):

    blocks = []
    addends = []
    data = dict()
    totals = dict()

    for i in results:
        if i.addend2 in totals:
            totals[i.addend2] += 1
        else:
            totals[i.addend2] = 1

        if i.correct:
            if (i.block,i.addend2) in data:
                data[(i.block,i.addend2)].append(i.time)
            else:
                data[(i.block,i.addend2)]=[i.time]
                if i.block not in blocks:
                    blocks.append(i.block)
                if i.addend2 not in addends:
                    addends.append(i.addend2)


    blocks.sort()
    addends.sort()
    rts =[]
    counts =[]
    for b in blocks:
        for a in addends:
            rts.append(sum(data[(b,a)]) / len(data[(b,a)]))
            counts.append(len(data[(b,a)]))

    if show:
        print_analysis(rts,counts,blocks,addends,[totals[i] for i in addends])

    return (rts, counts)
```

The print_analysis function takes five parameters that describe a set of data for the task. The data is a list of response times and a list of corresponding correct answers. Then there are two lists that indicate the blocks and addend conditions represented in the data and finally a list of the total number of correct trials in each addend condition over all blocks. Those data are then displayed in a table.

```
def print_analysis(rts,counts,blocks,addends,totals):

    print()
    print("         ", end="")
    for a,t in zip(addends,map(lambda x: x/len(blocks), totals)):
        print("%6s (%2d) " % (a, t),end="")
    print()
    for b in range(len(blocks)):
        print("Block  %d" % blocks[b],end="")
        for a in range(len(addends)):
            print(" %6.3f (%2d)" %
                 (rts[a+b*len(addends)],counts[a+b*len(addends)]),end="")
        print()
```

The data_set variable is created to hold lists that represent all the problems to present in one set of the task.

```
data_set = [["a","2","c","k"],["d","2","f","k"],
            ["b","3","e","k"],["e","3","h","k"],
            ["c","4","g","k"],["f","4","j","k"],
            ["a","2","d","d"],["d","2","g","d"],
            ["b","3","f","d"],["e","3","i","d"],
            ["c","4","h","d"],["f","4","k","d"],
            ["a","2","c","k"],["d","2","f","k"],
            ["b","3","e","k"],["e","3","h","k"],
            ["c","4","g","k"],["f","4","j","k"],
            ["a","2","d","d"],["d","2","g","d"],
            ["b","3","f","d"],["e","3","i","d"],
            ["c","4","h","d"],["f","4","k","d"]]
```

The create_set function takes two parameters. The first indicates a number to specify which block of the experiment is being performed and the second indicates whether or not the trials should be shown in a visible window. It returns a randomly ordered list of 24 trial objects that make up one set of the control condition of the experiment.

```
def create_set(block,visible):
    return list(map(lambda x: trial(block,*x,visible=visible),
                    actr.permute_list (data_set)))
```

AGI command defaults

Two of the commands which we have seen in previous units are used slightly differently in the zbrodoff experiment. Previously when clear-exp-window was used we passed it the window to clear, but here we did not. If there is only one window opened by the AGI then most of the commands will default to working with that window and it does not need to be provided. Thus, this experiment assumes that there is only one open window and doesn't pass one to the clear-exp-window command. Similarly, when using add-text-to-exp-window (and other similar functions for buttons and lines which will be used later in the tutorial) the first parameter can be specified as nil (Lisp) or None (Python) to indicate that the default window should be used instead of specifying one. If there are multiple windows open when a call is made that indicates using the default window it will result in a warning and nothing will happen.

Monitoring function notes

As discussed with the unit 2 code, functions that are called as monitors are evaluated in separate threads and may require additional protection on changes to items which are also accessed outside of that function. What wasn't mentioned at that time is that when ACT-R is running to generate those actions there is some protection because the actions performed in the model are not evaluated in parallel – the events are evaluated one at a time. Therefore, when the monitor for output-key is called because the model pressed a key, you do not need to worry about threading issues between the monitoring function and other functions which are called as events by ACT-R or the code which called run, but it is still a potential problem when a person is performing the task since the person could press the key in parallel with any of the code which is running the task while the monitor is active.

However, there is another issue to consider, and that is whether any functions that are used in the monitoring function have threading issues. In particular, when running a model you need to be careful about the ACT-R commands that are used since even though ACT-R will only evaluate one event at a time ACT-R itself is still "running". Most of the ACT-R commands presented in the tutorial are safe to use while ACT-R is running, in particular all the AGI commands for creating and manipulating windows are safe, but commands which run ACT-R cannot be used while it is already running and the commands for resetting and reloading a model cannot be used while it is still running. If you are using other ACT-R commands which are not described in the tutorial you will want to verify with the reference manual that they are safe for use while ACT-R is running, and if you're using the Lisp version it is strongly recommended that you not call any undocumented ACT-R functions since you won't know if they are safe to use.

The :ncnar Parameter

As was mentioned in the main text there is a new parameter being set in the models for this unit - :ncnar (normalize chunk names after run). This parameter toggles whether or not the system cleans up the references to merged chunks' names. If the parameter is set to **t**, which is the default, then the system will ensure that every slot of a chunk in the model which has a chunk as the value references the "true name" of the chunk in the slot i.e. the name of the original chunk in DM with which any copies have been merged. That operation can make debugging easier for the modeler because all the slot values will be consistent with the chunks shown to be in DM. However, if a model generates a lot of chunks and/or it makes many calls to one of the ACT-R commands to "run" the model it can take time to maintain that consistency. Thus it can be beneficial to turn this parameter off by setting it to **nil** when model debugging is complete and one just wants to collect the results or when the real time needed to run a model is important. For the models in the tutorial, leaving it enabled will typically not result in much of a run time increase (the paired model is the worst performer in this respect running around 10% slower with it enabled whereas the zbrodoff model shows effectively no difference since it does not generate a lot of chunks in DM and only involves a single call to run the model), but for tasks with more chunks in DM and/or more calls to run ACT-R one may find the savings from turning it off to be more significant.

Unit 5: Activation and Context

The goal of this unit is to introduce the components of the activation equation that reflect the context of a declarative memory retrieval.

5.1 Spreading Activation

The first context component we will consider is called spreading activation. The chunks in the buffers provide a context in which to perform a retrieval. Those chunks can spread activation to the chunks in declarative memory based on the contents of their slots. Those slot contents spread an amount of activation based on their relation to the other chunks, which we call their strength of association. This essentially results in increasing the activation of those chunks which are related to the current context.

The equation for the activation A_i of a chunk *i* including spreading activation is defined as:

$$A_i = B_i + \sum_k \sum_j W_{kj} S_{ji} + \varepsilon$$

Measures of Prior Learning, B_i: The base-level activation reflects the recency and frequency of practice of the chunk as described in the previous unit.

Across all buffers: The elements *k* being summed over are the buffers which have been set to provide spreading activation.

Sources of Activation: The elements *j* being summed over are the chunks which are in the slots of the chunk in buffer *k*.

Weighting: W_{kj} is the amount of activation from source *j* in buffer *k*.

Strengths of Association: S_{ji} is the strength of association from source *j* to chunk *i*.

ε: The noise value as described in the last unit.

The weights of the activation spread, W_{kj}, default to an even distribution from each slot within a buffer. The total amount of source activation for a buffer will be called W_k and is settable for each buffer. The W_{kj} values are then set to W_k /n_k where n_k is the number of slots which contain chunks in the chunk in buffer *k*.

The strength of association, S_{ji}, between two chunks *j* and *i* is 0 if chunk *j* is not the value of a slot of chunk *i* and *j* and *i* are not the same chunk. Otherwise, it is set using this equation:

$$S_{ji} = S - \ln(fan_j)$$

S: The maximum associative strength (set with the :mas parameter)

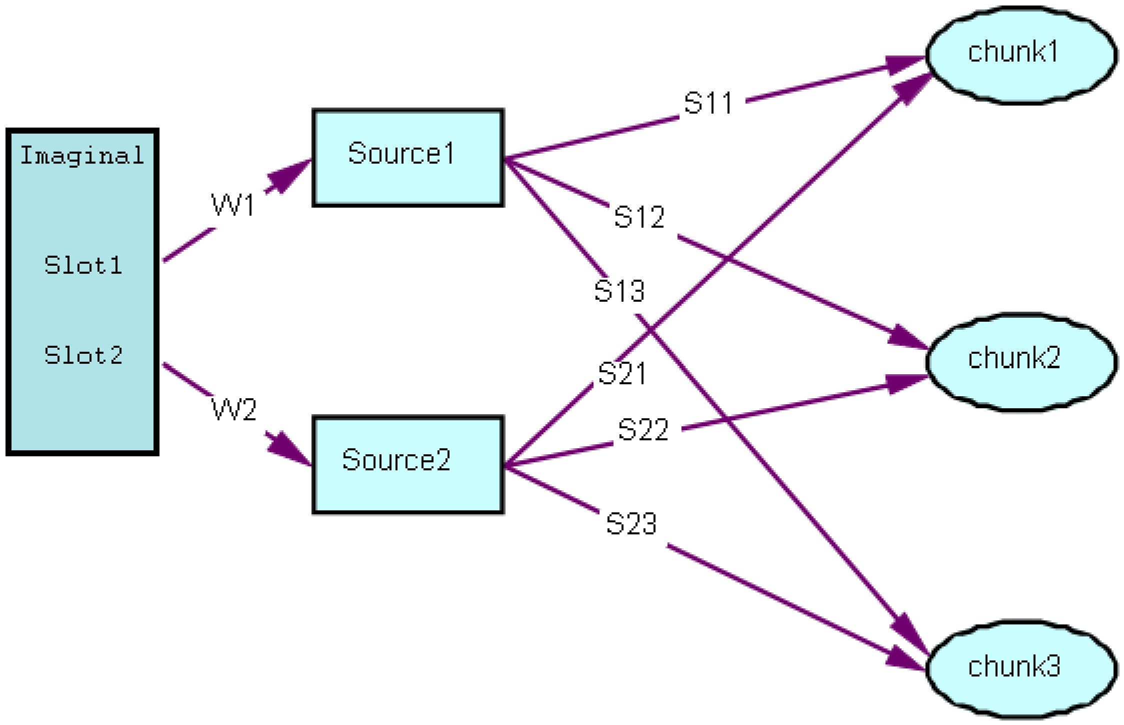
***fan$_j$*:** is the number of chunks in declarative memory in which *j* is the value of a slot plus one for chunk *j* being associated with itself. [That assumes the simple case where chunk *j* does not appear in more than one slot of any given chunk *i* which will be the case for the models in this unit. See the reference manual or the modeling text for this unit for the more general description.]

That is the general form of the spreading activation equation. However, by default, only the **imaginal** buffer serves as a source of activation. The ***W$_{imaginal}$*** value defaults to 1 (set with the :imaginal-activation parameter) and for all other buffers, ***W$_{buffer}$***, defaults to 0, but can be set to non-zero values with that buffer's spreading activation parameter. For the **goal** buffer that parameter is :ga and for all others it is :*<buffer>*-activation (where *<buffer>* is replaced with the actual name of the buffer for example :visual-activation for the **visual** buffer). Therefore, in the default case, the activation equation can be simplified to:

$$A_i = B_i + \sum_j W_j S_{ji} + \varepsilon$$

With W reflecting the value of the :imaginal-activation parameter and Wj being W/n where n is the number of chunks in slots of the current **imaginal** buffer chunk.

Here is a diagram to help you visualize how the spreading activation works. Consider an imaginal chunk which has two chunks in its slots when a retrieval is requested and that there are three chunks in declarative memory which match the retrieval request for which the activations need to be determined.



Each of the potential chunks also has a base-level activation which we will denote as B_i, and thus the total activation of the three chunks are:

$$A_1 = B_1 + W_1 S_{11} + W_2 S_{21}$$
$$A_2 = B_2 + W_1 S_{12} + W_2 S_{22}$$
$$A_3 = B_3 + W_1 S_{13} + W_2 S_{23}$$

and with the default value of 1.0 for :imaginal-activation W1 = W2 = 1/2.

There are two notes about using spreading activation. First, by default, spreading activation is disabled because :mas defaults to the value **nil**. In order to enable the spreading activation calculation :mas must be set to a positive value. The other thing to note is that there is no recommended value for the :mas parameter, but one almost always wants to set :mas high enough that all of the $\boldsymbol{S_{ji}}$ values are positive.

5.2 The Fan Effect

Anderson (1974) performed an experiment in which participants studied 26 facts such as the following sentences:

```
  1. A hippie is in the park.
  2. A hippie is in the church.
  3. A hippie is in the bank.
  4. A captain is in the park.
  5. A captain is in the cave.
  6. A debutante is in the bank.
  7. A fireman is in the park.
  8. A giant is in the beach.
  9. A giant is in the dungeon.
 10. A giant is in the castle.
 11. A earl is in the castle.
 12. A earl is in the forest.
 13. A lawyer is in the store.
 ...
```

After studying these facts, they had to judge whether they saw facts such as the following:

```
A hippie is in the park.
A hippie is in the cave.
A lawyer is in the store.
A lawyer is in the park.
A debutante is in the bank.
A debutante is in the cave.
A captain is in the bank.
```

which contained both studied sentences (targets) and new sentences (foils).

The people and locations for the study sentences could occur in any of one, two, or three of the study sentences. That is called their fan. The following tables show the recognition latencies from the experiment in seconds for targets and foils as a function of person fan and location fan:

	Targets				Foils			
Location	Person Fan				Person Fan			
Fan	1	2	3	Mean	1	2	3	Mean
1	1.111	1.174	1.222	1.169	1.197	1.221	1.264	1.227
2	1.167	1.198	1.222	1.196	1.250	1.356	1.291	1.299
3	1.153	1.233	1.357	1.248	1.262	1.471	1.465	1.399
Mean	1.144	1.202	1.357	1.20	1.236	1.349	1.340	1.308

The main effects in the data are that as the fan increases the time to respond increases and that the foil sentences take longer to respond to than the targets. We will now show how these effects can be modeled using spreading activation.

5.3 Fan Effect Model

There are two models for the fan effect included with this unit. Here we will work with the one found in the fan-model.lisp file in the unit 5 materials and the corresponding fan.lisp and fan.py experiment code files for presenting the testing phase of the experiment (the study portion of the task is not included for simplicity and the model already has chunks in declarative memory that encode all of the studied sentences). This version of the task uses the vision and motor modules to read the items and respond as we have seen in most of the tasks in the tutorial. The other version of the model and task work differently but produce the same results, and how it does that is discussed in the code document for this unit.

This model can perform one trial of the testing phase when run. Here is a trace of the model performing one trial for the target sentence "The lawyer is in the store":

```
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
0.000   VISION         visicon-update
0.000   PROCEDURAL     CONFLICT-RESOLUTION
0.050   PROCEDURAL     PRODUCTION-FIRED FIND-PERSON
0.050   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.050   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION         Find-location
0.050   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0
0.050   PROCEDURAL     CONFLICT-RESOLUTION
0.100   PROCEDURAL     PRODUCTION-FIRED ATTEND-VISUAL-LOCATION
0.100   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.100   PROCEDURAL     CLEAR-BUFFER VISUAL
0.100   PROCEDURAL     CONFLICT-RESOLUTION
0.185   VISION         Encoding-complete VISUAL-LOCATION0-1 NIL
0.185   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     CONFLICT-RESOLUTION
0.235   PROCEDURAL     PRODUCTION-FIRED RETRIEVE-MEANING
0.235   PROCEDURAL     CLEAR-BUFFER VISUAL
0.235   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.235   DECLARATIVE    start-retrieval
0.235   DECLARATIVE    RETRIEVED-CHUNK LAWYER
0.235   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL LAWYER
0.235   PROCEDURAL     CONFLICT-RESOLUTION
0.250   IMAGINAL       SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.250   PROCEDURAL     CONFLICT-RESOLUTION
0.300   PROCEDURAL     PRODUCTION-FIRED ENCODE-PERSON
0.300   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.300   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.300   VISION         Find-location
0.300   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
0.300   PROCEDURAL     CONFLICT-RESOLUTION
0.350   PROCEDURAL     PRODUCTION-FIRED ATTEND-VISUAL-LOCATION
0.350   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION
0.350   PROCEDURAL     CLEAR-BUFFER VISUAL
0.350   PROCEDURAL     CONFLICT-RESOLUTION
0.435   VISION         Encoding-complete VISUAL-LOCATION1-0 NIL
```

```
0.435   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
0.435   PROCEDURAL     CONFLICT-RESOLUTION
0.485   PROCEDURAL     PRODUCTION-FIRED RETRIEVE-MEANING
0.485   PROCEDURAL     CLEAR-BUFFER VISUAL
0.485   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.485   DECLARATIVE    start-retrieval
0.485   DECLARATIVE    RETRIEVED-CHUNK STORE
0.485   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL STORE
0.485   PROCEDURAL     CONFLICT-RESOLUTION
0.535   PROCEDURAL     PRODUCTION-FIRED ENCODE-LOCATION
0.535   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.535   PROCEDURAL     CONFLICT-RESOLUTION
0.585   PROCEDURAL     PRODUCTION-FIRED RETRIEVE-FROM-LOCATION
0.585   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.585   DECLARATIVE    start-retrieval
0.585   PROCEDURAL     CONFLICT-RESOLUTION
0.839   DECLARATIVE    RETRIEVED-CHUNK P13
0.839   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL P13
0.839   PROCEDURAL     CONFLICT-RESOLUTION
0.889   PROCEDURAL     PRODUCTION-FIRED YES
0.889   PROCEDURAL     CLEAR-BUFFER IMAGINAL
0.889   PROCEDURAL     CLEAR-BUFFER RETRIEVAL
0.889   PROCEDURAL     CLEAR-BUFFER MANUAL
0.889   MOTOR          PRESS-KEY KEY k
0.889   PROCEDURAL     CONFLICT-RESOLUTION
1.039   PROCEDURAL     CONFLICT-RESOLUTION
1.089   PROCEDURAL     CONFLICT-RESOLUTION
1.099   PROCEDURAL     CONFLICT-RESOLUTION
1.189   PROCEDURAL     CONFLICT-RESOLUTION
1.189   ------         Stopped because no events left to process
```

To run the model through one trial of the test phase you can use the function **fan-sentence** in the Lisp version or the function **sentence** in the fan module of the Python version. It takes four parameters. The first is a string of the person for the probe sentence. The second is a string of the location for the probe sentence. The third is whether the correct answer is true or false (t/nil in Lisp and True/False in Python), and the last is either person or location (as a symbol in Lisp and a string in Python) to choose which of the retrieval productions is used (more on that later). Here are examples which would produce the trace shown above:

```
? (fan-sentence "lawyer" "store" t 'person)
```

```
>>> fan.sentence('lawyer','store',True,'person')
```

The model can be run over each of the conditions to produce a data fit using the **fan-experiment** function in Lisp or the **experiment** function from the fan module in Python. Those functions take no parameters and will report the fit to the data along with the tables of response times and an indication of whether the answer provided was correct. [Note, you will probably want to set the :v parameter in the model to **nil** and reload it before running the whole experiment to disable the trace so that it runs must faster]:

```
CORRELATION:  0.864
MEAN DEVIATION:  0.053

TARGETS:
                         Person fan
  Location      1             2             3
    fan
     1       1.099 (T  )   1.157 (T  )   1.205 (T  )
     2       1.157 (T  )   1.227 (T  )   1.286 (T  )
     3       1.205 (T  )   1.286 (T  )   1.354 (T  )

FOILS:
     1       1.245 (T  )   1.290 (T  )   1.328 (T  )
     2       1.290 (T  )   1.335 (T  )   1.373 (T  )
     3       1.328 (T  )   1.373 (T  )   1.411 (T  )
```

Two ACT-R parameters were estimated to produce that fit to the data. They are the latency factor (:lf), which is the *F* in the retrieval latency equation from the last unit, set to .63 and the maximum associative strength (:mas), which is the *S* parameter in the S_{ji} equation above, set to 1.6. The spreading activation value for the **imaginal** buffer is set to the default value of 1.0. We will now look at how this model performs the task and how spreading activation leads to the effects in the data.

5.3.1 Model Representations

The study sentences are encoded in chunks placed into the model's declarative memory like this:

```
  (add-dm
   (p1 ISA comprehend-sentence relation in arg1 hippie arg2 park)
   (p2 ISA comprehend-sentence relation in arg1 hippie arg2 church)
   (p3 ISA comprehend-sentence relation in arg1 hippie arg2 bank)
   (p4 ISA comprehend-sentence relation in arg1 captain arg2 park)
   (p5 ISA comprehend-sentence relation in arg1 captain arg2 cave)
   (p6 ISA comprehend-sentence relation in arg1 debutante arg2 bank)
   (p7 ISA comprehend-sentence relation in arg1 fireman arg2 park)
   (p8 ISA comprehend-sentence relation in arg1 giant arg2 beach)
   (p9 ISA comprehend-sentence relation in arg1 giant arg2 castle)
   (p10 ISA comprehend-sentence relation in arg1 giant arg2 dungeon)
   (p11 ISA comprehend-sentence relation in arg1 earl arg2 castle)
   (p12 ISA comprehend-sentence relation in arg1 earl arg2 forest)
   (p13 ISA comprehend-sentence relation in arg1 lawyer arg2 store) ...)
```

They represent the items from the study portion of the experiment in the form of an association among the concepts e.g. p13 is encoding the sentence "The lawyer is in the store".

There are also meaning chunks which connect the text read from the display to the concepts. For instance, relevant to chunk p13 above we have:

```
   (lawyer ISA meaning word "lawyer")
   (store ISA meaning word "store")
```

The base-level activations of these meaning chunks have been set to 10 to reflect the fact that they are well practiced and should not fail to be retrieved, but the activations of the comprehend-sentence chunks are left at the default of 0 to reflect that they are relatively newer having only been learned during this experiment.

5.3.2 Perceptual Encoding

In this section we will briefly describe the productions that perform the perceptual parts of the task. This is similar to the steps that have been done in previous models and thus it should be familiar. One small difference is that this model does not use explicit state markers in the goal (in fact it does not place a chunk into the **goal** buffer at all) and instead relies on the states of the buffers and modules involved to constrain the ordering of the production firing.

Only the person and location are displayed for the model to perform the task. If the model were to read all of the words in the sentence it would be difficult to be able to respond fast enough to match the experimental data, and in fact studies of the fan effect done using an eye tracker verify that participants generally only fixate those two words from the sentences during the testing trials. Thus to keep the model simple only the critical words are shown on the display. To read and encode the words the model goes through a four step process.

The first production to fire issues a request to the **visual-location** buffer to find the person word and it also requests that the imaginal module create a new chunk to hold the sentence being read from the screen:

```
(P find-person
   ?visual-location>
      buffer      unrequested
   ?imaginal>
      state       free
   ==>
   +imaginal>

   +visual-location>
      ISA         visual-location
      screen-x    lowest)
```

The first query on the LHS of that production has not been used previously in the tutorial. The check that the **visual-location** buffer holds a chunk which was not requested is a way to test that a new display has been presented. The buffer stuffing mechanism will automatically place a chunk into the buffer if it is empty when the screen changes and because that chunk was not the result of a request it is tagged as unrequested. Thus, this production will match whenever the screen has recently changed if the **visual-location** buffer was empty at the time of the change and the **imaginal** module is not busy.

The next production to fire harvests the requested **visual-location** and requests a shift of attention to it:

```
(P attend-visual-location
   =visual-location>

   ?visual-location>
      buffer      requested
   ?visual>
      state       free
   ==>
   +visual>
      cmd         move-attention
      screen-pos  =visual-location)
```

Then the chunk in the **visual** buffer is harvested and a **retrieval** request is made to request the chunk that represents the meaning of that word:

```
(P retrieve-meaning
   =visual>
      ISA         visual-object
      value       =word
   ==>
   +retrieval>
      ISA         meaning
      word        =word)
```

Finally, the retrieved chunk is harvested and the meaning chunk is placed into a slot of the chunk in the **imaginal** buffer:

```
(P encode-person
   =retrieval>

   =imaginal>
      ISA         comprehend-sentence
      arg1        nil
   ==>
   =imaginal>
      arg1        =retrieval
   +visual-location>
      ISA         visual-location
      screen-x    highest)
```

This production also issues the **visual-location** request to find the location word and the same sequence of productions fire to attend and encode the location ending with the encode-location production firing instead of encode-person.

5.3.3 Determining the Response

After the encoding has happened the **imaginal** chunk will look like this for the sentence “The lawyer is in the store.”:

```
IMAGINAL: CHUNK0-0
CHUNK0-0
   ARG1  LAWYER
   ARG2  STORE
```

At that point one of these two productions will be selected and fired to retrieve a study sentence:

```
  (P retrieve-from-person
     =imaginal>
        ISA         comprehend-sentence
        arg1        =person
        arg2        =location
     ?retrieval>
        state       free
        buffer      empty
     ==>
     =imaginal>
     +retrieval>
        ISA         comprehend-sentence
        arg1        =person)

  (P retrieve-from-location
     =imaginal>
        ISA         comprehend-sentence
        arg1        =person
        arg2        =location
     ?retrieval>
        state       free
        buffer      empty
     ==>
     =imaginal>
     +retrieval>
        ISA         comprehend-sentence
        arg2        =location)
```

A thorough model of the task would have those two productions competing and one would randomly be selected. However, to simplify things for demonstration the experiment code which runs this task forces one or the other to be selected for each trial. The data is then averaged over two runs of each trial with one trial using retrieve-from-person and the other using retrieve-from-location.

One important thing to notice is that those productions request the retrieval of a studied chunk based only on one of the items from the probe sentence. By doing so it ensures that one of the study sentences will be retrieved instead of a complete failure in the event of a foil. If retrieval failure were used by the model to detect the foils then there would be no difference in response times for the foil probes because the time to fail is based solely upon the retrieval threshold. However, the data clearly shows that the fan of the items affects the time to respond to both targets and foils.

After one of those productions fires a chunk representing a study trial will be retrieved and one of the following productions will fire to produce a response:

```
(P yes
   =imaginal>
      ISA         comprehend-sentence
      arg1        =person
      arg2        =location
   =retrieval>
      ISA         comprehend-sentence
      arg1        =person
      arg2        =location
   ?manual>
      state       free
   ==>
   +manual>
      cmd         press-key
      key         "k")


(P mismatch-person
   =imaginal>
      ISA         comprehend-sentence
      arg1        =person
      arg2        =location
   =retrieval>
      ISA         comprehend-sentence
    - arg1        =person
   ?manual>
```

```
        state       free
     ==>
     +manual>
        ISA         press-key
        key         "d")

  (P mismatch-location
     =imaginal>
        ISA         comprehend-sentence
        arg1        =person
        arg2        =location
     =retrieval>
        ISA         comprehend-sentence
      - arg2        =location
     ?manual>
        state       free
     ==>
     +manual>
        cmd         press-key
        key         "d")
```

If the retrieved sentence matches the probe then the model responds with the true response, “k”, and if either one of the components does not match then the model responds with “d”.

5.4 Analyzing the Retrieval of the Critical Study Chunk in the Fan model

The perceptual and encoding actions the model performs for this task have a cost of .585 seconds and the time to respond after retrieving a comprehend-sentence chunk is .260 seconds. Those times are constant across all trials. The difference in the conditions will result from the time it takes to retrieve the studied sentence. Recall from the last unit that the time to retrieve a chunk *i* is based on its activation and specified by the equation:

$$Time_i = Fe^{-A_i}$$

Thus, it is differences in the activations of the chunks representing the studied items which will result in the different times to respond to different trials.

The chunk in the **imaginal** buffer at the time of the retrieval (after either retrieve-from-person or retrieve-from-location fires) will look like this:

```
IMAGINAL: CHUNK0-0
```

```
CHUNK0-0
   ARG1  person
   ARG2  location
```

where *person* and *location* will be the chunks that represent the meanings for the particular probe being presented.

The retrieval request will look like this for the person:

```
+retrieval>
    arg1 person
```

or this for the location:

```
+retrieval>
    arg2 location
```

depending on which of the productions was chosen to perform the retrieval.

The important thing to note is that because the sources of activation in the buffer are the same for either retrieval request the spreading activation will not differ between the two cases. You might wonder then why we would need to have both options. That will be described in the detailed examples below.

5.4.1 A note on chunks in buffers and the :dcnn parameter

Something that has been mentioned before is that buffers hold copies of chunks. A side effect of that is that when the name of that chunk is used (as is done with the retrievals in the encode-person and encode-location productions) it does not match the name of the original chunk. Thus, the chunk in the **imaginal** buffer will look like this after the encode-person production fires and the modification to the **imaginal** buffer occurs:

```
CHUNK0-0
   ARG1  LAWYER-0
```

because the copy of the lawyer chunk, named laywer-0, which was in the **retrieval** buffer was placed into the arg1 slot.

The lawyer-0 chunk is not modified by the model while it is in the **retrieval** buffer, and the **retrieval** buffer is also cleared by that production. Thus, once that clearing occurs the laywer-0 chunk is merged into declarative memory and because it is a perfect match to the lawyer chunk those two chunks are merged together so that the names lawyer-0 and lawyer refer to the same chunk. Both names may occur however in the slots of other chunks.

As discussed in the previous unit, the :ncnar parameter can be used to have the system normalize such references to make it easier to debug the system. If :ncnar is enabled (not **nil**) then the setting of the :dcnn (dynamic chunk name normalizing) parameter determines when those names are corrected. If :dcnn is set to **t**, which is the default value, then those changes are made immediately while the model runs. Since this model leaves the :ncnar and :dcnn parameters at their default values of **t**, the chunk in the **imaginal** buffer will be changed to look like this after the lawyer-0 and lawyer chunks are merged:

```
CHUNK0-0
   ARG1  LAWYER
```

If :dcnn were set to **nil**, then the arg1 slot would continue to hold the value lawyer-0 until the model stopped running at which time the chunk0-0 chunk would be updated if the :ncnar parameter were not **nil**. The important thing to note is that regardless of which name is shown in the slot, once those chunks have been merged the same chunk is being referenced whether a particular slot value is normalized or not.

Often having :dcnn and :ncnar set to **t** makes it easier to debug a model, but sometimes it may be useful to disable the dynamic updating so that one can more directly track a reference to a chunk from a buffer, even if it is eventually merged. For this unit the demonstration models leave :dcnn and :ncnar set to their default values of **t**, thus the slot values are dynamically adjusted as the model runs.

5.4.2 A simple target trial

The first case we will look at is the target sentence “The lawyer is in the store”. Both the person and location in this sentence have a fan of one in the experiment – they each only occur in that one study sentence.

The **imaginal** buffer’s chunk looks like this at the time of the critical retrieval (as discussed above):

```
CHUNK0-0
   ARG1  LAWYER
   ARG2  STORE
```

We will now look at the retrieval which results from the retrieve-from-person production firing. For the following traces we have enabled the activation trace parameter (:act) by setting it to **t**. That causes additional information to be displayed in the trace when a retrieval attempt is made. It shows all of the chunks that were attempted to be matched, and then for each that does match it shows all the details of the activation computation. Here is the trace of the model when that retrieval occurs:

```
0.585   DECLARATIVE    start-retrieval
Chunk P13 matches
Chunk P12 does not match
Chunk P11 does not match
Chunk P10 does not match
Chunk P9 does not match
Chunk P8 does not match
Chunk P7 does not match
Chunk P6 does not match
Chunk P5 does not match
Chunk P4 does not match
Chunk P3 does not match
Chunk P2 does not match
Chunk P1 does not match
Computing activation for chunk P13
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (STORE LAWYER)
    Spreading activation  0.45342642 from source STORE level  0.5 times Sji 0.90685284
    Spreading activation  0.45342642 from source LAWYER level  0.5 times Sji 0.90685284
Total spreading activation: 0.90685284
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P13 has an activation of: 0.90685284
Chunk P13 has the current best activation 0.90685284
Chunk P13 with activation 0.90685284 is the best
0.585   PROCEDURAL     CONFLICT-RESOLUTION
0.839   DECLARATIVE    RETRIEVED-CHUNK P13
```

In this case, the only chunk which matches the request is chunk p13. Note that this would look exactly the same if the retrieve-from-location production had fired because it would still be the only chunk that matched the request and the sources of activation are the same regardless of which one fires.

Remember that we have set the parameter *F* to .63, the parameter *S* to 1.6, and the base-level activation for the comprehend-sentence chunks is 0 in this model.

Looking at this trace, we see the S_{ji} values from store to p13 and lawyer to p13 are both approximately .907. That comes from the equation:

$$S_{ji} = S - \ln(fan_j)$$

The value of *S* was estimated to fit the data as 1.6 and the chunk fan of both the store and lawyer chunks is 2 (not the same as the fan from the experiment which is only one) because they each occur as a slot value in only the p13 chunk plus each chunk is always credited with a reference to itself. Then substituting into the equation we get:

$$S_{(store)(p13)} = S_{(lawyer)(p13)} = 1.6 - \ln(2) = 0.90685284$$

The W_j values (called the level in the activation trace) are .5 because the source activation from the **imaginal** buffer is the default value of 1.0 and there are two source chunks.

Thus the activation of chunk p13 is:

$$A_i = B_i + \sum_j W_j S_{ji}$$

$$A_{p13} = 0 + (.5 * .907) + (.5 * .907) = .907$$

Finally, we see the time to complete the retrieval (the time between the start-retrieval and the retrieved-chunk actions) is .254 seconds (.839- .585) which is determined from the retrieval time equation based on the chunk's activation:

$$Time_i = Fe^{-A_i}$$

$$Time_{p13} = .63e^{-.907} = 0.25435218$$

Adding that retrieval time to the fixed costs of .585 seconds to do the perception and encoding and the .26 seconds to perform the response gives us a total of 1.099 seconds, which is the value in the fan 1-1 cell of the model data for targets presented above.

Now that we have looked at the details of how the retrieval and total response times are determined for the simple case we will look at a few other cases.

5.4.3 A different target trial

The target sentence "The hippie is in the bank" is a more interesting case. Hippie is the person in three of the study sentences and bank is the location in two of them. Now we will see why it takes the model longer to respond to such a probe. Here are the critical components from the trace when retrieve-from-person is chosen:

```
Computing activation for chunk P3
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK HIPPIE)
    Spreading activation  0.25069386 from source BANK level  0.5 times Sji 0.5013877
    Spreading activation  0.10685283 from source HIPPIE level  0.5 times Sji 0.21370566
Total spreading activation: 0.3575467
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P3 has an activation of: 0.3575467
Chunk P3 has the current best activation 0.3575467
Computing activation for chunk P2
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK HIPPIE)
    Spreading activation  0.0 from source BANK level  0.5 times Sji 0.0
    Spreading activation  0.10685283 from source HIPPIE level  0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P2 has an activation of: 0.10685283
Computing activation for chunk P1
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK HIPPIE)
    Spreading activation  0.0 from source BANK level  0.5 times Sji 0.0
    Spreading activation  0.10685283 from source HIPPIE level  0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P1 has an activation of: 0.10685283
Chunk P3 with activation 0.3575467 is the best
```

There are three chunks that match the request for a chunk with an arg1 value of hippie. Each receives the same amount of activation being spread from hippie. Because hippie is a member of three chunks it has a chunk fan of 4 and thus the $S_{(hippie)i}$ value is:

$$S_{(hippie)i} = 1.6 - \ln(4) = 0.21370566$$

Chunk p3 also contains the chunk bank in its arg2 slot and thus receives the source spreading from it as well.

Now we will look at the case when retrieve-from-location fires for this probe sentence:

```
Computing activation for chunk P6
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK HIPPIE)
    Spreading activation  0.25069386 from source BANK level  0.5 times Sji 0.5013877
    Spreading activation  0.0 from source HIPPIE level  0.5 times Sji 0.0
Total spreading activation: 0.25069386
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P6 has an activation of: 0.25069386
Chunk P6 has the current best activation 0.25069386
Computing activation for chunk P3
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK HIPPIE)
    Spreading activation  0.25069386 from source BANK level  0.5 times Sji 0.5013877
    Spreading activation  0.10685283 from source HIPPIE level  0.5 times Sji 0.21370566
Total spreading activation: 0.3575467
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P3 has an activation of: 0.3575467
Chunk P3 is now the current best with activation 0.3575467
Chunk P3 with activation 0.3575467 is the best
```

In this case there are only two chunks which match the request for a chunk with an arg2 value of bank.

Regardless of which production fired to request the retrieval, chunk p3 had the highest activation because it received spreading activation from both sources. Thus, even if there is more than one chunk which matches the retrieval request issued by retrieve-from-person or retrieve-from-location the correct study sentence will always be retrieved because its activation will be the highest, and that activation value will be the same in both cases.

Notice that the activation of chunk p3 is less than the activation that chunk p13 had in the previous example because the source activation being spread to p3 is less. That is because the sources in that case have a higher fan, and thus a lesser S_{ji}. Because the activation is smaller, it takes longer to retrieve such a fact and that gives us the difference in response time effect of fan in the data.

5.4.4 A foil trial

Now we will look at a foil trial. The foil probe “The giant is in the bank” is similar to the target that we looked at in the last section. The person has an experimental fan of three and the location has an experimental fan of two. This time however there is no matching study sentence. Here are the critical components from the trace when retrieve-from-person is chosen:

```
Computing activation for chunk P10
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK GIANT)
    Spreading activation  0.0 from source BANK level  0.5 times Sji 0.0
    Spreading activation  0.10685283 from source GIANT level  0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P10 has an activation of: 0.10685283
Chunk P10 has the current best activation 0.10685283
Computing activation for chunk P9
Computing base-level
Starting with blc: 0.0
```

```
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK GIANT)
    Spreading activation  0.0 from source BANK level  0.5 times Sji 0.0
    Spreading activation  0.10685283 from source GIANT level  0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P9 has an activation of: 0.10685283
Chunk P9 matches the current best activation 0.10685283
Computing activation for chunk P8
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK GIANT)
    Spreading activation  0.0 from source BANK level  0.5 times Sji 0.0
    Spreading activation  0.10685283 from source GIANT level  0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P8 has an activation of: 0.10685283
Chunk P8 matches the current best activation 0.10685283
Chunk P10 chosen among the chunks with activation 0.10685283
```

There are three chunks that match the request for a chunk with an arg1 value of giant and each receives the same amount of activation being spread from giant. However, none contain an arg2 value of bank. Thus they only get activation spread from one source and have a lesser activation value than the corresponding target sentence had. Because the activation is smaller, the retrieval time is greater. This results in the effect of foil trials taking longer than target trials.

Before concluding this section however, let us look at the trace if retrieve-from-location were to fire for this foil:

```
Computing activation for chunk P6
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK GIANT)
    Spreading activation  0.25069386 from source BANK level  0.5 times Sji 0.5013877
    Spreading activation  0.0 from source GIANT level  0.5 times Sji 0.0
Total spreading activation: 0.25069386
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P6 has an activation of: 0.25069386
Chunk P6 has the current best activation 0.25069386
Computing activation for chunk P3
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (BANK GIANT)
    Spreading activation  0.25069386 from source BANK level  0.5 times Sji 0.5013877
    Spreading activation  0.0 from source GIANT level  0.5 times Sji 0.0
Total spreading activation: 0.25069386
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P3 has an activation of: 0.25069386
Chunk P3 matches the current best activation 0.25069386
Chunk P3 chosen among the chunks with activation 0.25069386
```

In this case there are only two chunks which match the request for a chunk with an arg2 value of bank. Again, the activation of the chunk retrieved is less than the corresponding target trial, but it is not the same as when retrieve-from-person fired. That is why the model is run with each of those productions fired once for each probe with the results being averaged together. Otherwise the foil data would only show the effect of fan for the item that was used to retrieve the study chunk.

5.5 Partial Matching

Up to now models have either always retrieved a chunk which matched the retrieval request or resulted in a failure to retrieve anything. Now we will look at modeling errors in recall in more detail. There are two kinds of errors that can occur. One is an error of commission when the wrong item is recalled. This will occur when the activation of the wrong chunk is greater than the activation of the correct chunk. The second is an error of omission when nothing is recalled. This will occur when no chunk has activation above the retrieval threshold.

We will continue to look at productions from the fan model for now. In particular, this production requests the retrieval of a chunk:

```
(P retrieve-from-person
     =imaginal>
        ISA         comprehend-sentence
        arg1        =person
        arg2        =location
     ?retrieval>
        state       free
        buffer      empty
     ==>
     =imaginal>
     +retrieval>
        ISA         comprehend-sentence
        arg1        =person)
```

In this case an attempt is being made to retrieve a chunk with a particular person (the value bound to =person) that had been studied. If =person were the chunk giant, this retrieval request would be looking for a chunk with giant in the arg1 slot. As was shown above, there were three chunks in the model from the study set which matched that request and one of those was retrieved.

However, let us consider the case where there had been no study sentences with the person giant but there had been a sentence with the person titan in the location being probed with giant i.e. there was a study sentence “The titan is in the bank” and the test sentence is now “The giant is in the bank”. In this situation one might expect that some human participants might incorrectly classify the probe sentence as one that was studied because of the similarity between the words giant and titan. The current model however could not make such an error.

Producing errors like that requires the use of the partial matching mechanism. When partial matching is enabled (by setting the :mp parameter to a number) the similarity between the chunks in the retrieval request and the chunks in the slots of the chunks in declarative memory are taken into consideration. The chunk with the highest activation is still the one retrieved, but with partial matching enabled that chunk might not have the exact slot values as specified in the retrieval request.

Adding the partial matching component into the activation equation, we now have the activation A_i of a chunk *i* defined fully as:

$$A_i = B_i + \sum_k \sum_j W_{kj} S_{ji} + \sum_l P M_{li} + \varepsilon$$

B_i, W_{kj}, S_{ji}, and ε have been discussed previously. The new term is the partial matching component.

***Specification elements l*:** The matching summation is computed over the slot values of the retrieval specification.

Match Scale, *P*: This reflects the amount of weighting given to the similarity in slot *l*. This is a constant across all slots and is set with the :mp parameter (it is also often referred to as the mismatch penalty).

Match Similarities, M_{li}: The similarity between the value *l* in the retrieval specification and the value in the corresponding slot of chunk *i*.

The similarity value, the M_{li}, can be set by the modeler along with the scale on which they are defined. The scale range is set with a maximum similarity (set using the :ms parameter) and a maximum difference (set using the :md parameter). By default, :ms is 0 and :md is -1.0. The similarity between anything and itself is automatically set to the maximum similarity and by default the similarity between any other pair of values is the maximum difference. Note that maximum similarity defaults to 0 and similarity values are actually negative. If a slot value matches the request then it does not penalize the activation, but if it mismatches then the activation is decreased. To demonstrate partial matching in use we will look at two example models.

5.6 Grouped Recall

The first of these models is called grouped and found in the gropued-model.lisp file with the unit 5 materials and the task is implemented in the grouped file for each language. This is a simple demonstration model of a grouped recall task which is based on a larger model of a complex recall experiment. As with the **fan** model, the studied items are already specified in the model, so it does not model the encoding and study of the items. In addition, the response times and error profiles of this model are not fit to any data. This demonstration model is designed to show the mechanism of partial matching and how it can lead to errors of commission and errors of omission. Because the model is not fit to any data, and the mechanism being studied does not rely on any of the perceptual or motor modules of ACT-R, they are not being used, and instead only a chunk in the **goal** buffer is used to hold both the task state and problem representation. This technique of using only the cognitive system in ACT-R can be useful when modeling a task where the timing is not important or other situations where accounting for “real world” interaction is not necessary to accomplish the objectives of the model. The experiment description text for this unit gives the details of how that is accomplished in this model and in an alternate version of the **fan** model which also does not use the perceptual and motor modules.

If you check the parameter settings for this model you will see that it has a value of .15 for the transient noise *s* parameter and a retrieval threshold of -.5. Also, to simplify the demonstration, the spreading activation described above is disabled by not providing a value for the :mas parameter. This model is set up to recall a list of nine items which are encoded in groups of three elements. The list that should be recalled is (123) (456) (789). To run the model, call the grouped-recall function in Lisp or the recall function from the grouped module in Python. That will print out the trace of the model doing the task and return a list of the model’s responses. Because the :seed parameter is set in the model you will always get the same result with the model recalling the sequence 1,2,3,4,6,5,7,8 (you can remove the setting of the :seed parameter to produce different results if you would like to explore the model further). It makes two errors in recalling that list, it mis-ordered the recall of the 5 and 6 and it failed to recall the last item, 9. We will now look at the details of how those errors happened.

5.6.1 Error of Commission

If one turns on the activation trace for this model you will again see the details of the activation computations taking place. The following is from the activation trace of the error of commission when the model recalls 6 in the second position of the second group instead of the correct item, 5. The critical comparison is between item5, which should be retrieved and item6, which is instead retrieved:

```
Computing activation for chunk ITEM5
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing partial matching component
  comparing slot GROUP
  Requested: = GROUP2  Chunk's slot value: GROUP2
  similarity: 0.0
  effective similarity value is 0.0
  comparing slot POSITION
  Requested: = SECOND  Chunk's slot value: SECOND
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.59634924
Adding permanent noise 0.0
Chunk ITEM5 has an activation of: -0.59634924
Chunk ITEM5 has the current best activation -0.59634924
Computing activation for chunk ITEM6
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing partial matching component
  comparing slot GROUP
  Requested: = GROUP2  Chunk's slot value: GROUP2
  similarity: 0.0
  effective similarity value is 0.0
  comparing slot POSITION
  Requested: = SECOND  Chunk's slot value: THIRD
  similarity: -0.5
  effective similarity value is -0.5
Total similarity score -0.5
Adding transient noise 0.11740411
Adding permanent noise 0.0
Chunk ITEM6 has an activation of: -0.3825959
Chunk ITEM6 is now the current best with activation -0.3825959
...
Chunk ITEM6 with activation -0.3825959 is the best
```

In these examples the base-level activations, B_i, have their default value of 0, the match scale, P, has the value 1, and the only noise value is the transient component with an s of 0.15. So the calculations are really just a matter of adding up the match similarities and adding the transient noise.

One thing to notice is that the :recently-retrieved request parameter is specified in all of the requests the model makes to retrieve the items, like this one:

```
    +retrieval>
       isa      item
       group    =group
       position second
       :recently-retrieved nil
```

Thus, only those chunks without a declarative finst are attempted for the matching. :recently-retrieved is not a slot of the chunk and thus does not undergo the partial matching calculation.

Looking at the matching of item5 above we see that it matches on both the group and position slots resulting in the addition of 0 to the base-level activation as a result of mismatch. It then receives an addition of about -0.596 in noise which is then its final activation value.

Next comes the matching of item6. The group slot matches the requested value of group2, but the position slots do not match. The requested value is second but item6 has a value of third. The similarity between second and third is set to -0.5 in the model, and that value is added to the activation. Then a transient noise of .117 is added to the activation for a total activation of -.383. This value is greater than the activation of item5 and thus because of random fluctuations item6 gets retrieved in error.

The similarities between the different positions are defined in the model using the set-similarities command:

```
(set-similarities
 (first second -0.5)
 (second third -0.5))
```

Similarity values are symmetric, thus it is not necessary to also specify (second first -0.5). The similarity between a chunk and itself has the value of maximum similarity by default therefore it is not necessary to specify (first first 0), and so on, for all the positions. Also, by default, different chunks have the maximum difference thus the similarity between first and third is -1 since it is not specified.

5.6.2 Error of Omission

Here is the portion of the detailed trace relevant to the failure to recall the ninth item:

```
Computing activation for chunk ITEM9
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing partial matching component
  comparing slot GROUP
  Requested: = GROUP3  Chunk's slot value: GROUP3
  similarity: 0.0
  effective similarity value is 0.0
  comparing slot POSITION
  Requested: = THIRD  Chunk's slot value: THIRD
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.5353896
Adding permanent noise 0.0
Chunk ITEM9 has an activation of: -0.5353896
Chunk ITEM9 has the current best activation -0.5353896
No chunk above the retrieval threshold: -0.5
```

We see that item9 starts out with an activation of 0 because it matches perfectly with the request and thus receives no penalty. However, it gets a transient noise of -.535 added to it which pushes its activation below the retrieval threshold and thus it cannot be retrieved. Because it is the only item chunk which is not marked as recently-retrieved it is the only one that can potentially be retrieved. Thus there are no chunks above the threshold and a retrieval failure occurs.

5.7 Simple Addition

The other example model for the unit which uses partial matching is fit to experimental data. The task is an experiment performed by Siegler and Shrager on the relative frequencies of different responses by 4 year olds to addition problems. The children were asked to recall the answers to simple addition problems without counting on their fingers or otherwise computing the answer. It seems likely that many of the kids did not know the answers to the larger problems that were tested. So we will only focus on the addition table from 1+1 to 3+3, and here are the data they reported:

	0	1	2	3	4	5	6	7	8	Other (includes no response)
1+1	-	.05	.86	-	.02	-	.02	-	-	.06
1+2	-	.04	.07	.75	.04	-	.02	-	-	.09
1+3	-	.02	-	.10	.75	.05	.01	.03	-	.06
2+2	.02	-	.04	.05	.80	.04	-	.05	-	-
2+3	-	-	.07	.09	.25	.45	.08	.01	.01	.06
3+3	.04	-	-	.05	.21	.09	.48	-	.02	.11

The **siegler** model is found in the siegler-model.lisp file of the unit and the code to perform the experiment is found in the siegler file of the code directories. As with the grouped task, there is no interface generated for the task, and thus it is not possible to run yourself through the experiment. That should not be too much of a problem however because one would guess that you would make very few errors if presented with such a task.

You can run the model through this task using the function called siegler-experiment in Lisp or experiment in the siegler module of Python. It requires one parameter which is how many times to present each of the problems, and it will report the results of those trials and the comparison to the data from the children. Since there is no learning involved with this experiment, for simplicity, the model is reset before each trial of the task. It is presented the numbers to add aurally and responds by speaking a number. Here is an example of the output:

```
CORRELATION:  0.966
MEAN DEVIATION:  0.054
       0     1     2     3     4     5     6     7     8   Other
1+1  0.01  0.12  0.74  0.11  0.01  0.00  0.00  0.00  0.00  0.02
1+2  0.00  0.00  0.14  0.70  0.12  0.00  0.00  0.00  0.00  0.03
1+3  0.00  0.00  0.00  0.15  0.76  0.04  0.00  0.00  0.00  0.04
2+2  0.00  0.00  0.01  0.13  0.79  0.03  0.00  0.00  0.00  0.04
2+3  0.00  0.00  0.00  0.02  0.25  0.51  0.07  0.01  0.00  0.14
```

```
3+3  0.00  0.00  0.00  0.00  0.01  0.09  0.64  0.07  0.00  0.18
```

Like the **fan** model, this model does not rely on the **goal** buffer at all for tracking its progress. The model builds up its representation of the problem in the **imaginal** buffer and relies on the module states and buffer contents to determine what needs to be done next. Most of the conditions and actions in the productions for this model are similar to those that have been used in other tutorial models. Thus, you should be able to understand and follow the basic operation of the model and we will not cover it in detail here. However, there are two productions which have actions that have not been used previously in the tutorial. We will describe those new actions and then look at how we determined the parameter settings used to match the data.

5.7.1 A Modification Request

This production in the siegler model has an action for the **imaginal** buffer which has not been discussed previously in the tutorial:

```
(p harvest-arg2
   =retrieval>
   =imaginal>
     isa          plus-fact
     addend2      nil
   ?imaginal>
     state        free
  ==>
   *imaginal>
     addend2      =retrieval)
```

An action for a buffer which begins with a * is called a modification request. It works similarly to a request which is specified with a + in that it is asking the buffer's module to perform some action which can vary from module to module. The modification request differs from the normal request in that it does not clear the buffer automatically in the process of making the request. Not every module supports modification requests, but both the goal and imaginal modules do and they both handle them in the same manner.

A modification request to the **goal** or **imaginal** buffer is a request for the module to modify the chunk that is in the buffer in the same way that a production would modify the chunk with an = action. With the goal module the only difference between an =goal action and a *goal action will be which module is credited with the action. Consider this production from the count model in unit1:

```
(p start
   =goal>
```

```
      ISA         count-from
      start       =num1
      count       nil
 ==>
   =goal>
      ISA         count-from
      count       =num1
   +retrieval>
      ISA         number
      number      =num1
   )
```

The =goal action in that production results in this output in the trace:

```
     0.050   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
```

Indicating that the procedural module modified the chunk in the **goal** buffer. If instead the start production used a *goal action like this:

```
(p start
   =goal>
      ISA         count-from
      start       =num1
      count       nil
 ==>
   *goal>
      ISA         count-from
      count       =num1
   +retrieval>
      ISA         number
      number      =num1
   )
```

Then the trace would look like this:

```
     0.050   PROCEDURAL             MODULE-MOD-REQUEST GOAL
     0.050   GOAL                   MOD-BUFFER-CHUNK GOAL
```

It shows that the procedural module made a modification request to the **goal** buffer and then the goal module actually performed the modification to the chunk in the buffer. That may not seem like an important distinction, but if one is trying to compare a model's actions to human brain activity then knowing where an action occurred is important.

For the imaginal module there is another difference between the =imaginal and the *imaginal actions. As we saw previously a request to create a chunk in the **imaginal** buffer has a time cost of 200ms. The same cost applies to modifications made to the chunk in the **imaginal** buffer by the imaginal module, whereas the production makes a modification immediately. Therefore if that start production were instead using the **imaginal** buffer (the **retrieval** request has also been removed since all that matters for this discussion is the modification of the **imaginal** buffer by this production):

```
(p start
   =imaginal>
      ISA         count-from
      start       =num1
      count       nil
 ==>
   =imaginal>
      ISA         count-from
      count       =num1
)
```

We would see this trace for the =imaginal action:

```
0.050   PROCEDURAL     PRODUCTION-FIRED START
0.050   PROCEDURAL     MOD-BUFFER-CHUNK IMAGINAL
```

However if the production used the *imaginal action (which should also include a query to test that the module is not busy as should be done for all requests):

```
(p start
   =imaginal>
      ISA         count-from
      start       =num1
      count       nil
   ?imaginal>
      state       free
 ==>
   *imaginal>
      ISA         count-from
      count       =num1
)
```

Then we would see this sequence of events in the trace:

```
0.050   PROCEDURAL     PRODUCTION-FIRED START
0.050   PROCEDURAL     MODULE-MOD-REQUEST IMAGINAL
```

```
0.250   IMAGINAL       MOD-BUFFER-CHUNK IMAGINAL
```

Which shows the 200ms cost before the imaginal module makes the modification to the chunk in the buffer.

The *imaginal action is the recommended way to make changes to the chunk in the **imaginal** buffer because it includes the time cost for the imaginal module to make the change. However if one is not as concerned about timing or the imaginal cost is not important for the task being modeled then the =imaginal actions can be used for simplicity as has been done up to this point in the tutorial.

5.7.2 An indirect request

This production in the siegler model has a **retrieval** request using syntax that has not been discussed previously in the tutorial:

```
(P harvest-answer
   =retrieval>
      ISA         plus-fact
      sum         =number
   =imaginal>
      isa         plus-fact
   ?imaginal>
      state       free
  ==>
   *imaginal>
      sum         =number
   +retrieval>    =number)
```

This production harvests the chunk in the **retrieval** buffer and copies the value from the sum slot of that chunk into the sum slot of the **imaginal** buffer and also makes what is called an indirect request to the **retrieval** buffer to retrieve the chunk which is contained in that slot. That chunk must be retrieved so that the value in its vocal-rep slot can be used to speak the response.

An indirect request can be made through any buffer by specifying a chunk or a variable bound to a chunk as the only component of the request. The actual request which is sent to the module in such a situation is constructed as if all of the slots and values of that chunk were specified explicitly. Thus, if in the production above =number were bound to the chunk eight from the model:

```
(eight ISA number aural-rep 8 vocal-rep "eight")
```

Then that retrieval request would be equivalent to this:

```
   +retrieval>
      aural-rep  8
      vocal-rep  "eight"
```

In fact, the module which receives the request will see it exactly like that – it has no access to the name of the chunk which was used to make the indirect request. Therefore, an indirect request will be handled by the module exactly the same way as a normal request.

In this case, since it is a **retrieval** request, it will undergo the same activation calculations and be subject to partial matching just like any other **retrieval** request. Thus an indirect request to the **retrieval** buffer is not guaranteed to put that chunk into the buffer. In this model, the correct chunk should always be retrieved because it will match on all of its slots and thus receive no penalty to its activation while all the other number chunks will receive twice the maximum difference penalty to their activation since they will mismatch on both slots and there are no similarities set in the model between numbers as used in the aural-rep slot or the strings used in the vocal-rep slot.

If one absolutely must place a copy of a specific chunk into a buffer in a production there is an action which will do that, but since that is not a recommended practice it will not be covered in the tutorial.

5.7.3 Parameters to be adjusted

To achieve that fit to the data we will be using partial matching and adjusting the base-level activations of the plus-facts for the model. We are not going to use spreading activation for this demonstration. However, if you would like to explore the effect it has on the model feel free to enable it and experiment with adjusting the source spread from the **imaginal** buffer which is holding the contextual information in this task.

The specific parameters that we will need to adjust for the model are those related to activation in general (the retrieval threshold, :rt, and the activation noise, :ans), those related to partial matching (the similarities among the number chunks used as the addends of the plus-facts and the match scale value, :mp), and the base-level activation values of the chunks.

That is potentially a lot of free parameters in the model. Treating them that way one could likely produce an extremely strong fit to the reported data. However, doing that is not very practical nor does it result in a model that is of much use for demonstrating anything other than the ability to fit 60 data points using more than 60 parameters.

In the following sections we will describe the effects that the particular parameters have on the model’s performance and outline an approach which can be taken to arrive at the parameter settings in a model.

5.7.4 Initial model

The first thing to do for the model is make sure that it can do the task. In this case that is hear the numbers, attempt to retrieve an addition fact, and then speak the result. To do that, we will start without enabling the subsymbolic components of the system. Making sure the model works right with basic symbolic information is a good start for modeling complex tasks because once the subsymbolic components are enabled and more sources of randomness or indeterminate behavior are introduced it can be very difficult to find potential errors in the productions or basic logic of the model.

The assumptions for the model are that the children know the numbers from zero through nine and that they have encountered the addition facts for problems with addends from zero to five. Thus these will be the declarative memory elements with which the model will start. Along with that, we are assuming that the children are not going to use any complex problem solving to try to remember the answers and that if there is a failure to remember a fact after one try the model will just give-up and answer that it does not know. For a task of this nature where we are modeling the aggregate data, using a single idealized strategy for the model is often a reasonable approach, and has been how all the other models seen so far in the tutorial operate. In other circumstances, particularly when individual participant data is being modeled, the specific strategy used to perform the task may be important, and in those cases it may be necessary to include different strategies into the model to account for the data.

With the model working correctly in a purely symbolic fashion we should see it answering correctly on every trial and here are the results of the model in that case providing the starting point for the adjustments to be made:

```
CORRELATION:  0.943
MEAN DEVIATION:  0.127
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
1+2  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00  0.00
1+3  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00
2+2  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00
2+3  0.00  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00
3+3  0.00  0.00  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00
```

5.7.5 Making errors

Now that we have a model which performs perfectly we need to consider how we want it to model the errors. For this task we have chosen to use partial matching to do that. Specifically, we want the model to retrieve an incorrect addition fact as it does the task and also to sometimes fail to retrieve an addition fact at all (an important source of the “other” results for the model). What we do not want it to do is retrieve an incorrect number chunk or fail to retrieve one while encoding the audio input or producing the vocal output. The reason for that is because we are assuming that the children know their numbers and thus do not produce errors because they are failing to understand what they hear or failing to say an answer correctly. That is important because we are not just looking to have the model fit the data but to actually have it do so in a manner which seems plausible for the task.

To make those errors through partial matching requires that the model occasionally retrieve the wrong chunk for the critical request which looks like this:

```
  +retrieval>
    ISA          plus-fact
    addend1      =val1
    addend2      =val2
```

where the =val1 and =val2 variables are bound to number chunks, for example one and three. Thus, the items which need to be similar are those number chunks which are the values requested in the retrieval, and that is where we will start in setting the parameters.

5.7.6 Setting similarities

The similarity settings between the number chunks will affect the distribution of incorrect retrievals. While this looks like a lot of free parameters to be fit, in practice that is just not reasonable. For a situation like this, where the chunks represent numbers, it is better to set the similarity between two numbers based on the numerical difference between them using a single formula to specify all of the similarities. There is a lot of research into how people rate the similarity of numbers and there are many equations which have been proposed to describe it. For this task, we are going to use a linear function of the difference between the numbers.

Also, to keep things simple we will use the default range of similarity values for the model, which are from 0.0 for most similar to -1.0 for most dissimilar. Since we are working with numbers from 0-9 an obvious choice for setting them seems to be:

$$Similarity(a,b) = -(0.1 * |a-b|)$$

To set those similarities, we need to use the set-similarities command. Because the similarities are symmetric we only need to set each pair of numbers once and we do not need to set the similarity between a chunk and itself because that defaults to the most similar value. We also note that since the model only has chunks for encoding the facts with addends from 0-5 we only need to set the similarities for the chunks which are relevant to the task. Thus, here are the initial similarity values set in the model:

```
(Set-similarities
  (zero one -0.1)  (one two -0.1)  (two three -0.1)(three four -0.1)(four five -0.1)
  (zero two -0.2)  (one three -0.2)(two four -0.2) (three five -0.2)
  (zero three -0.3)(one four -0.3) (two five -0.3)
  (zero four -0.4) (one five -0.4)
  (zero five -0.5))
```

In addition to the similarities, we will also need to set the match scale parameter for the model. Adjusting the match scale will determine how much the similarity values affect the activation of the chunks since it is used to multiply the similarity values. Because we have chosen a linear scale for our similarity values we will actually be able to just use the match scale parameter to handle all of our adjustments instead of needing to adjust the available range or the parameter we chose in our similarity equation.

The similarity value and the match scale are going to determine how close the activations between the correct and incorrect chunks are. How large that needs to be to create the effect we want is going to depend on other settings in the model. Thus, there is not really a good guideline for determining where it should be initially, but from experience we know that it is often easier to adjust the parameters later if we start with values that allow us to see the effect each has on the results. Therefore we want to make sure that we pick a value here which ensures that the similarity will make a difference in the activation values. Since the default base-level activation of chunks is 0.0 when the learning is off we are going to choose a large initial :mp value, like 5, to make sure that the activations will differ noticeably.

With just these settings however, the model will still not make any errors because the correct chunk will always have the highest activation and be the one retrieved. To actually get some errors we will need to also add some noise to the activation values.

5.7.7 Activation noise

In the previous unit, we saw how the activation noise affects the probability that a chunk will be above the retrieval threshold. Now, since there are multiple chunks which could all be above the threshold, it is also going to affect the frequency of retrieving the correct chunk among the incorrect alternatives. The more noise there is the less likely it is that the correct chunk will have the highest activation.

As with the :mp value, choosing the initial value for the noise is not obvious because its effect is determined by other settings in the model. For this parameter however, we do have some general guidelines to work with based on past experience. For many models that have been created in the past an activation noise value in the range of 0.0-1.0 has been a good setting and for most of those the value tends to fall somewhere between 0.2 and 0.5. So, based on that, we will start this model off with a value of .5, as was used for the models of the previous unit, and then adjust things from there if needed later.

Now, given these settings, :ans .5 and :mp 5 with the similarities set as shown above, we can run the model and see what happens. Here is what we see if we just run it to collect the data:

```
CORRELATION: -0.030
MEAN DEVIATION:  0.336
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.02  0.02  0.02  0.01  0.01  0.00  0.00  0.00  0.91
1+2  0.00  0.01  0.03  0.04  0.02  0.01  0.00  0.00  0.00  0.88
```

```
1+3  0.00  0.00  0.02  0.02  0.04  0.03  0.01  0.00  0.00  0.88
2+2  0.00  0.00  0.02  0.02  0.05  0.03  0.02  0.01  0.00  0.85
2+3  0.00  0.00  0.00  0.03  0.03  0.02  0.03  0.02  0.00  0.87
3+3  0.00  0.00  0.00  0.00  0.01  0.03  0.04  0.01  0.02  0.88
```

The model is almost never correct and most of the errors are in the other category which means that it probably did not respond. The important thing to do next is to understand why that is happening. One should not just start adjusting the parameters to try to improve the fit without understanding why the model is performing in that way.

5.7.8 Retrieval threshold and base-levels

Running the model on a few single trials and stepping through its operations shows that the problem is happening because the model is failing to retrieve chunks during all the retrieval requests, including the initial encoding of the numbers. We want the model to sometimes fail on the retrieval of the addition fact chunks, but we do not want it to be failing during the encoding steps.

So, there are two changes which we will make at this point. The first is to adjust the retrieval threshold so that we eliminate most, if not all, of the retrieval failures. This will allow us to work on setting the other parameters to match the data with the model answering the questions. Then we can come back to the retrieval threshold later and increase it to introduce more of the non-answer responses into the model. Thus, for now we will set the retrieval threshold to a value of -10.0 to make it very unlikely that any chunk will have an activation below the threshold.

The other thing we will do at this time is consider how to keep the number chunks from failing once we bring the retrieval threshold back up to a reasonable value. The easiest way to handle that is to increase the base-level activation of the number chunks so that the noise will be unlikely to ever take them below the retrieval threshold. The justification for doing so in the model is that it is assumed the children have a strong knowledge of the numbers and do not confuse or forget them and thus we need to provide the model with a comparable ability.

To do that we will use the set-base-levels command which works similar to the set-all-base-levels command that was used in the last unit. The difference is that for set-base-levels we can specify specific chunks instead of applying the change to all of them. Again, this seems like it is a lot of free parameters, but since we are not measuring the response time in this model all that matters is that the chunks have a value large enough to not fail to be retrieved – differences among them will not affect the error rate results as long as they are all being retrieved. We will start by assigning them a value of 10 which is significantly larger than the retrieval threshold we have now of -10 which should result in no failures for retrieving number chunks. When we increase the retrieval threshold later we may need to adjust this value, but for now we will add these settings to the model:

```
(set-base-levels
  (zero 10) (one 10) (two 10) (three 10) (four 10) (five 10)
  (six 10) (seven 10)(eight 10) (nine 10))
```

Unlike the similarities where we only needed to set the values for the numbers from 0-5 based on the task, here we need to set all of the numbers from 0-9 since any of those values is a potential sum of an addition fact in the model's declarative memory which may need to be retrieved.

After making those additions to the model, running it produces this output:

```
CORRELATION:  0.708
MEAN DEVIATION:  0.152
       0     1     2     3     4     5     6     7     8   Other
1+1  0.03  0.19  0.34  0.20  0.12  0.07  0.02  0.02  0.00  0.00
1+2  0.01  0.07  0.16  0.29  0.22  0.16  0.06  0.02  0.01  0.00
1+3  0.01  0.02  0.06  0.21  0.32  0.17  0.10  0.05  0.03  0.01
2+2  0.00  0.03  0.10  0.19  0.27  0.23  0.11  0.04  0.01  0.00
2+3  0.00  0.01  0.05  0.09  0.22  0.31  0.21  0.10  0.01  0.01
3+3  0.00  0.00  0.02  0.06  0.10  0.23  0.28  0.18  0.10  0.04
```

That shows a better fit to the data than the last one, though still not as good as we want, or in fact as good as it was when perfect. Looking at the trace of a few individual runs seems to indicate that the model is working as we would expect – the errors are only due to retrieving the wrong addition fact because of partial matching.

5.7.9 Adjusting the parameters

The next step to take depends on what the objectives of the modeling task are – what are you trying to accomplish with the model and what do you consider as a sufficient fit to the data. If that fit to the data is good enough, then as a next step you would then want to start bringing the retrieval threshold up to introduce more of the "other" responses (failure to respond) and hopefully improve things a little more. In this case however we are not going to consider that sufficient and will first investigate other settings for the :ans and :mp parameters before moving on to adjusting the retrieval threshold.

To do that we are going to search across those parameters for values which improve the model's fit to the data. When searching for parameters in a model there are a lot of approaches which can be taken. In this case, we are going to keep it simple and try manually adjusting the parameters and running the model to see if we can find some better values. When the number of parameters to search is small, the model runs fairly quickly, and one is not looking to precisely model every point this method can work reasonably well. For other tasks, which require longer runs or which have many more parameters to adjust other means may be required. That can involve writing some code to adjust the parameters and perform a more thorough search or going as far as creating an abstraction of the model based on the underlying equations and using a tool like MATLAB or Mathematica to solve those for the best values.

The approach that we use when searching by hand is to search on only one parameter at a time. Pick one parameter and then adjust that to get a better fit. Then, fix that value and pick another parameter to adjust. Do that until each of the parameters has been adjusted. Often, one pass through each of the parameters will result in a much better fit to the data, but sometimes it may require multiple passes to arrive at the performance level you desire (assuming of course that the model is capable of producing such a fit through manipulating the parameters).

Sometimes it is also helpful to work with only a subset of the parameters if you have an idea of the effects which they will have on the data. For example, in this task we know that the retrieval threshold will primarily determine the frequency with which the model gives up. Thus we are going to hold back on trying to fit that parameter until we have adjusted the others to better fit the majority of the data for the trials where it produces an answer.

Since we are starting with a noise value that was based on other tasks and our match scale value was chosen somewhat arbitrarily we will start searching across the match scale parameter. Keeping the noise value at .5 we found that a value of 16 for the match scale parameter seems to be our best fit:

```
CORRELATION:  0.942
MEAN DEVIATION:  0.074
       0     1     2     3     4     5     6     7     8   Other
1+1  0.01  0.13  0.75  0.10  0.01  0.00  0.00  0.00  0.00  0.00
1+2  0.00  0.00  0.12  0.74  0.13  0.01  0.00  0.00  0.00  0.00
1+3  0.00  0.00  0.01  0.13  0.75  0.10  0.01  0.00  0.00  0.00
2+2  0.00  0.00  0.01  0.11  0.75  0.12  0.01  0.00  0.00  0.00
2+3  0.00  0.00  0.00  0.00  0.14  0.73  0.13  0.00  0.00  0.00
3+3  0.00  0.00  0.00  0.00  0.01  0.13  0.76  0.10  0.01  0.00
```

Then, fixing the match scale parameter at 16 and adjusting the noise value we do not seem to find a value which does any better than the starting value of .5. So, we will adjust the retrieval threshold to introduce more of the other responses and hopefully improve the fit some more. Searching there finds that a value of .7 improves the fit to this:

```
CORRELATION:  0.949
MEAN DEVIATION:  0.065
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.07  0.74  0.09  0.01  0.00  0.00  0.00  0.00  0.09
1+2  0.00  0.00  0.10  0.70  0.08  0.00  0.00  0.00  0.00  0.11
1+3  0.00  0.00  0.00  0.10  0.71  0.09  0.01  0.00  0.00  0.08
2+2  0.00  0.00  0.01  0.12  0.66  0.12  0.00  0.00  0.00  0.08
2+3  0.00  0.00  0.00  0.00  0.13  0.68  0.09  0.00  0.00  0.10
3+3  0.00  0.00  0.00  0.00  0.01  0.08  0.70  0.08  0.00  0.13
```

One important thing to do at this point is to make sure that the model is still doing the task as we expect – that changing the parameters has not introduced some problems, like failing to retrieve the number chunks. For the current model, looking at a couple of single trial runs in detail shows that things are still working as expected. So, at this point we could go back and perform another pass through all the parameters trying to find a better fit, but instead we are going to stop and look at where our model seems to be deviating from the experimental data before trying to just find better parameters.

5.7.10 Adjusting the model

It seems that one trend in the data which we are missing is that the children seem to respond correctly more often to the smaller problems and that when they respond incorrectly the answers are more often smaller than the correct answer. There seems to be a bias for the smaller answers. This agrees with other research which finds that addition facts with smaller addends are encountered more frequently in the world.

Accounting for that component of the data is going to require making some adjustment to the model other than just modifying the parameters which we have. The research which finds that the smaller problems occur more frequently suggests a possible approach to take. The base-level activation of a chunk represents its history of use, and thus by increasing the base-level activation of the smaller plus-fact chunks we can simulate that increase in frequency and increase the probability that the model will retrieve them. This should help improve the data fit in a plausible manner.

Like the similarities, this is another instance where it looks like there are a lot of free parameters that could be used to fit the data, but again a principled approach is advised. In this case we are going to increase the base-level activation of all the small plus facts (which we have chosen to be those with a sum less than or equal to four), and we are going to give all of those chunks the same increase to their base-level activation. The default base-level activation for the plus-facts is 0. So we are going to set those chunks to have a value above that by using the set-base-levels command as we have done with the number chunks like this:

```
(set-base-levels
 (f00 .1)(f01 .1)(f02 .1)(f03 .1)(f04 .1)
 (f10 .1)(f11 .1)(f12 .1)(f13 .1)
 (f20 .1)(f21 .1)(f22 .1)
 (f30 .1)(f31 .1)
 (f40 .1))
```

Using the values for the other parameters found previously we will search for a base-level value which improves the data fit and what we find is that a value of .5 seems to improve things to this:

```
CORRELATION:  0.962
MEAN DEVIATION:  0.059
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.10  0.77  0.10  0.00  0.00  0.00  0.00  0.00  0.02
1+2  0.00  0.00  0.12  0.75  0.11  0.00  0.00  0.00  0.00  0.01
1+3  0.00  0.00  0.00  0.11  0.82  0.04  0.00  0.00  0.00  0.03
2+2  0.00  0.00  0.00  0.14  0.77  0.05  0.00  0.00  0.00  0.03
2+3  0.00  0.00  0.00  0.02  0.17  0.67  0.09  0.00  0.00  0.05
3+3  0.00  0.00  0.00  0.00  0.01  0.11  0.64  0.13  0.00  0.11
```

Given that, we will make one more pass over all the parameters (noise, match scale, retrieval threshold, and small plus-fact base-level offset) to find the final set of parameter values which are set in the given model and produce this fit to the data:

```
CORRELATION:  0.966
MEAN DEVIATION:  0.054
```

```
       0     1     2     3     4     5     6     7     8   Other
1+1  0.01  0.12  0.74  0.11  0.01  0.00  0.00  0.00  0.00  0.02
1+2  0.00  0.00  0.14  0.70  0.12  0.00  0.00  0.00  0.00  0.03
1+3  0.00  0.00  0.00  0.15  0.76  0.04  0.00  0.00  0.00  0.04
2+2  0.00  0.00  0.01  0.13  0.79  0.03  0.00  0.00  0.00  0.04
2+3  0.00  0.00  0.00  0.02  0.25  0.51  0.07  0.01  0.00  0.14
3+3  0.00  0.00  0.00  0.00  0.01  0.09  0.64  0.07  0.00  0.18
```

We could continue to search over the parameters or attempt other changes, like modifying the similarities used to something other than linear, but these results are sufficient for this demonstration. You are free to explore other changes to the parameters or the model if you are interested.

5.8 Learning from experience

The task for this assignment will be to create a model which can learn to perform a task better based on the experience it gains while doing the task. One way to do that is using declarative memory to retrieve a past experience which can be used to decide on an action to take. The complication however is that in many situations one may not have experienced exactly the same situation in the past. Thus, one will need to retrieve a similar experience to guide the current action, and the partial matching mechanism provides a model with a way to do that.

Instead of writing a model to fit data from an experiment, in this unit we will be writing a model which can perform a more general task. Specifically, the model must learn to play a game better. The model will be assumed to know the rules of the game, but will not have any initial experience with the game thus must learn the best actions to take as it plays. In the following sections we will introduce the rules of the game, how the model interacts with the game, a description of the starting model, what is expected of your model, and how to use the provided code to run the game.

5.8.1 1-hit Blackjack

The game we will be playing is a simplified version of the casino game Blackjack or Twenty-one. In our variant there are only two players and they each have only one decision to make.

The game is played with 2 decks of cards, one for each player, consisting of cards numbered 1-10. The number of cards in the decks and the distribution of the cards in the decks are not known to the players in advance. A game will consist of several hands. On each hand, the objective of the game is to collect cards whose sum is less than or equal to 21 and greater than the sum of the opponent's cards. When summing the values of the cards a 1 card may be counted as 11 if that sum is not greater than 21, otherwise it must be considered as only 1. At the start of a hand each player is dealt two cards. One of the cards is face up and the other is face down. A player can see both of his cards' values but only the value of the face up card of the opponent. Each player then decides if he would like one additional card or not. Choosing to take an additional card is referred to as a “hit” and choosing to not take a card is referred to as a “stay”. This choice is made without knowing your opponent’s choice – each player makes the choice in secret. An additional constraint is that the players must act quickly. The choice must be made within a preset time limit to prevent excessive calculation or contemplation of the actions and to keep the game moving. If a player hits then he is given one additional card from his deck and his final score is the sum of the three cards (with a 1 counted as 11 if that does not exceed 21). If a player stays then his score is the sum of the two starting cards (with a 1 counted as 11). Once any extra cards have been given both players show all of the cards in their hands and the outcome is determined. If a player’s total is greater than 21 then he has lost. That is referred to as “busting”. It is possible for both players to bust in the same hand. If only one player busts then the player who did not bust wins the hand. If neither player busts then the player with the greater total wins the hand. If the players have the same total then that is considered a loss for both players. Thus to win a hand a player must have a total less than or equal to 21 and either have a greater total than the opponent’s total or have the opponent bust. After a hand is over the cards are returned to the players’ decks, they are reshuffled and another hand begins. The objective of the game is to win as many hands as possible.

There are many unknown factors in this game making it difficult to know what the optimal strategy is at the start. However, over the course of many hands one should be able to improve their winnings as they acquire more information about the current game. One complication is that the opponent may also be adapting as the game goes on. To simplify things for this assignment we will assume that the model’s opponent always plays a fixed strategy, but that that strategy is not known to the model in advance. Thus, the model will start out without knowing the specifics of the game it is playing, but should still be able to learn and improve over time.

5.8.2 General modeling task description

To keep the focus of this modeling task on the learning aspect we have abstracted away from a real interface to the game in much the same way as the **fan** and **grouped** models abstracted away from a simulation of the complete experimental task. Thus the model will not have to use either the visual or aural module for acquiring the game state. Similarly, the model will also not have to compute the scores or determine the specific outcomes of each hand. The model will be provided with all of the available game state information in a chunk in the **goal** buffer at two points in the hand and will only need to make one of two key presses to signal its action.

At the start of the hand the model will be given its two starting cards, the sum of those cards, and the value of the opponent’s face up card. The model then must decide whether to hit or stay. The choice is made by pressing either the H key to hit or the S key to stay. The model has exactly 10 seconds in which to make this choice and if it does not press either key within that time it is considered as staying for the hand. After 10 seconds have passed, the model’s **goal** chunk will be modified to reflect the actions of both players and the outcome of the game. The model will then have all of its own cards’ values, all of the opponent’s cards’ values, the final totals for its hand and the opponent’s hand, as well as the outcome for each player. The model must then use that information to determine what, if anything, it should learn from this hand before the next hand begins. The time between the feedback and the next hand will also be 10 seconds.

5.8.3 Goal chunk specifics

Here is the chunk-type definition which specifies the slots used for the chunk that will be placed into the **goal** buffer:

```
(chunk-type game-state mc1 mc2 mc3 mstart mtot mresult
                       oc1 oc2 oc3 ostart otot oresult state)
```

The slots of the chunk in the **goal** buffer will be set by the game playing code for the model as follows:

- At the start of a new hand
    - **state** slot will be the value **start**
    - **mc1** slot will hold the value of the model's first card
        - a number from 1-10
    - **mc2** slot will hold the value of the model's second card
        - a number from 1-10
    - **mstart** slot will hold the score of the model's first two cards
        - a number from 4-21 because one 1 will be counted as an 11
    - **oc1** slot will hold the value of the opponent's face up card
        - a number from 1-10
    - **ostart** slot will hold the opponent's starting score
        - a number from 2-11 because a 1 will count as 11
    - none of the other slots will be set in the chunk
- After the 10 seconds for deciding have expired and players should have made a response
    - **state** slot will be set to the value **results**

- **mc1**, **mc2**, **mstart**, **oc1**, and **ostart** slots will be set to the same values as at the start of the hand described above
- **mc3**
  - if the model hits the slot will be the value of the model's third card
    - a number from 1-10
  - if the model stays the **mc3** slot will not be set
- **mtot** slot will hold the final total for the model's two or three card hand
  - a number from 4-30
- **mresult** slot will be the model's result for the hand
  - one of **win**, **lose**, or **bust**
- **oc2** will be the opponent's second card
  - a number from 1-10
- **oc3**
  - if the opponent hits the slot will be the value of the opponent's third card
    - a number from 1-10
  - if the opponent stays the **oc3** slot will not be set
- **otot** slot will be the final total for the opponent's two or three card hand
  - a number from 4-30
- **oresult** slot will hold the opponent's result for the hand
  - one of **win**, **lose**, or **bust**

For testing, the model will be played through a series of 100 hands and its percentage of winning in each group of 5 hands will be computed. For a fixed opponent's strategy and particular distribution of cards in the decks there is an optimal strategy and it may be possible to create rules which play a “perfect” game under those circumstances. However, since the model will not know that information in advance it will have to learn to play better, and the objective is to have a model which can improve its performance over time for a variety of different opponents and different possible decks of cards. Of course, since the cards received are likely random for any given sequence of 100 hands the model’s performance will vary and even a perfect strategy could lose all of them. Thus to determine the effectiveness of the model it will play several games of 100 hands and the results will be averaged to determine how well it is learning.

5.8.4 Starting model

A starting model for this task can be found in the 1hit-blackjack-model.lisp file with the unit 5 materials and the code for playing the game can be found in the onehit.lisp and onehit.py files. The given model uses a very simple approach to learn to play the game. It attempts to retrieve a chunk which contains an action to perform that is similar to the current hand from those which it created based on the feedback on previous hands. If it can retrieve such a chunk it performs the action that it contains, and if not it chooses to stay. Then, based on the feedback from the hand the model may create a new chunk which holds the learned information for this hand to use on future hands. As described below however, the feedback used by this model is not very helpful in producing a useful chunk for learning about the game – it learns a strategy of always hitting.

Here are the productions from the starting model:

```
(p start
   =goal>
     isa game-state
     state start
     MC1 =c
  ==>
   =goal>
     state retrieving
   +retrieval>
     isa learned-info
     MC1 =c
   - action nil)

(p cant-remember-game
   =goal>
     isa game-state
     state retrieving
   ?retrieval>
     buffer  failure
   ?manual>
     state free
  ==>
```

```
     =goal>
       state nil
     +manual>
       cmd press-key
       key "s")

  (p remember-game
     =goal>
       isa game-state
       state retrieving
     =retrieval>
       isa learned-info
       action =act
     ?manual>
       state free
    ==>
     =goal>
       state nil
     +manual>
       cmd press-key
       key =act

     @retrieval>)

  (p results-should-hit
     =goal>
       isa game-state
       state results
       mresult =outcome
       MC1 =c
     ?imaginal>
       state free
    ==>
     !output! (I =outcome)
     =goal>
       state nil
     +imaginal>
       MC1 =c
       action "h")

  (p results-should-stay
     =goal>
       isa game-state
       state results
       mresult =outcome
```

```
        MC1 =c
      ?imaginal>
        state free
     ==>
      !output! (I =outcome)
      =goal>
        state nil
      +imaginal>
        MC1 =c
        action "s")

   (p clear-new-imaginal-chunk
      ?imaginal>
        state free
        buffer full
      ==>
      -imaginal>)
```

The operation of most of those productions should be fairly straight forward, but there is something different in the action of the **remember-game** production which is worth noting. On its RHS we see this action:

```
     @retrieval>
```

The @ prefix is an action operator that has not been used in previous models of the tutorial. It is called the overwrite action and its purpose is to have the production modify the chunk in a buffer, much like the = operator. The difference between the overwrite and modification operators is that with the overwrite action only the slots and values specified in the overwrite action will remain in the chunk in the buffer – all other slots and values are erased. When it is used without any modifications, as is the case here, the buffer will be empty as a result of the action and the chunk which was there is not cleared and sent to declarative memory, it is simply erased from the buffer as if it did not exist.

That is done in this model to prevent that chunk from merging back into declarative memory and strengthening the chunk which was retrieved. The reason for that is because the chunk which was retrieved may not be the best action to take in the current situation either because it was retrieved due to noise or because the model does not yet have enough experience to accurately determine the best move. So, the model erases that information and waits for the feedback on the hand before creating a new chunk to represent the action to take on this hand. If it did not do that, then it could continue to strengthen and retrieve a chunk that makes a bad play just because it was created and retrieved often early on in its learning.

The overwrite action is not often used because typically one wants the model to accumulate the information it is using, and there are other ways to handle the situation of not reinforcing a memory that may not be useful. One would be to mark it in some way to indicate that it was a guess so that it did not reinforce the existing chunk, for example it could be modified like this:

```
    =retrieval>
       guess t
```

and then restrict the retrieval so that it avoids the previous guesses when trying to determine what to do:

```
    +retrieval>
       - guess t
       ...
```

Another alternative would be to just eliminate the slots that contain the critical information so that it cannot merge with the original chunk, which for the starting model would be:

```
    =retrieval>
       mc1 nil
       action nil
```

For this task however, to keep things simple and make it easier to look at declarative memory and see the chunks which the model is using we have chosen to use the overwrite action.

Because there are many more potential starting configurations than hands which will be played, this model uses the partial matching mechanism to allow it to retrieve a similar chunk when a chunk which matches exactly is not available. The given code provides the model with similarity values between numbers by using a function. This is done through the use of a “hook function” parameter in the model. A hook function parameter allows the modeler to override or modify an internal computation through code and there are several which can be specified for a model. In this case we are setting the :sim-hook parameter to compute the similarity values for the model. This is being done because the set-similarities command only allows the modeler to set the similarity value between chunks, but here we are using numbers to represent the card values and hand totals. Even if we had used chunks to represent the card values however it would still have been easier to use the hook function to compute the similarity values instead of having to explicitly specify all of the possible values for the similarities between the numbers which can occur while playing the game – essentially all the possible pairs for numbers from 1 to 30.

The equation that is used to set the similarities between the card values is:

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

This ratio has two features which should work well for this task and it corresponds to results found in the psychology literature. First, the similarity is relative to the difference between the numbers so that the closer the numbers are to each other the more similar they are. Thus, 1 and 2 are more similar than 1 and 3. The other feature is that larger numbers are more similar than smaller numbers for a given difference. Therefore 21 and 22 are more similar than 1 and 2 are.

There are several other parameters which are also set in the starting model. Those are divided into two sets. The first set is those which control how the model is configured (which learning mechanisms are enabled and how the system operates), and the second set is those which control the parameters of the mechanisms used in the model. This is the first set of parameters:

```
(sgp :esc t :bll .5 :ol t :sim-hook "1hit-bj-number-sims" :cache-sim-hook-results t :er t :lf 0)
```

It enables the subsymbolic components of ACT-R. It turns on the base-level learning for declarative memory with a decay of .5 (the recommended value) and specifies that the optimized learning equation be used for the base-level calculation. It specifies the name of the command which will compute the similarity values, and sets a flag to let the declarative module cache the similarity values returned by that function i.e. it will only call the function once to get the similarity for a given pair of numbers. Randomness is enabled to break ties for activations and during conflict resolution. Finally, the latency factor is set to 0 so that all retrievals complete immediately which is a simplification to avoid having to tune the model's chunk activation values to achieve appropriate timing since we are not matching latency data, but do have a time constraint of 10 seconds. Those settings should also be used in the assignment model which you write.

The other set of parameters in the starting model specifies the things which you may want to modify:

```
(sgp :v nil :ans .2 :mp 10.0 :rt -60)
```

In addition to the :v flag to control the trace it sets the activation noise to a reasonable value. The mismatch penalty for partial matching is set at 10 and the retrieval threshold is set very low so that the model should always be able to retrieve some relevant chunk if there are any. These values worked well for the solution model, but may need to be adjusted for your model.

5.8.5 The Assignment

The assignment for the task is to create a model which can learn to play better in a 100 hand game without knowing the details of the opponent or the distribution of cards in the decks in advance. Thus, it must learn based on the information it acquires as it plays the game.

Although the specific information learned by the starting model does not do a good job of learning to play better it does represent a reasonable approach for a model of this game. The recommended way to approach this assignment is to modify that starting model so that it learns to play better. You are not required to use that model, but your solution must use partial matching and it must be able to learn verses a variety of opponents and with different distributions of cards in the decks – it should not incorporate any information specific to the strategy of the default opponent provided or the default distribution of cards in the decks.

Here is a high level description of how this model plays the game in English from the model's perspective. At the start of the game, can I remember a similar situation and the action I took? If so, make that action. If not then I should stay. When I get the feedback is there some pattern in the cards, actions, and results which indicates the action I should have taken? If so create a memory of the situation for this game and the action to take. Otherwise, just wait for the next hand to start.

If you choose to use the starting model, then there are two things that you will need to change about it to make it learn to play the game better. One is the information which it considers when making its initial choice, and the other is how it interprets the feedback so that it creates chunks which have appropriate information about the actions it should take based on the information that is available.

Specifically, you will need to do the following things:

- Change the learned-info chunk-type to specify the slots which hold the information your model needs to describe the situation for the game (the current model only considers one of its own cards).

- Change the start production to retrieve a chunk given the information you have determined is appropriate.

- Change or replace the results-should-hit and results-should-stay productions to better test the information available at the end of the game to decide on a good action to learn (the current model does not require any particular pattern and just has two productions, one of which says it should hit regardless of the details and one which says stay regardless of the details). You may want to add more than two options as well because there are many reasonable ways to decide what move would have been "right", but you should probably start with a small set of simple choices and see how it works before trying to cover all possible options. If you have overlapping patterns you may want to set the utilities to provide a preference between them (the current model sets the results-should-hit production to have a higher utility than results-should-stay).

With an appropriate choice of initial information and some reasonable handling of the feedback that should be sufficient to produce a model which can learn against a variety of opponents.

There are other things which you could do that may improve that model's learning even more, and some of those are listed below. It is strongly recommended that you get a simple model which can learn to play the game using the basic operation described above before attempting to improve it with any of these or other mechanisms:

- Change the noise level and the mismatch penalty parameters to adjust the learning rate or flexibility of the model.

- Add more productions to analyze the starting position either before or after retrieving an appropriate chunk.

- Provide a strategy other than just choosing to stay when no relevant information can be retrieved.

The important thing to remember however is that the model should not make any assumptions about the distribution of cards in the decks or the strategy of the opponent.

5.8.6 Running the Game and Model

There are three functions that can be used to run a model through the game against an opponent implemented in the game code. Each is described along with examples below.

Playing a number of hands

To play some number of hands of the game the onhit-hands function in Lisp or the hands function in the onehit module of Python can be used. It takes one required parameter which is the number of hands to play and an optional parameter which controls whether it prints out the details of each of those hands. It returns a list of four items. The items in the list are the counts of the model's wins, the opponent's wins, the times when both players bust, and the times when they are tied.

Here is an example of running it in Lisp without the optional parameter:

```
? (onehit-hands 5)
(1 4 0 0)
```

Here is an example of running it in Python with the optional parameter to show the details of the hands played:

```
>>> onehit.hands(4,True)
Model: 10  5   -> 15 (lose)   Opponent: 10  2  4 -> 16 (win )
Model:  2  1  8  -> 21 (win )   Opponent:  1  7 -> 18 (lose)
Model:  8  9 10  -> 27 (bust)   Opponent:  8  2  6 -> 16 (win )
Model: 10  9  7  -> 26 (bust)   Opponent:  1 10 -> 21 (win )
[1, 3, 0, 0]
```

An important thing to note is that these functions do not reset the model. Thus it will retain any information which it has learned from one use to the next, and if you want to start the model over you will have to reset it explicitly.

Playing blocks of hands

The onehit-blocks function in Lisp and blocks function in the onehit module in Python take two parameters which are the number of blocks to run and the number of hands to run in each block. It runs the model through all of those hands and returns the list of results per block where each block is represented by a list as returned by the function for running hands shown above. Here is an example with running two blocks of 5 hands each in both Lisp and Python:

```
? (onehit-blocks 2 5)
((3 1 1 0) (2 3 0 0))

>>> onehit.blocks(2,5)
[[3, 2, 0, 0], [3, 2, 0, 0]]
```

Note that these functions also do not reset the model.

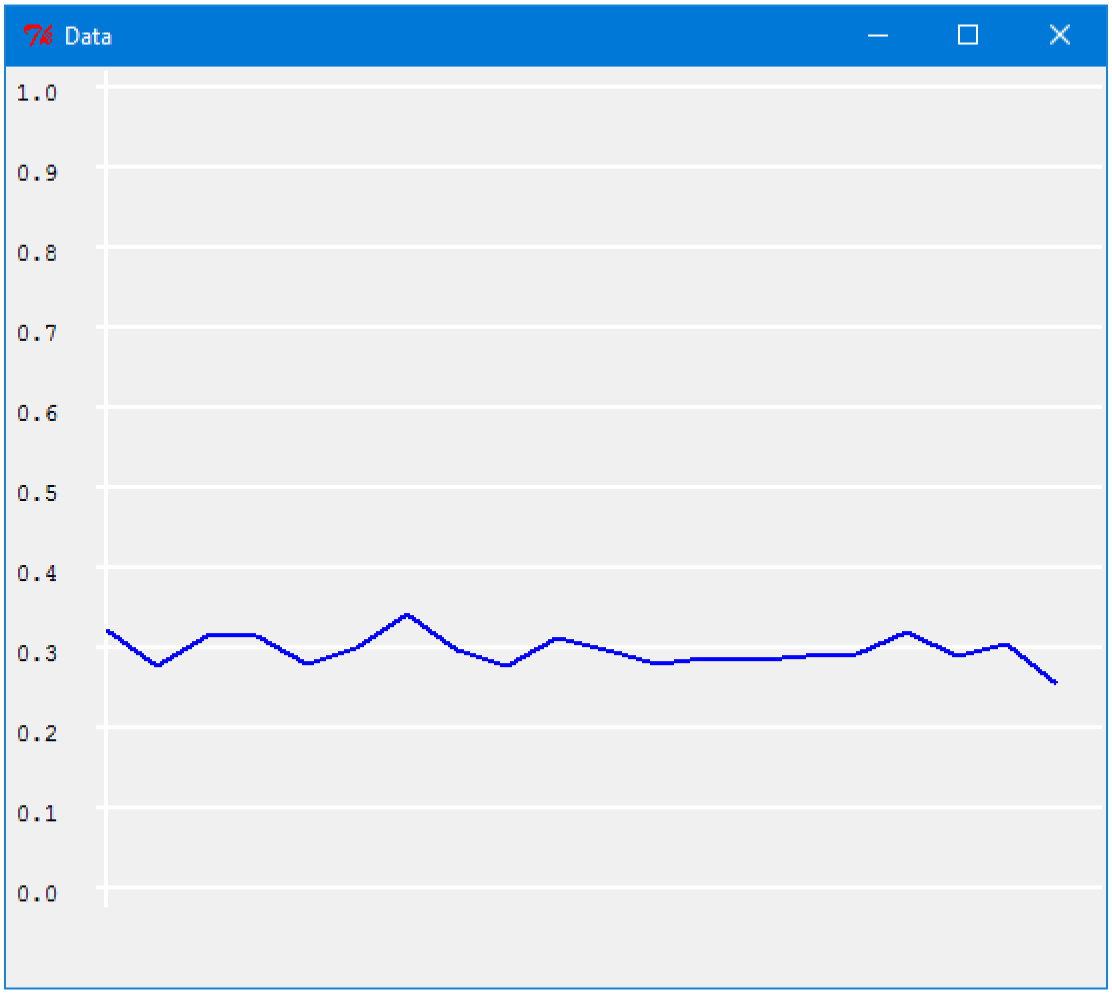
Showing learning over multiple runs of blocks of hands

The onehit-learning function in Lisp and learning function in the onehit module of Python are used to run a model through multiple 100 hand games for analysis. It takes one required parameter which is the number of games to run the model through and then average over. It also takes an optional parameter to indicate whether or not a graph of the results should be drawn and an optional parameter to indicate a function for specifying the game details. The model will be reset before running each 100 hand game. The results of those games are collected and averaged as both 4 blocks of 25 hands and as 20 blocks of 5 hands. It returns a list of two lists. The first list is the proportion of wins in the 4 blocks of 25. This list should give a quick indication of whether or not the model is improving over the course of the game. The second list is the proportion of wins considering 20 blocks of 5 hands to provide a more detailed description of the learning. If the first optional parameter is not given or is specified as a true value, then an experiment window will be opened and the detailed win data will be displayed in a graph. Here is an example call and resulting graph of a run of the example model:

```
? (onehit-learning 100)
```

```
((0.3012 0.3052 0.28679997 0.292)(0.322 0.276 0.314 0.314 0.28 0.3 0.342 0.296 0.276
0.31199998 0.296 0.28 0.28599998 0.28399998 0.28800002 0.292 0.32 0.29000002 0.304
0.254))
```

```
>>> onehit.learning(100)
[[0.3216, 0.28280000000000005, 0.3096, 0.2944], [0.284, 0.33799999999999997, 0.30
2, 0.33799999999999997, 0.346, 0.27599999999999997, 0.252, 0.298, 0.32, 0.268, 0.
284, 0.29, 0.34199999999999997, 0.312, 0.32, 0.292, 0.286, 0.302, 0.31, 0.282]]
```



If you do not want to see the graph then specifying the second parameter as a non-true value will suppress it:

```
? (onehit-learning 100 nil)
```

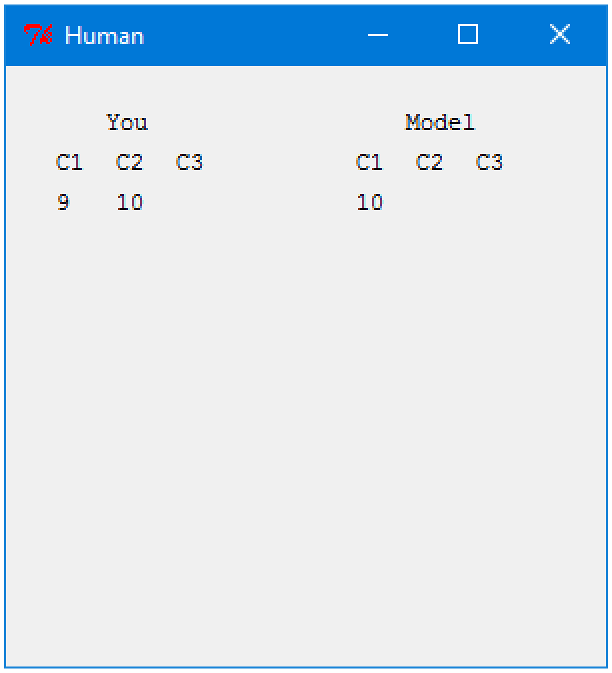
```
((0.2972 0.29839998 0.29520002 0.30479997) (0.344 0.28599998 0.302 0.276 0.278 0.278
0.268 0.348 0.292 0.306 0.29000002 0.276 0.306 0.28599998 0.31800002 0.314 0.294 0.28
0.324 0.31199998))
```

```
>>> onehit.learning(100,False)
[[0.3216, 0.28280000000000005, 0.3096, 0.2944], [0.284, 0.33799999999999997, 0.30
2, 0.33799999999999997, 0.346, 0.27599999999999997, 0.252, 0.298, 0.32, 0.268, 0.
284, 0.29, 0.34199999999999997, 0.312, 0.32, 0.292, 0.286, 0.302, 0.31, 0.282]]
```

The other optional parameter can be used to change the details of the decks of cards and the strategy for the opponent, and it is described in this unit's code description text.

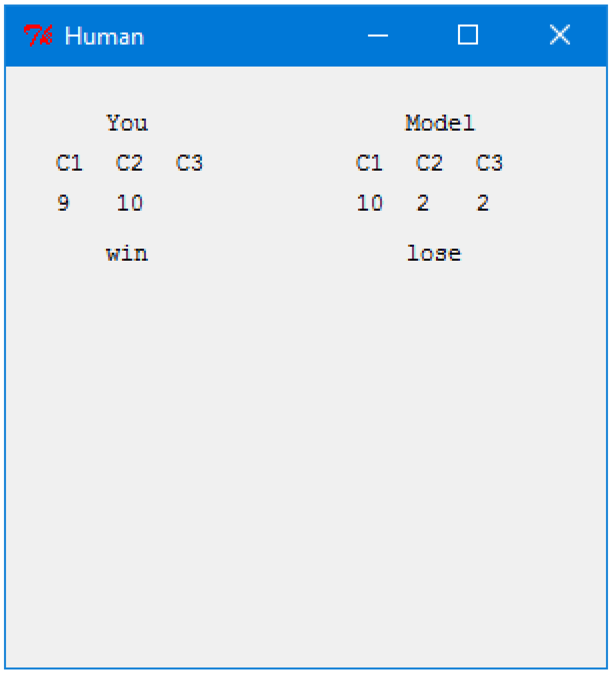
There is one other function which can be used to run the model called play-against-model in Lisp and play_against_model in the onehit module of Python. This function is similar to the one for playing hands except that instead of running an opponent from code it opens a window like that shown below which allows you to play against the model. It requires a number of hands to play as a parameter along with an optional parameter indicating whether or not to print the hand results.

At the start you will see your starting cards and the model's face up card for 10 seconds in which time you must respond by pressing h to hit or s to stay.



```
    You                 Model
C1   C2   C3        C1   C2   C3
9    10             10
```

After 10 seconds pass it will then show all of your cards and all of the model's cards along with the results for 10 seconds before going on to the next hand.

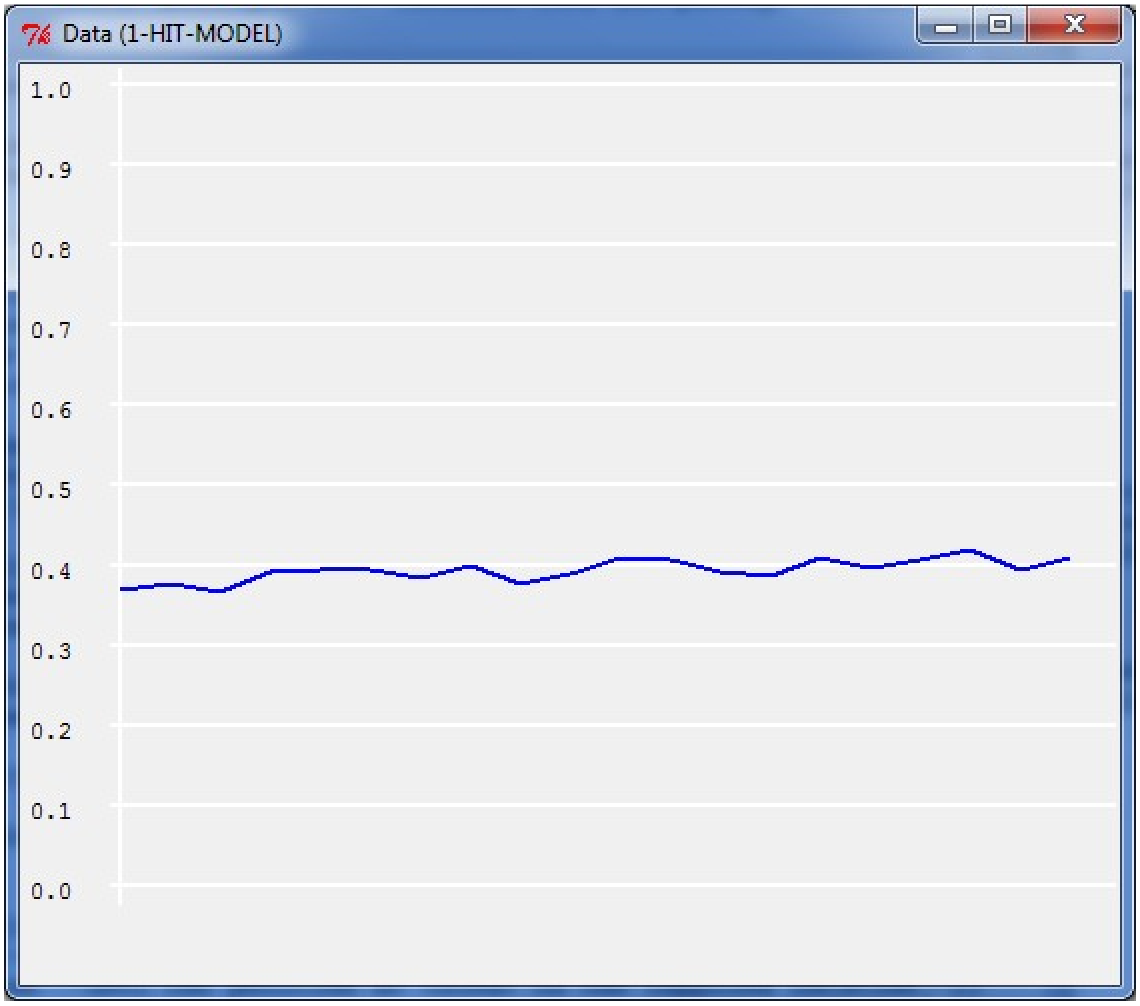


```
    You                 Model
C1   C2   C3        C1   C2   C3
9    10             10   2    2
    win                 lose
```

5.8.7 The Default Game

The default game your model will be playing is an opponent who always stays with a score of 15 or more and both decks have an effectively infinite number of cards in a distribution like a normal deck of playing cards (an equal distribution of cards with the values from 1-9 and four times as many cards with a value of 10). Under those circumstances the optimal strategy against that opponent would win about 46% of the time and choosing randomly wins about 32% of the time.

The reference solution model is able to improve from winning about 37% of the time in the first block to winning around 40% in the final block on average over the 100 hands as shown in this graph from running 500 games.



It is also able to learn against other opponents and when the distribution of cards in the deck differs. Your model should show similar or better performance for that default game and also still be able to learn in other situations i.e. just encoding an optimal strategy for the default opponent and deck distributions into your model is not an adequate solution to the task.

After producing a model which learns to play the default game you may want to try testing it with different opponents or other distributions of cards in the decks. Details on how to change the game code and some suggestions for other game situations are found in the code description text for this unit.

References

Anderson, J. R. (1974). Retrieval of propositional information from long-term memory. *Cognitive Psychology, 5*, 451 – 474.

Siegler, R. S., & Shrager, J. (1984). Strategy choices in addition and subtraction: How do children know what to do? *In C. Sophian (Ed.), Origins of cognitive skills (pp. 229-293)*. Hillsdale, NJ: Erlbaum.

Unit 5 Code Description

The experiments for this unit are a little more complex than the previous ones. They use a few new ACT-R commands for creating perceptual information as well as several new commands for interacting with the model more directly through code. First we will describe the code for implementing the tasks in this unit. Then we will describe the new commands, and that will include some discussion of some special issues with regard to providing parameter values for ACT-R commands. Finally, there is a section at the end of the document which describes how to add new game configurations to the 1-hit blackjack task along with some suggested alternate games for testing the assignment model.

Fan Experiment

There are two versions of the fan experiment code and model included with this unit. One pair, fan and fan-model which are used in the main unit text, uses an experiment window to present and perform the task as you have seen for most of the tasks in the tutorial. The other one runs the model through the task without using the visual interface for input or the motor module for output and produces exactly the same timing results. Instead it puts the input into the slots of the goal chunk before running the model and reads the response from a slot of the goal chunk upon completion. This is done through the use of commands to access the model's buffers and chunks. Mechanisms like these can be used when the details of the visual/motor systems are not of interest in the task being modeled and an abstraction of the experiment is acceptable for the objectives of the modeling work.

The code which implements the experiment window version of the task is very similar to many of the previous experiments and will not be described here. Instead, we will look at the code in the fan-no-pm.lisp and fan_no_pm.py files which perform the task without using an experiment window, and also look at some of the differences in the corresponding fan-no-pm-model.lisp file.

Lisp

First load the corresponding model file for this version of the task.

```
(load-act-r-model "ACT-R:tutorial;unit5;fan-no-pm-model.lisp")
```

Define a global variable that holds the experimental data to which the model will be compared.

```
(defvar *person-location-data* '(1.11 1.17 1.22
                                  1.17 1.20 1.22
                                  1.15 1.23 1.36
                                  1.20 1.22 1.26
                                  1.25 1.36 1.29
                                  1.26 1.47 1.47))
```

The fan-sentence function takes 4 parameters which are very similar to the ones that were passed to the fan-sentence function described in the main unit text, and for now we will ignore the difference but it will be described later. The first, person, is the string of the person named in the sentence. The second, location, is the string of the location named in the sentence. The third, target, is either t or nil to indicate whether this is a target or a foil trial respectively, and the fourth, term, specifies which of the productions to favor in retrieving the study item and should be either the symbol person or location. The function returns a list of two items. The first is the time the model spends running in seconds and the second item is t if the response was correct or nil if there was no response or the response was incorrect.

```
(defun fan-sentence (person location target term)
```

Start by resetting the model.

```
(reset)
```

Depending on which item should be used one of the retrieving productions is disabled using the new pdisable command.

```
(case term
  (person (pdisable retrieve-from-location))
  (location (pdisable retrieve-from-person)))
```

Instead of presenting the items visibly just modify the chunk which will be placed into the goal buffer when the model runs.

```
(mod-chunk-fct 'goal (list 'arg1 person 'arg2 location 'state 'test))
```

Run the model for up to 30 seconds recording how much time passed and get the model's response from the state slot of the chunk in the goal buffer. Using the value returned by run as a response time is not generally a useful approach because there are often actions which are executed at the end of a run which did not affect the time of the model's actual response – things like the motor module returning to the free state after an action, an unneeded retrieval request completing or failing, etc. This model was written so that it "stops" at the appropriate time for determining the response since it does not actually perform any actions, but that can be difficult to do in general and it is usually better to record the actual time of responses as has been used in most of the experiments in the tutorial.

```
(let ((response-time (run 30.0))
    (response (chunk-slot-value-fct (buffer-read 'goal) 'state)))
```

Return the list of the time and whether the correct answer was given.

```
(list response-time
      (or (and target (string-equal response "k"))
          (and (null target) (string-equal response "d"))))))
```

The do-person-location function takes one required parameter, term, which specifies which production to favor for the model as specified for fan-sentence. It iterates over all of the test sentences for the task collecting the data as returned by the fan-sentence function. It presents and returns a list of responses in the same order as the results in the global data list.

```
(defun do-person-location (term)
  (let ((test-set '(("lawyer" "store" t)("captain" "cave" t)("hippie" "church" t)
                    ("debutante" "bank" t)("earl" "castle" t)("hippie" "bank" t)
                    ("fireman" "park" t)("captain" "park" t)("hippie" "park" t)
                    ("fireman" "store" nil)("captain" "store" nil)
                    ("giant" "store" nil)("fireman" "bank" nil)
                    ("captain" "bank" nil)("giant" "bank" nil)
                    ("lawyer" "park" nil)("earl" "park" nil)
                    ("giant" "park" nil)))
        (results nil))

    (dolist (sentence test-set)
      (push (apply 'fan-sentence (append sentence (list term))) results))

    (reverse results)))
```

The fan-experiment function takes no parameters and will run the model through the experiment. It calls the do-person-location function once to favor the person and once to favor the location and prints out a table of the average response times and whether or not both responses were correct.

```
(defun fan-experiment ()
  (output-person-location (mapcar (lambda (x y)
                                    (list (/ (+ (car x) (car y)) 2.0)
                                          (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))
```

The output-person-location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that data to the experimental results and prints a table of the response times for targets and foils.

```
(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%                         Person fan~%")
    (format t  "  Location      1             2             3~%")
    (format t "    fan")

    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))

    (format t "~%~%FOILS:")
    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data))))))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

It loads the corresponding model file for this version of the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit5;fan-no-pm-model.lisp")
```

Define a global variable that holds the experimental data to which the model will be compared.

```
person_location_data = [1.11, 1.17, 1.22,
                        1.17, 1.20, 1.22,
                        1.15, 1.23, 1.36,
                        1.20, 1.22, 1.26,
                        1.25, 1.36, 1.29,
                        1.26, 1.47, 1.47]
```

The sentence function takes 4 parameters which are very similar to the ones that were passed to the sentence function described in the main unit text, and for now we will ignore the difference but it will be described later. The first, person, is the string of the person named in the sentence. The second, location, is the string of the location named in the sentence. The third, target, is to be either True or False to indicate whether this is a target or a foil trial respectively, and the fourth, term, specifies which of the productions to favor in retrieving the study item and should be either the string person or location. The function returns a tuple of two items. The first is the time that the model spent running in seconds and the second item is True if the response was correct or False if there was no response or the response was incorrect.

```
def sentence (person, location, target, term):
```

Start by resetting the model.

```
    actr.reset()
```

Depending on which item should be used one of the retrieving productions is disabled using the new pdisable command.

```
    if term == 'person':
        actr.pdisable("retrieve-from-location")
    else:
        actr.pdisable("retrieve-from-person")
```

Instead of presenting the items visibly just modify the chunk which will be placed into the goal buffer when the model runs.

```
    actr.mod_chunk("goal","arg1",person,"arg2",location,"state","test")
```

Run the model for up to 30 seconds recording how much time passed (the first value returned by run) and get the model's response from the state slot of the chunk in the goal buffer. Using the value returned by run as a response time is not generally a useful approach because there are often actions which are executed at the end of a run which did not affect the time of the model's actual response – things like the motor module returning to the free state after an action, an unneeded retrieval request completing or failing, etc. This model was written so that it "stops" at the appropriate time for determining the response since it does not actually perform any actions, but that can be difficult to do in general and it is usually better to record the actual time of responses as has been used in most of the experiments in the tutorial.

```
response_time = actr.run(30)[0]
response = actr.chunk_slot_value(actr.buffer_read("goal"),"state")
```

Return the list of the time and whether the correct answer was given.

```
if target:
    if response.lower() == "'k'".lower():
        return (response_time ,True)
    else:
        return (response_time ,False)
else:
    if response.lower() == "'d'".lower():
        return (response_time ,True)
    else:
        return (response_time ,False)
```

The do_person_location function takes one required parameter, term, which specifies which production to favor for the model as specified for the sentence function. It iterates over all of the test sentences for the task collecting the data as returned by the sentence function. It presents and returns a list of responses in the same order as the results in the global data list. You may notice something odd about the way the words are specified below and how the response was tested above. The reason for that will be described below with the new commands.

```
def do_person_location(term):

    data = []

    for person,location,target in [("'lawyer'", "'store'", True),
                                   ("'captain'", "'cave'", True),
                                   ("'hippie'", "'church'", True),
                                   ("'debutante'", "'bank'", True),
                                   ("'earl'", "'castle'", True),
                                   ("'hippie'", "'bank'", True),
                                   ("'fireman'", "'park'", True),
                                   ("'captain'", "'park'", True),
                                   ("'hippie'", "'park'", True),
                                   ("'fireman'", "'store'", False),
                                   ("'captain'", "'store'", False),
                                   ("'giant'", "'store'", False),
                                   ("'fireman'", "'bank'", False),
                                   ("'captain'", "'bank'", False),
                                   ("'giant'", "'bank'", False),
                                   ("'lawyer'", "'park'", False),
                                   ("'earl'", "'park'", False),
                                   ("'giant'", "'park'", False)]:

        data.append(sentence(person,location,target,term))

    return data
```

The experiment function takes no parameters and will run the model through the experiment. It calls the do_person_location function once to favor the person and once to favor the location and prints out a table of the average response times and whether or not both responses were correct.

```
def experiment():

         output_person_location(list(map(lambda  x,y:((x[0]+y[0])/2,(x[1]  and
y[1])),
                                    do_person_location('person'),
                                    do_person_location('location'))))
```

The output_person_location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that data to the experimental results and prints a table of the response times for targets and foils.

```
def output_person_location(data):

    rts = list(map(lambda x: x[0],data))

    actr.correlation(rts,person_location_data)
    actr.mean_deviation(rts,person_location_data)

    print("\nTARGETS:\n                         Person fan")
    print("  Location      1             2             3")
    print("    fan")

    for i in range(3):
        print("     %d      " % (i+1),end="")
        for j in range(3):
            print("%6.3f (%-5s)" % (data[j + (i * 3)]),end="")
        print()

    print()
    print("FOILS:")
    for i in range(3):
        print("     %d      " % (i+1),end="")
        for j in range(3):
            print("%6.3f (%-5s)" % (data[j + ((i + 3) * 3)]),end="")
        print()
```

Fan-no-pm Model

The model for the task which does not use the perceptual and motor modules is very similar to the one described in the unit. It goes through the same steps of encoding the person and location and then retrieving the item, but it does not have to read them from the display nor perform a key press to respond. Because those perceptual and motor actions take time, a model that does not perform them will be faster at the task than one that does. To still fit the human performance on the task this model adjusts the time it takes to fire the productions from the default time of 50ms to account for that difference. That is done using the spp command which was used in unit 3 to adjust the starting utility of the productions. These settings at the end of the model file adjust the :at value (action time) of the productions which controls how long they take to fire:

```
  (spp mismatch-location-no :at .21)
  (spp mismatch-person-no :at .21)
  (spp respond-yes :at .21)
  (spp start :at .250)
  (spp harvest-person :at .285)
```

Those settings are free parameters in the model, and they are not unique – there are many ways to set them to get the same results and it could have all been attributed to a single production. In fitting this data one also has to adjusted the latency factor and maximum associative strength values which control the timing and activation of the underlying chunks. Using the perceptual and motor modules to perform the task gives the model a reasonable starting point for human performance and saves having to estimate the additional action time along with the declarative memory performance, but it may involve more work to setup the task and the model. Something else to note is that we kept the two versions of this model similar for comparison purposes, but the one that does not use the perceptual and motor components could have been made even simpler by skipping the encoding steps and just performing the retrieval after placing the appropriate values into the slots and then testing the result in the code as well instead of with productions. There is no single best approach for creating the model and task, and you will have to consider the options and their tradeoffs when approaching a new task with respect to the objectives of the modeling effort.

Another difference is that this model uses the goal buffer instead of the imaginal module to hold the items. Therefore it must set the :ga parameter to have any activation spreading from those items since by default only the imaginal buffer is a source of activation. It is set to 1 in this model to match the other version since that is the default value for activation spreading from the imaginal buffer.

Grouped

The grouped model is really just a demonstration of partial matching. The experiment code is only there to collect and display key presses of the model. It is not an interactive experiment which a person can perform nor does it have a direct comparison to any existing data.

The experiment code is very simple, and does not use any new commands. However, the model which interacts with that code does use a new production capability which will be described after the experiment code.

Lisp

It loads the corresponding model, and creates a global variable to hold the responses that the model makes:

```
(load-act-r-model "ACT-R:tutorial;unit5;grouped-model.lisp")

(defvar *response* nil)
```

The grouped-recall function is fairly simple. It adds a new command for the record-response function defined later, clears the global response variable, resets the model, runs it for up to 20 seconds, removes the new command, and then returns the response list after reversing it.

```
(defun grouped-recall ()
  (add-act-r-command "grouped-response" 'record-response
                     "Response recording function for the tutorial grouped model.")
  (setf *response* nil)
  (reset)
  (run 20)
  (remove-act-r-command "grouped-response")
  (reverse *response*))
```

Unlike previous tasks where the data collection was done through monitoring the model's output actions, here we create a simple function for collecting the data directly which just pushes the values provided onto the response list. How that function actually gets called by the model will be described below.

```
(defun record-response (value)
  (push value *response*))
```

Python

It imports the actr module, loads the corresponding model, and creates a global variable to hold the responses that the model makes:

```
import actr

actr.load_act_r_model("ACT-R:tutorial;unit5;grouped-model.lisp")

response = []
```

The recall function is fairly simple. It adds a new command for the record_response function defined later, clears the global response variable, resets the model, runs it for up to 20 seconds, removes the new command, and then returns the response list.

```
def recall ():

    actr.add_command("grouped-response",record_response,
                     "Response recording function for the tutorial grouped model.")
    global response
    response = []
    actr.reset()
    actr.run(20)
    actr.remove_command("grouped-response")
    return response
```

Unlike previous tasks where the data collection was done through monitoring the model's output actions, here we create a simple function for collecting the data directly which just appends the values provided onto the response list. How that function actually gets called by the model will be described below.

```
def record_response (item):

    global response
    response.append(item)
```

Calling ACT-R commands from productions

It is possible to call a command which is available in ACT-R from within a production, and that is how this model provides its responses – it directly calls the grouped-response command that is added by the experiment code. That is done in the harvest-first-item, harvest-second-item, and harvest-third-item productions. Here is the harvest-first-item production.

```
(p harvest-first-item
   =goal>
      isa      recall-list
      element  first
      group    =group
   =retrieval>
      isa      item
      name     =name
  ==>
   =goal>
      element  second
   +retrieval>
      isa      item
      group    =group
      position second
      :recently-retrieved nil
   !eval! ("grouped-response" =name))
```

The new operation shown in that production is !eval!. [The '!' is called bang in Lisp, so that's pronounced bang-eval-bang.] It can be placed on either the LHS or RHS of a production and must be followed by a call to an ACT-R command using Lisp style syntax specifying the string which names the command followed by any parameters to pass to it inside of parentheses. [Note: a valid Lisp expression can also be provided for a !eval! operation instead of specifying an ACT-R command using the string of its name.]

On the RHS of a production all the !eval! operation does is evaluate the expression provided. When the production fires, the !eval! is just another action that occurs. If that expression contains variables from the production the current binding of that variable in the instantiation is what will be used in the expression.

On the LHS of a production a !eval! specifies a condition that must be met before the production can be selected just like all the other items on the LHS. The value returned by the evaluation of a LHS !eval! must be true for the production to be selected (where true is technically anything that is not nil in Lisp and if the command is implemented in some other language the values must not be the equivalent of nil e.g. False and None in Python are equivalent to nil in Lisp). Because it will be called during the selection process, a LHS !eval! is likely to be evaluated very often and may be called even when the production that it is in is not the one that will be eventually selected and fired.

Using !eval! can be a powerful tool, but it can easily be abused. Using it to call commands as an abstraction for an aspect of the model which is not necessary to model in detail for a particular task is the recommended use. In general, the predictions of the model should not depend heavily on the use of !eval!, otherwise there is not really any point to using ACT-R to create a model – you might as well just generate some functions to produce the data you want.

In this model !eval! is used to collect the model's responses without needing the overhead of creating an experiment with which to interact. Because this task is not presenting information to the model or concerned with the response time there is not really a need for an interactive experiment and !eval! provides an easy alternative. For the assignment tasks in the tutorial however you should **not** be using !eval! in any of the productions which you write.

Siegler

The **Siegler** task is only available for a model to perform because there is no display to see and it records the model's speech as a response. There is only one new function used in the code, so it should be easy to follow.

Lisp

It starts by loading the corresponding model.

```
(load-act-r-model "ACT-R:tutorial;unit5;siegler-model.lisp")
```

It defines global variables to hold the response, record whether or not it is already monitoring the speech output command, and to hold the experimental data to which the model will be compared.

```
(defvar *response*)
(defvar *monitor-installed* nil)

(defvar *siegler-data* '((0   .05 .86  0  .02  0  .02  0   0  .06)
                         (0   .04 .07 .75 .04  0  .02  0   0  .09)
                         (0   .02  0  .10 .75 .05 .01 .03  0  .06)
                         (.02  0  .04 .05 .80 .04  0  .05  0   0)
                         (0    0  .07 .09 .25 .45 .08 .01 .01 .06)
                         (.04  0   0  .05 .21 .09 .48  0  .02 .11)))
```

The record-model-speech function will monitor the output-speech command to record the model's vocal response.

```
(defun record-model-speech (model string)
  (declare (ignore model))

  (setf *response* string))
```

Because there are multiple functions which can be used to perform different subsets of the task we create some functions to add and remove the monitoring function and keep track of whether or not it has already been created so we do not have to keep adding and removing it for efficiency when running the whole task repeatedly.

```
(defun add-speech-monitor ()
  (unless *monitor-installed*
    (add-act-r-command "siegler-response" 'record-model-speech "Siegler task model response")
    (monitor-act-r-command "output-speech" "siegler-response")
    (setf *monitor-installed* t)))

(defun remove-speech-monitor ()
  (remove-act-r-command-monitor "output-speech" "siegler-response")
  (remove-act-r-command "siegler-response")
  (setf *monitor-installed* nil))
```

The siegler-trial function takes two parameters which are the numbers to present.

```
(defun siegler-trial (arg1 arg2)
```

It resets the model and installs the microphone device to record the speech. In previous tasks that wasn't necessary because it is installed automatically with an experiment window, but because there is no window this time it must be installed explicitly.

```
  (reset)


  (install-device (list "speech" "microphone"))
```

Check and record whether the speech monitor has already been added.

```
  (let ((need-to-remove (add-speech-monitor)))
```

It schedules the presentation of the digits to the model aurally with the new-digit-sound command which is very similar to the new-tone-sound command seen previously in the tutorial.

```
(new-digit-sound arg1)
(new-digit-sound arg2 .75)
```

It clears the response variable, runs the model for up to 30 seconds, removes the speech monitor if it was installed with this trial, and then returns the response.

```
    (setf *response* nil)
    (run 30)
    (when need-to-remove
      (remove-speech-monitor))

  *response*))
```

The siegler-set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses. Like the single trial function it also checks the monitoring state and removes the monitor if it had to add it.

```
(defun siegler-set ()

  (let ((need-to-remove (add-speech-monitor))
        (data (list (siegler-trial 1 1)
                    (siegler-trial 1 2)
                    (siegler-trial 1 3)
                    (siegler-trial 2 2)
                    (siegler-trial 2 3)
                    (siegler-trial 3 3))))
    (when need-to-remove
      (remove-speech-monitor))
    data))
```

The siegler-experiment function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It installs the speech output monitor, runs that many times over the set of stimuli, removes the monitor, and then outputs the results.

```
(defun siegler-experiment (n)
  (add-speech-monitor)
  (let ((data nil))
    (dotimes (i n)
      (push (siegler-set) data))
    (remove-speech-monitor)
    (analyze data)))
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the answers from zero to eight or other from the given trial data and calls display-results to print out that information.

```
(defun analyze (responses)
  (display-results
   (mapcar (lambda (x)
             (mapcar (lambda (y)
                       (/ y (length responses))) x))
     (apply 'mapcar
            (lambda (&rest z)
              (let ((res nil))
                (dolist (i '("zero" "one" "two" "three" "four"
                             "five" "six" "seven" "eight"))
                  (push (count i z :test 'string-equal) res)
                  (setf z (remove i z :test 'string-equal)))
                (push (length z) res)
                (reverse res)))
            responses))))
```

The display-results function takes one parameter which is a list of six lists where each sublist is a list of ten items which represent the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *siegler-data*)
    (mean-deviation results *siegler-data*)
    (format t "       0     1     2     3     4     5     6     7     8   Other~%")
    (dotimes (i 6)
      (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results)))))
```

Python

It starts by importing the actr module and loading the corresponding model.

```
import actr

actr.load_act_r_model("ACT-R:tutorial;unit5;siegler-model.lisp")
```

It defines global variables to hold the response, record whether or not it is already monitoring the speech output command, and to hold the experimental data to which the model will be compared.

```
response = False
monitor_installed = False

siegler_data = [[0, .05, .86,  0,  .02,  0, .02, 0, 0, .06],
                [0, .04, .07, .75, .04,  0, .02, 0, 0, .09],
                [0, .02, 0, .10, .75, .05, .01, .03, 0, .06],
                [.02, 0, .04, .05, .80, .04, 0, .05, 0, 0],
                [0, 0, .07, .09, .25, .45, .08, .01, .01, .06],
                [.04, 0, 0, .05, .21, .09, .48, 0, .02, .11]]
```

The record_model_speech function will monitor the output-speech command to record the model’s vocal response.

```
def record_model_speech (model,string):
    global response
    response = string.lower()
```

Because there are multiple functions which can be used to perform different subsets of the task we create some functions to add and remove the monitoring function and keep track of whether or not it has already been created so we do not have to keep adding and removing it for efficiency when running the whole task repeatedly.

```
def add_speech_monitor():
    global monitor_installed

    if monitor_installed == False:
        actr.add_command("siegler-response",record_model_speech,
                         "Siegler task model response")
        actr.monitor_command("output-speech","siegler-response")
        monitor_installed = True
        return True
    else:
        return False

def remove_speech_monitor():

    actr.remove_command_monitor("output-speech","siegler-response")
    actr.remove_command("siegler-response")

    global monitor_installed
    monitor_installed = False
```

The trial function takes two parameters which are the numbers to present.

```
def trial(arg1,arg2):
```

It resets the model and installs the microphone device to record the speech. In previous tasks that wasn't necessary because it is installed automatically with an experiment window, but because there is no window this time it must be installed explicitly.

```
    actr.reset()
    actr.install_device(["speech","microphone"])
```

Check and record whether the speech monitor has already been added.

```
    need_to_remove = add_speech_monitor()
```

It schedules the presentation of those digits to the model aurally with the new_digit_sound function which is very similar to the new_tone_sound function seen previously in the tutorial.

```
    actr.new_digit_sound(arg1)
    actr.new_digit_sound(arg2,.75)
```

It clears the response variable, runs the model for up to 30 seconds, removes the speech monitor if it was installed with this trial, and then returns the response.

```
global response
response = False
actr.run(30)
if need_to_remove:
    remove_speech_monitor()

return response
```

The set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses. Like the single trial function it also checks the monitoring state and removes the monitor if it had to add it.

```
def set ():

    need_to_remove = add_speech_monitor()

    data = [trial(1,1),trial(1,2),trial(1,3),
            trial(2,2),trial(2,3),trial(3,3)]

    if need_to_remove:
        remove_speech_monitor()

    return data
```

The experiment function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It installs the speech output monitor, runs that many times over the set of stimuli, removes the monitor, and then outputs the results.

```
def experiment(n):

    add_speech_monitor()

    data = []

    for i in range(n):
        data.append(set())

    remove_speech_monitor()
    analyze(data)
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the answers from zero to eight or other from the given trial data and calls display_results to print out that information.

```
def analyze(responses):

    results = [[0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0]]
    positions = {'zero':0,'one':1,'two':2,'three':3,'four':4,'five':5,
                 'six':6,'seven':7,'eight':8}


    for r in responses:
        for i in range(6):
            if r[i] in positions:
                results[i][positions[r[i]]] += 1
            else:
                results[i][9] += 1

    n = len(responses)

    for i in range(6):
        for j in range(10):
            results[i][j] /= n

    display_results(results)
```

The display_results function takes one parameter which is a list of six lists where each sublist is a list of ten items which represent the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
def display_results(results):

    questions = ["1+1","1+2","1+3","2+2","2+3","3+3"]

    actr.correlation(results,siegler_data)
    actr.mean_deviation(results,siegler_data)


    print("       0     1     2     3     4     5     6     7     8   Other")
    for i in range(6):
        print(questions[i],end="")

        for j in range(10):
            print("%6.2f" % results[i][j],end="")
        print()
```

1-hit Blackjack

Next, we'll look at the 1-hit blackjack task. It has a lot of code to go with it to play the game, control the opponent, analyze the results, and allow a person to play against the model, and this code is flexible in that it allows for the game and opponent to be changed without having to change the existing code. How exactly to use that will be discussed in a separate section below, and for now we will just describe the code.

Lisp

Unlike the other tasks this one starts by defining a function and adding a new command before loading the corresponding model. The function computes the similarities between numbers for the model and it must be added as a command before loading the model because the model's :sim-hook parameter setting specifies the "1hit-bj-number-sims" command so it needs to be available when it is loaded.

```
(defun 1hit-bj-number-sims (a b)
  (when (and (numberp a) (numberp b))
    (- (/ (abs (- a b)) (max a b)))))

(add-act-r-command "1hit-bj-number-sims" '1hit-bj-number-sims
                   "Similarity between numbers for 1-hit blackjack task.")

(load-act-r-model "ACT-R:tutorial;unit5;1hit-blackjack-model.lisp")
```

It defines a lot of global variables which are used to hold the game information, player responses, and to control the default opponent for the model.

```
(defvar *deck1*)
(defvar *deck2*)
(defvar *opponent-rule*)
(defvar *opponent-feedback*)
(defvar *model-action*)
(defvar *human-action*)
(defvar *opponent-threshold*)
(defvar *key-monitor-installed* nil)
```

The respond-to-keypress function will be monitoring the output-key command and records the response for either a model or person playing the game and like the siegler code it has functions for adding and removing the monitor for efficiency.

```
(defmethod respond-to-keypress (model key)
  (if model
      (setf *model-action* key)
    (setf *human-action* key)))

(defun add-key-monitor ()
  (unless *key-monitor-installed*
    (add-act-r-command "1hit-bj-key-press" 'respond-to-keypress
                       "1-hit blackjack task key output monitor")
    (monitor-act-r-command "output-key" "1hit-bj-key-press")
    (setf *key-monitor-installed* t)))

(defun remove-key-monitor ()
  (remove-act-r-command-monitor "output-key" "1hit-bj-key-press")
  (remove-act-r-command "1hit-bj-key-press")
  (setf *key-monitor-installed* nil))
```

The onehit-hands function takes one required parameter which is the number of hands to play and an optional parameter which indicates whether or not to print the details of each hand played.

```
(defun onehit-hands (hands &optional (print-game nil))
```

Initialize a list of results and add the key press monitor if needed and record whether it was added.

```
  (let ((scores (list 0 0 0 0))
    (need-to-remove (add-key-monitor)))
```

For each hand deal the cards for the players from the appropriate decks, show the model its first two cards and the opponents first card and record its response, and then do the same for the opponent.

```
(dotimes (i hands)
  (let* ((mcards (deal *deck1*))
         (ocards (deal *deck2*))
         (mchoice (show-model-cards (butlast mcards) (first ocards)))
         (ochoice (show-opponent-cards (butlast ocards) (first mcards))))
```

If a player hit set their hand to all three cards, otherwise only the first two.

```
    (unless (string-equal "h" mchoice)
      (setf mcards (butlast mcards)))
    (unless (string-equal "h" ochoice)
      (setf ocards (butlast ocards)))
```

Determine the appropriate values for the players hands, determine their final outcomes, and then show those outcomes to the model and the opponent.

```
    (let* ((mtot (score-cards mcards))
           (otot (score-cards ocards))
           (mres (compute-outcome mcards ocards))
           (ores (compute-outcome ocards mcards)))

      (show-model-results mcards ocards mres ores)
      (show-opponent-results ocards mcards ores mres)
```

If the details are requested output those to *standard-output*.

```
      (when print-game
        (format t "Model: ~{~2d ~} -> ~2d (~4s)   Opponent: ~{~2d ~}-> ~2d (~4s)~%"
          mcards mtot mres ocards otot ores))
```

Update the scores based on the outcomes.

```
      (setf scores (mapcar '+ scores
                     (list (if (eq mres 'win) 1 0)
                           (if (eq ores 'win) 1 0)
                           (if (and (eq mres 'bust) (eq ores 'bust)) 1 0)
                           (if (and (= mtot otot)
                                    (not (eq mres 'bust))
                                    (not (eq ores 'bust))) 1 0)))))))
```

When done check whether the monitor needs to be removed and return the total scores.

```
(when need-to-remove
  (remove-key-monitor))

scores))
```

The onehit-blocks function takes two parameters: a number of blocks to run and how many hands to play in each block. It uses onehit-hands to play each block and returns the list of block scores.

```
(defun onehit-blocks (blocks block-size)
  (let (res
        (need-to-remove (add-key-monitor)))
    (dotimes (i blocks)
      (push (onehit-hands block-size) res))
    (when need-to-remove
      (remove-key-monitor))
    (reverse res)))
```

The game0 function sets all of the global variables to the values which describe the default game. *deck1* and *deck2* are set to the functions that implement the regular card deck. The *opponent-rule* is set to the fixed-threshold function, and the threshold used is set to 15. The default opponent does not change its play and does not need to see the feedback.

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 15)
  (setf *opponent-feedback* nil))
```

The onehit-learning function takes one required parameter which is the number of 100 hand games to play and average. It takes two optional parameters. The first indicates whether or not a graph of the model's win proportions should be drawn and the second is the function to call to initialize the game control information. The defaults are to draw the graph and play the game set by the game0 function. It plays the indicated number of games resetting the model before each and collecting the data. If requested the graph is drawn, and then the lists of the win proportions are returned grouped into blocks of size 25 and 5.

```
(defun onehit-learning (n &optional (graph t) (game 'game0))
  (let ((data nil)
        (need-to-remove (add-key-monitor)))
    (dotimes (i n)
      (reset)
      (funcall game)
      (if (null data)
          (setf data (onehit-blocks 20 5))
        (setf data (mapcar (lambda (x y)
                             (mapcar '+ x y))
                     data (onehit-blocks 20 5)))))

    (when need-to-remove
      (remove-key-monitor))

    (let ((percentages (mapcar (lambda (x) (/ (car x) n 5.0)) data)))
      (when graph
        (draw-graph percentages))

      (list (list (/ (apply '+ (subseq percentages 0 5)) 5)
                  (/ (apply '+ (subseq percentages 5 10)) 5)
```

```
                    (/ (apply '+ (subseq percentages 10 15)) 5)
                    (/ (apply '+ (subseq percentages 15 20)) 5))
                    percentages))))
```

The draw-graph function takes one parameter which is a list of win percentages. It opens an experiment window and then draws a graph of those results in that window.

```
(defun draw-graph (points)
  (let ((w (open-exp-window "Data" :visible t :width 550 :height 460)))
    (add-line-to-exp-window w '(50 0) '(50 420) 'white)
    (dotimes (i 11)
      (add-text-to-exp-window w (format nil "~3,1f" (- 1 (* i .1)))
                              :x 5 :y (+ 5 (* i 40)) :width 35)
      (add-line-to-exp-window w (list 45 (+ 10 (* i 40)))
                              (list 550 (+ 10 (* i 40))) 'white))

    (let ((x 50))
      (mapcar (lambda (a b)
                (add-line-to-exp-window w (list x (floor (- 410 (* a 400))))
                                          (list (incf x 25) (floor (- 410 (* b 400))))
                                          'blue))
        (butlast points) (cdr points)))))
```

The deal function takes one parameter which is a function that implements a deck of cards for a player. That function is called three times to get the next three cards for a player and returns them in a list.

```
(defun deal (deck)
  (list (funcall deck)
        (funcall deck)
        (funcall deck)))
```

The score-cards function takes one required parameter which is a list of cards and an optional parameter which indicates the value over which a player busts. It returns the total value for the cards provided counting 1s as 11 as long as it does not bust.

```
(defun score-cards (cards &optional (bust 21))
  (let ((total (apply '+ cards)))
    (dotimes (i (count 1 cards))
      (when (<= (+ total 10) bust)
        (incf total 10)))
    total))
```

The compute-outcome function takes two required parameters which are the lists of cards for the two players and an optional parameter which is the limit before busting. It computes the score for each player and returns one of the symbols bust, win, or lose to indicate the result for the player holding the first set of cards provided.

```
(defun compute-outcome (p1cards p2cards &optional (bust 21))
  (let ((p1tot (score-cards p1cards bust))
        (p2tot (score-cards p2cards bust)))
    (if (> p1tot bust)
        'bust
      (if (or (> p2tot bust) (> p1tot p2tot))
          'win
        'lose))))
```

The show-model-cards function takes two parameters. The first is a list of the model's starting cards, and the second is the visible card of the opponent. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for the start of a hand, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds and the response it makes returned.

```
(defun show-model-cards (mcards ocard)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards) mc2 ,(second mcards) mc3 nil
                     mtot nil mstart ,(score-cards mcards) mresult nil
                     oc1 ,ocard oc2 nil oc3 nil otot nil
                     ostart ,(score-cards (list ocard)) oresult nil
                     state start))
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                                  mc2 ,(second mcards)
                                  mstart ,(score-cards mcards)
                                  oc1 ,ocard ostart ,(score-cards (list ocard))
                                  state start))))))
  (setf *model-action* nil)
  (run-full-time 10)
  *model-action*)
```

The show-model-results function takes four parameters. The first is a list of the model's cards, and the second is the list of the opponent's cards. The next two are the symbols representing the outcome for the model and the opponent respectively. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for providing the model the results, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds.

```
(defun show-model-results (mcards ocards mres ores)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards)  mc2 ,(second mcards)
                   mc3 ,(third mcards) mtot ,(score-cards mcards)
                   mstart ,(score-cards (subseq mcards 0 2))
                   mresult ,mres oc1 ,(first ocards)
                   oc2 ,(second ocards) oc3 ,(third ocards)
                   otot ,(score-cards ocards)
                   ostart ,(score-cards (list (first ocards)))
                   oresult ,ores state results))
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                               mc2 ,(second mcards) mc3 ,(third mcards)
                               mtot ,(score-cards mcards)
                               mstart ,(score-cards (subseq mcards 0 2))
                               mresult ,mres oc1 ,(first ocards)
                               oc2 ,(second ocards) oc3 ,(third ocards)
                               otot ,(score-cards ocards)
                               ostart ,(score-cards (list (first ocards)))
                               oresult ,ores state results))))))
  (run-full-time 10))
```

The play-human function is similar to the show-model-cards function but for a human player. It is takes two parameters. The first is a list of the player's starting cards, and the second is the visible card of the opponent. It opens an experiment window and displays the available information and then waits 10 seconds. It returns any keypress made during that time, or "s" indicating stay if no response is made.

```
(defun play-human (cards oc1)
  (let ((win (open-exp-window "Human")))
    (add-text-to-exp-window win "You" :x 50 :y 20)
    (add-text-to-exp-window win "Model" :x 200 :y 20)
    (dotimes (i 2)
      (dotimes (j 3)
        (add-text-to-exp-window win (format nil "C~d" (1+ j))
                                :x (+ 25 (* j 30) (* i 150)) :y 40 :width 20)
        (cond ((and (zerop i) (< j 2))
               (add-text-to-exp-window win (princ-to-string (nth j cards))
                                       :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20))
              ((and (= i 1) (zerop j))
               (add-text-to-exp-window win (princ-to-string oc1)
                                       :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20)))))
    (setf *human-action* nil)

    (let ((start-time (get-time nil)))
      (while (< (- (get-time nil) start-time) 10000)
        (process-events)))

    (if *human-action*
        *human-action*
      "s")))
```

The show-human-results function is similar to the show-model-results function but for a human player. It takes four parameters. The first is a list of the player's cards, and the second is the list of the opponent's cards. The next two are the symbols representing the outcome for the player and the opponent respectively. It displays the information in an experiment window and then waits for 10 seconds to pass.

```
(defun show-human-results (own-cards others-cards own-result others-result)
  (let ((win (open-exp-window "Human")))
    (add-text-to-exp-window win "You" :x 50 :y 20)
    (add-text-to-exp-window win "Model" :x 200 :y 20)
    (dotimes (i 2)
      (dotimes (j 3)
        (add-text-to-exp-window win (format nil "C~d" (1+ j))
                                :x (+ 25 (* j 30) (* i 150)) :y 40 :width 20)
        (if (zerop i)
            (when (nth j own-cards)
              (add-text-to-exp-window win (princ-to-string (nth j own-cards))
                                      :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20))
            (when (nth j others-cards)
              (add-text-to-exp-window win (princ-to-string (nth j others-cards))
                                      :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20)))))
    (add-text-to-exp-window win (princ-to-string own-result) :x 50 :y 85)
    (add-text-to-exp-window win (princ-to-string others-result) :x 200 :y 85)
    (let ((start-time (get-time nil)))
      (while (< (- (get-time nil) start-time) 10000)
        (process-events)))))
```

The play-against-model function can be used to play as a person against the current model. It takes one required parameter which is the number of hands to play, and if the optional parameter is provided as non-nil then it will print out the details for each hand. If a visible window can be displayed, then it sets the interface variables to those necessary for a person to play, plays the requested number of hands, and then restores the interface variables to their previous values.

```
(defun play-against-model (count &optional (print-game nil))
  (if (visible-virtuals-available?)
      (let* ((old-rule *opponent-rule*)
             (old-feedback *opponent-feedback*)
             (*opponent-rule* 'play-human)
             (*opponent-feedback* 'show-human-results))
        (unwind-protect
            (onehit-hands count print-game)
          (progn
            (setf *opponent-rule* old-rule)
            (setf *opponent-feedback* old-feedback))))
       (print-warning "Cannot play against the model without a visible
window.")))
```

The show-opponent-cards function takes two parameters which are the list of cards and the other player’s face up card. It calls the function which has been set to determine the model’s opponent’s action to take using that information.

```
(defun show-opponent-cards (cards mc1)
  (funcall *opponent-rule* cards mc1))
```

The show-opponent-results function takes four parameters. The first is a list of the player’s cards, and the second is the list of the model’s cards. The next two are the symbols representing the outcome for the player and the model respectively. It calls the function which has been set to provide feedback to the model’s opponent if there is such a function.

```
(defun show-opponent-results (ocards mcards ores mres)
  (when *opponent-feedback*
    (funcall *opponent-feedback* ocards mcards ores mres)))
```

The regular-deck function returns card values as if they are dealt from an infinitely large deck of cards in the regular card deck proportions – one card for each of 1-9 and four 10s.

```
(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))
```

The fixed-threshold function implements an opponent for the model that will hit if the total value of their starting cards is less than the value of *opponent-threshold*.

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*)
      "h"
    "s"))
```

Create a global variable for holding the list of cards in a deck that is stacked in a particular fashion prior to the deal.

```
(defvar *card-list* nil)
```

The always-hit function implements an opponent for the model that will hit regardless of the cards it is dealt.

```
(defun always-hit (cards mc1)
  "h")
```

The load-stacked-deck function creates a list of six cards to deal which are stacked as described for game 1 in the unit – the face up card for the opponent is a perfect indicator for the action the model must take to win.

```
(defun load-stacked-deck ()
  (let* ((c1 (+ 5 (act-r-random 6)))
         (c2 (+ 7 (act-r-random 4)))
         (c4 (if (> (act-r-random 1.0) .5) 2 8))
         (c3 (if (= c4 2) 10 (- 21 (+ c1 c2))))
         (c5 10)
         (c6 (if (= c4 2) 10 2)))
    (list c1 c2 c3 c4 c5 c6)))
```

The stacked-deck function deals the cards from the list created by load-stacked-deck, and it generates a new list of cards when needed.

```
(defun stacked-deck ()
  (cond (*card-list* (pop *card-list*))
        (t (setf *card-list* (load-stacked-deck))
           (pop *card-list*))))
```

The game1 function takes no parameters and sets the variables which control the game to those necessary for playing game 1 as described in the unit text, and it can be passed to onehit-learning as the third parameter.

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit)
  (setf *opponent-feedback* nil))
```

Call the game0 function to set the default values for the control variables to those needed for playing game 0.

```
(game0)
```

Python

Start by importing the actr module and some other modules for the floor function and the Number class.

```
import actr
import math
import numbers
```

Unlike the other tasks this one starts by defining a function and adding a new command before loading the corresponding model. The function computes the similarities between numbers for the model and it must be added as a command before loading the model because the model's :sim-hook parameter setting specifies the "1hit-bj-number-sims" command so it needs to be available when it is loaded.

```
def onehit_bj_number_sims(a,b):

    if isinstance(b,numbers.Number) and isinstance(a,numbers.Number):
        return (- ( abs(a - b) / max(a,b)))
    else:
        return False

actr.add_command("1hit-bj-number-sims",onehit_bj_number_sims,
                 "Similarity between numbers for 1-hit blackjack task.")

actr.load_act_r_model("ACT-R:tutorial;unit5;1hit-blackjack-model.lisp")
```

It defines a lot of global variables which are used to hold the game information, player responses, and to control the default opponent for the model.

```
deck1 = None
deck2 = None
opponent_rule = None
opponent_feedback = None
model_action = None
human_action = None
opponent_threshold = None
key_monitor_installed = False
```

The respond_to_keypress function will be monitoring the output-key command and records the response for either a model or person playing the game and like the siegler code it has functions for adding and removing the monitor for efficiency.

```
def respond_to_keypress(model,key):
    global model_action,human_action

    if model:
        model_action = key
    else:
        human_action = key



def add_key_monitor():
    global key_monitor_installed

    if key_monitor_installed == False:
        actr.add_command("1hit-bj-key-press",respond_to_keypress,
                         "1-hit blackjack task key output monitor")
        actr.monitor_command("output-key","1hit-bj-key-press")
        key_monitor_installed = True
        return True
    else:
        return False

def remove_key_monitor():

    actr.remove_command_monitor("output-key","1hit-bj-key-press")
    actr.remove_command("1hit-bj-key-press")

    global key_monitor_installed
    key_monitor_installed = False
```

The hands function takes one required parameter which is the number of hands to play and an optional parameter which indicates whether or not to print the details of each hand played.

```
def hands(hands,print_game=False):
```

Initialize a list of results and add the key press monitor if needed and record whether it was added.

```
    scores = [0,0,0,0]
    need_to_remove = add_key_monitor()
```

For each hand deal the cards for the players from the appropriate decks, show the model its first two cards and the opponents first card and record its response, and then do the same for the opponent.

```
    for i in range(hands):
        mcards = deal(deck1)
        ocards = deal(deck2)
        mchoice = show_model_cards(mcards[0:2],ocards[0])
        ochoice = show_opponent_cards(ocards[0:2],mcards[0])
```

If a player hit set their hand to all three cards, otherwise only the first two.

```
        if not(mchoice.lower() == 'h'):
            mcards = mcards[0:2]

        if not(ochoice.lower() == 'h'):
            ocards = ocards[0:2]
```

Determine the appropriate values for the players hands, determine their final outcomes, and then show those outcomes to the model and the opponent.

```
        mtot = score_cards(mcards)
        otot = score_cards(ocards)
        mres = compute_outcome(mcards,ocards)
        ores = compute_outcome(ocards,mcards)

        show_model_results(mcards,ocards,mres,ores)
        show_opponent_results(ocards,mcards,ores,mres)
```

If the details are requested print them out.

```
        if print_game:
            print("Model: ",end="")
            for c in mcards:
                print("%2d "%c,end="")
            print(" -> %2d (%-4s)   Opponent: "%(mtot,mres),end="")
            for c in ocards:
                print("%2d "%c,end="")
            print("-> %2d (%-4s)"%(otot,ores))
```

Update the scores based on the outcomes.

```
        if mres == 'win':
            scores[0] += 1
        if ores == 'win':
            scores[1] += 1
        if mres == 'bust' and ores == 'bust':
            scores[2] += 1
        if mtot == otot and not(mres == 'bust') and not(ores == 'bust'):
            scores[3] += 1
```

When done check whether the monitor needs to be removed and return the total scores.

```
    if need_to_remove:
        remove_key_monitor()

    return scores
```

The blocks function takes two parameters: a number of blocks to run and how many hands to play in each block. It uses hands to play each block and returns the list of block scores.

```
def blocks(blocks,block_size):

    res = []
    need_to_remove = add_key_monitor()

    for i in range(blocks):
        res.append(hands(block_size))

    if need_to_remove:
        remove_key_monitor()
    return res
```

The game0 function sets all of the global variables to the values which describe the default game. deck1 and deck2 are set to the functions that implement the regular card deck. The opponent_rule is set to the fixed_threshold function, and the threshold used is set to 15. The default opponent does not change its play and does not need to see the feedback.

```
def game0():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 15
    opponent_feedback = None
```

The sum_lists function returns a list which is the sum of the values from the two lists provided, and is added to make the learning function a little easier to read.

```
def sum_lists(x,y):
    return list(map(lambda v,w: v + w,x,y))
```

The learning function takes one required parameter which is the number of 100 hand games to play and average. It takes two optional parameters. The first indicates whether or not a graph of the model's win proportions should be drawn and the second is the function to call to initialize the game control information. The defaults are to draw the graph and play the game set by the game0 function. It plays the indicated number of games resetting the model before each and collecting the data. If requested the graph is drawn, and then the lists of the win proportions are returned grouped into blocks of size 25 and 5.

```
def learning(n,graph=True,game=game0):

    data = [[0,0,0,0]]*20
    need_to_remove = add_key_monitor()

    for i in range(n):
        actr.reset()
        game()
        data = list(map(lambda x,y: sum_lists(x,y),data,blocks(20,5)))

    if need_to_remove:
        remove_key_monitor()

    percentages = list(map(lambda x: x[0] / n / 5,data))

    if graph:
        draw_graph(percentages)

    return [[sum(percentages[0:5])/5,
             sum(percentages[5:10])/5,
             sum(percentages[10:15])/5,
             sum(percentages[15:20])/5],
                 percentages]
```

The draw_graph function takes one parameter which is a list of win percentages. It opens an experiment window and then draws a graph of those results in that window.

```
def draw_graph(points):

    w = actr.open_exp_window('Data', visible=True, width=550, height=460)
    actr.add_line_to_exp_window(w,[50,0],[50,420],'white')

    for i in range(11):
        actr.add_text_to_exp_window(w,"%3.1f"%(1.0 - (i * .1)),
                                    x=5, y=(5 + (i * 40)), width=35)
        actr.add_line_to_exp_window(w,[45,10 + (i * 40)],
                                    [550,10 + (i * 40)],'white')

    x = 50

    for (a,b) in zip(points[0:-1],points[1:]):
        actr.add_line_to_exp_window(w,[x,math.floor(410 - (a * 400))],
                                    [x+25,math.floor(410 - (b * 400))],
                                    'blue')
        x += 25
```

The deal function takes one parameter which is a function that implements a deck of cards for a player. That function is called three times to get the next three cards for a player and returns them in a list.

```
def deal(deck):
    return [deck(),deck(),deck()]
```

The score_cards function takes one required parameter which is a list of cards and an optional parameter which indicates the value over which a player busts. It returns the total value for the cards provided counting 1s as 11 as long as it does not bust.

```
def score_cards(cards,bust=21):

    total = sum(cards)

    for i in range(cards.count(1)):
        if (total + 10) <= bust:
            total += 10

    return total
```

The compute_outcome function takes two required parameters which are the lists of cards for the two players and an optional parameter which is the limit before busting. It computes the score for each player and returns one of the strings bust, win, or lose to indicate the result for the player holding the first set of cards provided.

```
def compute_outcome(p1cards,p2cards,bust=21):
    p1tot = score_cards(p1cards,bust)
    p2tot = score_cards(p2cards,bust)
    if p1tot > bust:
        return 'bust'
    elif p2tot > bust:
        return 'win'
    elif p1tot > p2tot:
        return 'win'
    else:
        return 'lose'
```

The show_model_cards function takes two parameters. The first is a list of the model's starting cards, and the second is the visible card of the opponent. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for the start of a hand, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds and the response it makes returned (the default action is to stay unless the model hits a key).

```
def show_model_cards(mcards,ocard):

    if actr.buffer_read('goal'):
        actr.mod_focus('mc1',mcards[0],'mc2', mcards[1],'mc3',None,'mtot',None,
                       'mstart',score_cards(mcards),'mresult',None,'oc1',ocard,
                       'oc2',None,'oc3',None,'otot',None,'ostart',score_cards([ocard]),
                       'oresult',None,'state','start')
    else:
        actr.goal_focus(actr.define_chunks(['isa','game-state','mc1',mcards[0],
                                            'mc2', mcards[1],'mstart',score_cards(mcards),
                                            'oc1', ocard,'ostart',score_cards([ocard]),
                                            'state','start'])[0])

    global model_action
    model_action = 's'
    actr.run_full_time(10)
    return model_action
```

The show_model_results function takes four parameters. The first is a list of the model's cards, and the second is the list of the opponent's cards. The next two are the strings representing the outcome for the model and the opponent respectively. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for providing the model the results, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds.

```
def show_model_results(mcards,ocards,mres,ores):

    if len(mcards) ==3:
        mhit = mcards[2]
    else:
        mhit = 'nil'

    if len(ocards) ==3:
        ohit = ocards[2]
    else:
        ohit = 'nil'

    if actr.buffer_read('goal'):
        actr.mod_focus('mc1',mcards[0],'mc2', mcards[1],'mc3',mhit,
                       'mtot',score_cards(mcards),'mstart',score_cards(mcards[0:2]),
                       'mresult',mres,'oc1',ocards[0],'oc2',ocards[1],'oc3',ohit,
                       'otot',score_cards(ocards),
                       'ostart',score_cards(ocards[0:1]),'oresult',ores,'state','results')
    else:
                                             actr.goal_focus(actr.define_chunks(['isa','game-
state','mc1',mcards[0],'mc2',mcards[1],
                                          'mc3',mhit,'mtot',score_cards(mcards),
                                          'mstart',score_cards(mcards[0:2]),'mresult',mres,
                                          'oc1',ocards[0],'oc2',ocards[1],'oc3',ohit,
                                          'otot',score_cards(ocards),
                                          'ostart',score_cards(ocards[0:1]),'oresult',ores,
                                          'state','results'])[0])

    actr.run_full_time(10)
```

The play_human function is similar to the show_model_cards function but for a human player. It is takes two parameters. The first is a list of the player's starting cards, and the second is the visible card of the opponent. It opens an experiment window and displays the available information and then waits 10 seconds. It returns any keypress made during that time, or "s" indicating stay if no response is made.

```
def play_human(cards,oc1):

    win = actr.open_exp_window('Human')
    actr.add_text_to_exp_window(win,'You',50,20)
    actr.add_text_to_exp_window(win,'Model',200,20)

    for i in range(2):
        for j in range(3):
            actr.add_text_to_exp_window(win,"C%d"%(j+1),x=(25 + (j * 30) + (i * 150)),
                                        y=40, width=20)
            if i == 0 and j < 2:
                actr.add_text_to_exp_window(win,str(cards[j]),
                                            x=(25 + (j * 30) + (i * 150)), y=60, width=20)
            if i == 1 and j == 0:
                actr.add_text_to_exp_window(win,str(oc1),x=(25 + (j * 30) + (i * 150)),
                                            y=60, width=20)

    global human_action
    human_action = None

    start_time = actr.get_time(False)

    while (actr.get_time(False) - start_time) < 10000:
        actr.process_events()

    if human_action:
        return human_action
    else:
        return 's'
```

The show_human_results function is similar to the show_model_results function but for a human player. It takes four parameters. The first is a list of the player's cards, and the second is the list of the opponent's cards. The next two are the strings representing the outcome for the player and the opponent respectively. It displays the information in an experiment window and then waits for 10 seconds to pass.

```
def show_human_results(own_cards,others_cards,own_result,others_result):

    win = actr.open_exp_window('Human')
    actr.add_text_to_exp_window(win,'You',50,20)
    actr.add_text_to_exp_window(win,'Model',200,20)

    for i in range(2):
        for j in range(3):
            actr.add_text_to_exp_window(win,"C%d"%(j+1),x=(25 + (j * 30) + (i * 150)),
                                        y=40, width=20)
            if i == 0:
                if j < len(own_cards):
                    actr.add_text_to_exp_window(win,str(own_cards[j]),
                                                x=(25 + (j * 30) + (i * 150)),
                                                y=60, width=20)
            else:
                if j < len(others_cards):
                    actr.add_text_to_exp_window(win,str(others_cards[j]),
                                                x=(25 + (j * 30) + (i * 150)),
                                                y=60, width=20)

    actr.add_text_to_exp_window(win,own_result,50,85)
    actr.add_text_to_exp_window(win,others_result,200,85)

    start_time = actr.get_time(False)

    while (actr.get_time(False) - start_time) < 10000:
        actr.process_events()
```

The play_against_model function can be used to play as a person against the current model. It takes one required parameter which is the number of hands to play, and if the optional parameter is provided as True then it will print out the details for each hand. If a visible window can be displayed, then it sets the interface variables to those necessary for a person to play, plays the requested number of hands, and then restores the interface variables to their previous values.

```
def play_against_model(count,print_games=False):
    global opponent_rule,opponent_feedback

    if actr.visible_virtuals_available():
        old_rule = opponent_rule
        old_feedback = opponent_feedback

        opponent_rule = play_human
        opponent_feedback = show_human_results

        result = ()
        try:
            result = hands(count,print_games)
        finally:
            opponent_rule = old_rule
            opponent_feedback = old_feedback
        return(result)
    else:
        actr.print_warning("Cannot play against the model without a visible
                            window.")
```

The show_opponent_cards function takes two parameters which are the list of cards and the other player’s face up card. It calls the function which has been set to determine the model’s opponent’s action to take using that information.

```
def show_opponent_cards(ocards,mc1):
    return opponent_rule(ocards,mc1)
```

The show_opponent_results function takes four parameters. The first is a list of the player’s cards, and the second is the list of the model’s cards. The next two are the strings representing the outcome for the player and the model respectively. It calls the function which has been set to provide feedback to the model’s opponent if there is such a function.

```
def show_opponent_results(ocards,mcards,ores,mres):
    if opponent_feedback:
        opponent_feedback(ocards,mcards,ores,mres)
```

The regular_deck function returns card values as if they are dealt from an infinitely large deck of cards in the regular card deck proportions – one card for each of 1-9 and four 10s.

```
def regular_deck():
    return min(10,actr.random(13)+1)
```

The fixed_threshold function implements an opponent for the model that will hit if the total value of their starting cards is less than the value of opponent_threshold.

```
def fixed_threshold(cards,mc1):

    if score_cards(cards) < opponent_threshold:
        return 'h'
    else:
        return 's'
```

Create a global variable for holding the list of cards in a deck that is stacked in a particular fashion prior to the deal.

```
card_list = []
```

The always_hit function implements an opponent for the model that will hit regardless of the cards it is dealt.

```
def always_hit(cards,mc1):
    return 'h'
```

The load_stacked_deck function creates a list of six cards to deal which are stacked as described for game 1 in the unit – the face up card for the opponent is a perfect indicator for the action the model must take to win.

```
def load_stacked_deck():
    c1 = 5 + actr.random(6)
    c2 = 7 + actr.random(4)
    if actr.random(1.0) > .5:
        c4 = 2
    else:
        c4 = 8
    if c4 == 2:
        c3 = 10
        c6 = 10
    else:
        c3 = 21 - (c1 + c2)
        c6 = 2
    c5 = 10

    return [c1,c2,c3,c4,c5,c6]
```

The stacked_deck function deals the cards from the list created by load_stacked_deck, and it generates a new list of cards when needed.

```
def stacked_deck():
    global card_list

    if len(card_list) == 0:
        card_list = load_stacked_deck()

    c = card_list[0]
    card_list = card_list[1:]

    return c
```

The game1 function takes no parameters and sets the variables which control the game to those necessary for playing game 1 as described in the unit text, and it can be passed to the learning function as the third parameter.

```
def game1():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback,card_list

    card_list = []
    deck1 = stacked_deck
    deck2 = stacked_deck
    opponent_rule = always_hit
    opponent_feedback = None
```

Call the game0 function to set the default values for the control variables to those needed for playing game 0.

```
game0()
```

1hit-blackjack-model

In addition to the new parameters set in the model definition, which were described in the main unit text, there are three new commands used in this model. The first is this one:

```
  (declare-buffer-usage goal game-state :all)
```

That command is there to avoid the style warnings from the procedural module because the goal buffer is being tested for chunks which are not generated by the model itself. The declare-buffer-usage command informs the procedural module that a buffer will be set with chunk slot names other than those which are set in the model. It takes two required parameters which are the name of a buffer and the name of a chunk-type defined in the model. It takes an arbitrary number of additional parameters which indicate the slots from that chunk-type which will be set from outside of the model, or as is used here, the symbol :all to indicate that all of the slots from that chunk-type may be set externally.

This command is used to avoid the style warnings like these which would normally be printed when slots that are not set in the model are used in productions:

```
#|Warning: Productions test the MRESULT slot in the GOAL buffer which is not
requested or modified in any productions. |#

#|Warning: Productions test the MC1 slot in the GOAL buffer which is not
requested or modified in any productions. |#
```

Alternatively, instead of using declare-buffer-usage to specify the details of slots used in the goal buffer from outside of the model itself we could just turn off the warnings, but that is not recommended because with the warnings off one may miss other serious problems.

The next new command used is this one:

```
(define-chunks win lose bust retrieving start results)
```

That command is also used in the code which implements the game and will be described in detail below. The reason this exists in the model is to create the chunks for the items that are used in the productions to indicate the game information and the model’s internal state. That avoids the warnings for undefined chunks as was shown in the semantic model of unit 1, and because those chunks have no slots we can simply specify their names to create them.

The last new command is actually one that we have used in many of the previous experiments, but this time we have included it in the model in a slightly different form:

```
(install-device '("motor" "keyboard"))
```

Previously we have seen install-device used to have the model interact with a window in a task. It is more general than that however, and can be used to install any ACT-R device which has been created for a model to use. In this case, we are installing the keyboard device for the motor module to use. That has not been done previously in the experiments because the experiment windows automatically install a keyboard and mouse for the motor module along with the window for the vision module, but since this task does not have a visual interface for the model we are not creating an experiment window and need to install the keyboard explicitly so that we can collect the model's actions from key presses. It is also used in the siegler task in this unit to install the microphone device for the speech module to record the model's vocal responses.

Command Information

There were several new commands used in the tasks and models of this unit. However, before describing those commands we will first discuss some details about working with ACT-R from code.

Names

The code which implements ACT-R is written in Lisp, and in Lisp one of the fundamental data types is called a symbol. Very roughly speaking any sequence of alpha-numeric characters which is not purely numeric represents a symbol. They can be used for a variety of purposes, and a convenient use of symbols is to name things. Most of what you see in the model files are Lisp symbols e.g. everything in the count.lisp file for unit 1 except for the parentheses and numbers are symbols. The important issue is that internally all of the names of things in ACT-R are symbols – chunks, productions, parameters, slots, buffers, modules, etc. A Lisp symbol is different from a Lisp string which is specified in double quotes e.g. "goal" (like strings in most other languages) and that distinction can be an important factor when creating the chunks for a model (as we will discuss later with respect to the fan model from this unit). However, most other languages do not provide a construct like a symbol and the communication protocol used to connect to ACT-R also does not provide such a default type. Therefore, to accommodate interacting with other systems the ACT-R commands evaluated through the dispatcher accept strings as the names of items and convert strings to symbols to get the names, and similarly, names are converted to strings when returning them through the dispatcher. If you have been looking at the Python versions of the tasks and interface functions from other units you will have seen strings being passed to the functions and strings being returned, with the buffer-chunk function shown in unit2 as a good example where the Lisp version used symbols and the Python version used strings:

```
? (buffer-chunk goal)
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1

(SECOND-GOAL-0)

>>> actr.buffer_chunk('goal')
```

```
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1
['SECOND-GOAL-0']
```

Strings in slots

Generally, using strings to name things should not cause an issue, but there is one place where a little extra effort is required when using commands through the dispatcher. That situation is when one wants to put a string value into a chunk's slot or get the value of a slot which contains a string, as is done in the version of the fan experiment that does not use the perceptual and motor modules. Since the assumption is that strings from the external interface should be converted to symbols to specify names, to specify that something should be a string requires additional effort. To indicate a value is a string instead of a name one needs to add a set of single quotes around the item in the string. Similarly, when returning a value which should be considered as a string it will contain single quotes around the item. That can be seen in the code for the fan task above where it is testing the value which the model placed into the state slot since that was a string:

```
    response = actr.chunk_slot_value(actr.buffer_read("goal"),"state")

    if target:
        if response.lower() == "'k'".lower():
     ...
```

and also in the specification of the values which are being placed into the slots of the goal buffer by the code:

```
    for person,location,target in [("'lawyer'", "'store'", True),
                                   ("'captain'", "'cave'", True),
                                   ("'hippie'", "'church'", True),
    ...
```

Having to work with strings in the slots of chunks using commands through the dispatcher is probably not the sort of thing one will need to do very often, but when needed, some extra care will be required.

Additional Lisp command information

For many of the ACT-R commands the Lisp version provides both a macro and a function for accessing the command. The biggest distinction from the user's perspective is that when using a function in Lisp the parameters are evaluated first whereas a macro does not evaluate its parameters. Many of the ACT-R commands you've seen are actually macros e.g. chunk-type, add-dm, and p. They are macros so that you don't have to worry about quoting symbols or lists and other issues with Lisp syntax, but because they don't evaluate their parameters there are some things that you can't do with the macros (at least not without using an explicit call to eval and/or backquoting but those are Lisp programming techniques which will not be described here). For example, say you have a variable called *number* and you'd like to create a chunk that has the value of *number* in one of its slots. The following is not going to work:

```
? (defvar *number* 3)
*NUMBER*
? (chunk-type test slot)
TEST
? (add-dm (foo isa test slot *number*))
(FOO)
```

The add-dm macro doesn't evaluate the variable *number* to get its value. Instead that will create a chunk that literally contains *number* as shown here using the dm command shown in unit 1:

```
? (dm foo)
FOO
   SLOT  *NUMBER*
```

To allow modelers to do things like that most of the ACT-R macros have a corresponding function that does the same thing and the naming convention in the ACT-R interface is to add "-fct" to the name for the functional form of a command. When using the functional form of an ACT-R command you have to pay more attention to Lisp syntax and make sure that symbols are quoted and lists are constructed appropriately, and often the required parameters are a little different – things that do not need to be in a list for the macro version need to be in a list for the function. For example, add-dm-fct requires a list of lists as its only parameter instead of an arbitrary number of lists. Here is the code that would generate the chunk as desired above:

```
? (add-dm-fct (list (list 'foo 'isa 'test 'slot *number*)))
(FOO)
? (dm foo)
FOO
   SLOT  3
```

Not all of the ACT-R commands have both a macro and function available from Lisp, but for those that do we will provide the details from this point on when describing the new commands.

New Commands

Below we will describe the new commands used in this unit.

pdisable – This command can be used to disable productions. The Lisp macro and the Python function take any number of production names as parameters. The Lisp function **pdisable-fct** requires a list of production names as its only parameter. The named productions will be disabled in the current model. A disabled production cannot be selected during the conflict resolution process. Disabled productions will be enabled again if the model is reset or if explicitly enabled using the corresponding **penable** command. It returns a list with the names of all the productions which have been disabled in the current model.

define-chunks – Define-chunks is similar to add-dm which has been used in all of the previous models to create chunks and add them to the model's declarative memory. The difference between define-chunks and add-dm is that define-chunks creates the chunks specified but does not add them to the model's declarative memory. Keeping unnecessary information out of the model's declarative memory can be useful in multiple situations. One would be that as we saw in this unit the spreading activation depends on the fan of items and that fan is based on the contents of the chunks in declarative memory. So, putting chunks which do not represent the actual knowledge of the model into declarative memory could affect the activation of the important chunks. It is also important when creating chunks for a buffer which might later need to be retrieved after the model has manipulated them e.g. if the 1hit-blackjack model were retrieving the game-state chunks to make its decision we would not want the starting goal chunk placed into declarative memory prior to the model actually playing that game. Finally, it can also make things easier on the modeler when inspecting declarative memory while working with the model because it can be easier to find the relevant information if there are not a lot of irrelevant chunks there as well.

The Lisp macro (**define-chunks**) and the Python function (**define_chunks**) take any number of lists of chunk descriptions or names for chunks which will have no slots whereas the Lisp function (**define-chunks-fct**) requires a list of chunk description lists or names for chunks with no slots. They return a list of the names of the chunks that were created.

goal-focus – We have seen goal-focus used in models throughout the tutorial to schedule an action to place a chunk into the goal buffer. Here we are calling it from code to do the same thing. All versions, the Lisp macro, Python function (**goal_focus**), and the Lisp function (**goal-focus-fct**) take one optional parameter which names a chunk to place into the goal buffer. If the parameter is not specified then the command will print out the chunk in the goal buffer. It returns the name of the chunk which will be placed into the goal buffer or the chunk that is already there if no parameter is provided and no previous goal-focus action remains unexecuted.

mod-chunk - This command is used to modify a chunk. The Lisp macro version and the Python function (**mod_chunk**) require a chunk name and then an even number of additional parameters which indicate slots and values whereas the Lisp function (**mod-chunk-fct**) requires a chunk name and then a list of slots and values. Each of the slots specified for the chunk is given the corresponding value. It returns the name of the chunk which was modified.

One important thing to note is that once a chunk enters declarative memory it cannot be modified. That is another reason why one may want to use define-chunks instead of add-dm.

mod-focus – This command is very similar to **mod-chunk** except that it does not require the name of a chunk, only the slots and values, and it schedules those changes to be made at the current time to the chunk which is in the goal buffer, and it returns the name of the chunk which is in the goal buffer (or the chunk which is scheduled to enter the goal buffer if a goal-focus command has scheduled one to be put there at the current time as well).

chunk-slot-value – This command returns the value in a particular slot of a chunk. The parameters for all versions, Lisp macro (**chunk-slot-value**), Python function (**chunk_slot_value**), and Lisp function (**chunk-slot-value-fct**), are the name of the chunk and the name of the slot. It returns the value of that slot from that chunk or nil (Lisp) or None (Python) if the chunk does not have the specified slot.

buffer-read - This command is used to get the name of the chunk in a buffer. The Lisp function (**buffer-read**) and the Python function (**buffer_read**) both take one parameter which must be the name of a buffer. If that buffer contains a chunk then its name is returned otherwise nil or None is returned to indicate the buffer is empty.

new-digit-sound – This command creates a new sound stimulus for the model to provide numeric information and is similar to the new-tone-sound command which was used in unit 3 to present a tone. Both the Lisp function (**new-digit-sound**) and the Python function (**new_digit_sound**) require one parameter which is the number to present and also take two optional parameters. The first optional parameter can be provided to specify the time at which the digit should be presented (the current time will be used if a specific onset time is not given). The second optional parameter can be specified as a true value to indicate that the time is specified in milliseconds instead of the default units of seconds.

add-line-to-exp-window – this is similar to the Lisp function **add-text-to-exp-window** and the Python function **add_text_to_exp_window** which have been used in previous units. This function draws a line in an experiment window. It takes three required parameters and one optional parameter. The first required parameter is the window in which to draw the line. The other two are each a list of two integers which indicate the pixel coordinates of the end points of the line to be drawn (given as x and then y). The optional parameter can be provided to indicate the color of the line, and the default is black if it is not provided.

Modifying 1-hit blackjack

The last section for this unit will be to discuss how to change the decks and/or the opponent for the 1-hit blackjack game as well as provide some suggestions for some other opponents to test a model against.

To make the game flexible the code relies on functions being specified to handle the three changeable components of the game: the model's deck of cards, the opponent's deck of cards, and the code to determine whether or not the opponent will hit or stay. The functions are stored in global variables which are then called when needed. Thus, writing new functions for those parts of the game and setting the corresponding variables to those functions will change the way the game plays. To help make that more manageable, a function can be written to set all the appropriate values for a particular game scenario and that function can be passed to the onehit-learning or learning function as the (optional) third parameter. That setup function will be called at the start of each of the rounds that is played. Thus, to play 5 rounds of a game specified by a function named game1 and display the graph of the model's results these would be how that is called for the Lisp and Python versions (assuming the game1 function was also defined in the onehit.py file for the Python version):

```
? (onehit-learning 5 t 'game1)
```

```
>>> onehit.learning(5,True,onehit.game1)
```

The functions for the decks are held in the variables *deck1* and *deck2* for Lisp and deck1 and deck2 for Python. Deck1 holds the deck for the model's cards and deck2 the opponent's cards. A deck function should return a number from 1-10 representing the value of the card being dealt. On every hand each of the deck functions will be called three times. The first call will be for the face up card's value, the second call will be for the player's hidden card and the third call will be for the card that the player will receive on a hit. All three cards are dealt at the start of the hand, but only shown to the players when appropriate. The model's cards are dealt before the opponent's cards. Thus, if the same deck function is used for both players, as it is for the example games and the suggested alternatives, then the function will be called 6 times per hand with the first three calls returning the model's cards and the second set of three being for the opponent's cards.

For the default game described in the unit text both the model and opponent are dealt cards from these functions in Lisp or Python respectively:

```
(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))

def regular_deck():
    return min(10,actr.random(13)+1)
```

and the decks for both players are set to that value in the game0 functions like this:

```
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
```

or

```
    deck1 = regular_deck
    deck2 = regular_deck
```

The function for the model's opponent is called to determine if the opponent will hit or stay. The opponent function is stored in the variable *opponent-rule* for Lisp and opponent_rule in Python. It will be passed two parameters. The first parameter is a list of the opponent's two starting cards. The second parameter is the number of the model's face up card. That function should return either the string "h" if the opponent will hit the hand or the string "s" if the opponent will stay for that hand. Because we are only using a fixed opponent for the model there is no function called to provide feedback on the outcome of the hand to the opponent i.e. it has no way to learn about the game or the model, but the code does provide a variable for that (*opponent-feedback* or opponent_feedback) if one wanted to create a learning opponent to play against (that is used if you use the function to play the game yourself against the model).

For the default game from the unit the opponent has a simple rule of always hitting when it has a total of less than 15 for its starting cards. These are the functions which implement that:

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*) "h" "s"))


def fixed_threshold(cards,mc1):

    if score_cards(cards) < opponent_threshold:
        return 'h'
    else:
        return 's'
```

The score-cards and score_cards functions compute the score for a list of card values taking into account the rule of a 1 being counted as 11 when possible. Also note that we have introduced another variable for manipulating the game in these functions. The value at which the opponent will stay is set with the variable *opponent-threshold* or opponent_threshold. All of these variables are set to create the default game scenario in the game0 functions:

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 15)
  (setf *opponent-feedback* nil))


def game0():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 15
    opponent_feedback = None
```

That game is the default value for the third parameter to onehit-learning and learning if no game function is specified.

By writing a different game function to set the control variables one can easily modify the game that is played by the model. The simplest change would be to just adjust the threshold at which the default opponent stays. That could be done by adding a new game function to set the variables appropriately and then passing that function to the onehit-learning or learning function. A function like this would change the opponent to stay when it has a score of 12 or more instead:

```
(defun newgame ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 12)
  (setf *opponent-feedback* nil))
```

```
def newgame():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 12
    opponent_feedback = None
```

Then to run that game we would pass that newgame function to the onehit-learning or learning function like this:

```
? (onehit-learning 5 t 'newgame)

>>> onehit.learning(5,True,onehit.newgame)
```

There is a second game already programmed and available in the given code. It is called game1 and the setup function looks like this:

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit)
  (setf *opponent-feedback* nil))
```

```
def game1():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback,card_list

    card_list = []
    deck1 = stacked_deck
    deck2 = stacked_deck
    opponent_rule = always_hit
    opponent_feedback = None
```

The details of the functions used there can be found in the description of the code above.

There are some other game scenarios which we have designed to test the model's ability to learn, but which have not been included in the code. These can be implemented as an additional exercise if you would like, and of course, you are also free to create game scenarios of your own design for testing. The testing scenarios we outline here all assume the same deck function will be used for both the model and the opponent to keep things simpler, but that is not necessary since the two variables can be specified separately.

Game 2: In this game the deck consists of only cards numbered 7 and the opponent will always stay. In this game the model will always win if it hits and it should be able to learn that fairly quickly.

Game 3: The deck consists of an essentially infinite number of cards with only the values 8, 9, and 10 in equal proportions and again the opponent will always stay. The model should also learn to always stay because it will always lose if it hits. Staying on every hand will result in winning about 38.9% of the games.

Game 4: The deck consists of the cards 2, 4, 6, 8, and 10 being cycled in that order repeatedly. Thus there are only 5 possible hand combinations which will be cycled through in order:

Model's cards	**Opponent's cards**
2 4 6	8 10 2
4 6 8	10 2 4
6 8 10	2 4 6
8 10 2	4 6 8
10 2 4	6 8 10

The opponent for this game is one which always hits. In this game the model should be able to learn the correct move to win the 4 which can be won out of the 5 possible hands (80% wins by the end).

Game 5: The deck consists of only the following possible triples each equally likely in any deal to either player: (2 10 10) (4 9 9) (6 8 8) (8 8 5) (9 9 3) (10 10 1). These hands are designed so that if the player's initial score is small (12, 13 or 14) then a hit will always bust and if the initial score is large (16, 18, or 20) then a hit will always score 21 total. The opponent for this game will randomly hit or stay with equal probability. The optimal strategy for this game will result in winning about 54% of the hands.

Declarative Memory Related Issues

In this text we are going to investigate some of the most common problems which can arise when working with models that rely on the use of declarative retrievals when the subsymbolic calculations are enabled and thus activation affects the results. Unlike the models in previous modeling texts, the model which accompanies this text will not be performing any particular task. Instead, it is just a set of productions and declarative knowledge which allows us to see some of the issues which can occur when working with declarative memory and discuss how to determine what has happened and ways to address that in the model.

The Model Design

The model for this exercise, declarative-issues.lisp, is designed to perform a sequence of declarative retrievals and its declarative memory has been initialized with chunks that allow for the demonstration of specific retrieval situations. The **goal** and **imaginal** buffers will be used to control the sequencing of the productions and also to facilitate some of the retrieval situations. The productions are intended to fire in order by the number in their names, from p1 through p8. The specific details of the chunk-types and chunks involved will be described in the sections where it is meaningful below. The subsymbolic calculations for the model have been enabled using all of the activation equation's components. Most of the other parameters are left at their default values, or, in the case of those which are off by default like noise, have been set to values which reflect reasonable starting points based on the models from the tutorial and/or past ACT-R research.

Loading the Model

When we load the model we see the following warnings:

```
#|Warning: Production P3 has a condition for buffer RETRIEVAL with an isa that provides no tests. |#
#|Warning: Production P6 has a condition for buffer IMAGINAL with an isa that provides no tests. |#
```

Those are indications that there are no slots tested in the conditions for the indicated buffers even though a chunk-type has been declared with an isa like this one in P3:

```
  (p p3
     =goal>
       isa task-state
       state r3
     =retrieval>
       isa simple-value
    ==>
     =goal>
       state r4
     +retrieval>
       isa simple-fact
       attribute pink)
```

That warning is a reminder that the isa specification is not in and of itself a condition for the production. That is not a problem in this model because those productions only need to test that there is some chunk in the indicated buffer. So those can be safely ignored, and if you want to just turn off the warnings without fixing the productions you can add a setting of **nil** for the style-warnings to the model:

```
(sgp :style-warnings nil)
```

However, if you want to fix them then one thing that could be done is to just remove the isa and chunk-type specification from those buffer conditions. Alternatively, for the retrieval buffer condition in production **p3** we could avoid the warning by using a query to determine if there is a chunk in the buffer instead of using an =retrieval condition with no slots to test the buffer. We can't use a query for the **imaginal** buffer case in **p6** however because if a production makes a modification or modification request to a buffer as an action it must have an =<buffer> condition on the LHS of the production.

Since there are no other warnings or indicated issues which must be addressed in the model we can go right to trying to run it.

First Retrieval Request

Because there is no task associated with this model we will just use the ACT-R run command to run it. The first retrieval which this model makes specifies the chunk-type simple-value and indicates that it must have some value in the result slot (it is not **nil**) made by this production:

```
  (p p1
     ?goal>
       buffer empty
   ==>
     +goal>
       isa task-state
       state r2
     +retrieval>
       isa simple-value
      - result nil)
```

This is the definition of the simple-value chunk-type:

```
(chunk-type simple-value result)
```

and here are the initial chunks specified with that type which are placed into the model's declarative memory with add-dm:

```
(v1 isa simple-value result "true")
(v2 isa simple-value result "false")
(v3 isa simple-value result nil)
```

In addition to that, the initial activations of those chunks have been set as follows:

```
(set-base-levels (v1 1 -1500)
                 (v2 1 -1500)
                 (v3 1 -1500))
```

Those settings reflect one past occurrence for each chunk 1500 seconds ago. We will consider that as an unchangeable property of these chunks for purposes of addressing issues with their retrieval (assume that they were learned as the result of some previous actions which we consider to be working correctly). Based on the initial declarative memory and the request that **p1** makes we expect either chunk **v1** or **v2** to be retrieved. When we run the model however we find that it retrieves **chunk 2-1**:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.138   DECLARATIVE            RETRIEVED-CHUNK 2-1
     0.138   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL 2-1
     0.138   PROCEDURAL             CONFLICT-RESOLUTION
```

Chunk 2-1 is defined like this:

```
(2-1 isa math-fact arg1 two arg2 one result one operator subtract)
```

It is created using the chunk-type math-fact, but notice that it does have a slot named result. That's an important thing to remember about requests in productions. The chunk-type specified by isa is **not** part of the request. Thus, all that is being asked for in that request is a chunk which has a value in its result slot, which is true of **chunk 2-1** and all of the other math-fact chunks which are in the model's declarative memory. If we want to restrict the retrieval to only the simple-value chunks then we must specify something in the request which distinguishes them from other chunks.

Currently there are no other slots in the simple-value chunk type which we can use to distinguish it in the request. We could look at the other chunk-types with a slot named result and make the request to explicitly exclude them, for example since all of the math-fact chunks have values in the arg1 slot we could change the request to something like this:

```
+retrieval>
  isa simple-value
  - result nil
  arg1 nil
```

While that would work in this simple demo, it's not really a good solution because it is an arbitrary choice and if this were a more complex model it might be the case that the model could generate some math-fact chunk with an empty arg1 value at some point which could cause problems later. Doing that would also lead to these warnings when the model is loaded since the simple-value chunk-type doesn't have a slot named arg1:

```
#|Warning: Slot ARG1 invalid for type SIMPLE-VALUE but chunk-spec definition still created. |#
#|Warning: Production P1 action has invalid slot ARG1 for type SIMPLE-VALUE. |#
```

We could eliminate the isa specification in the request to avoid that warning:

```
+retrieval>
  - result nil
  arg1 nil
```

But the benefit of using the isa is for consistency and clarity in the model specification which we now lose. Therefore we should look for some way to make the simple-value chunks distinct, and there are basically two options. The first is to change the name of the slot in the simple-value chunk-type from result to something which isn't used elsewhere. The other is to add an additional unique slot to the chunk-type which we can set to a value to mark these chunks as distinct. For this model either of those seems like a reasonable choice. We will choose the second option, and add a slot named simple-value-chunk to the chunk-type and specify that slot with the value **t** in all of the chunks which we create using the simple-value chunk-type:

```
(chunk-type simple-value result simple-value-chunk)

(v1 isa simple-value result "true" simple-value-chunk t)
(v2 isa simple-value result "false" simple-value-chunk t)
(v3 isa simple-value result nil simple-value-chunk t)
```

A potential issue with adding slots to a chunk is that if the model is using spreading activation then having additional slots with chunks in them will affect the fan of the value placed into the slot and the amount of activation which spreads from those chunks with the additional slot if they are in a buffer. To avoid that one can use the special value **t** for the slot because **t** is not a chunk. It represents the Lisp value for true which is the opposite of the Lisp value **nil**, and since **nil** is used to indicate slots that don't exist **t** can be used to indicate slots that do exist without using a chunk to do so.

Now we also need to change the request in production **p1** to specify that slot and value:

```
     +retrieval>
       isa simple-value
     - result nil
       simple-value-chunk t
```

Later in the tutorial we will introduce a way to handle some of that additional specification with a slot that has a fixed value automatically, but for now we will just make the changes directly to the chunks and the request in the production.

After making that change when we run the model we get the following result:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     1.050   DECLARATIVE            RETRIEVAL-FAILURE
     1.050   PROCEDURAL             CONFLICT-RESOLUTION
     1.050   ------                 Stopped because no events left to process
```

The model stopped because it failed to retrieve a chunk which prevents any further productions from firing. We now need to figure out why it didn't retrieve either of the simple-value chunks we were expecting it to retrieve. There are several ways one could go about doing that, and we will describe several of them here.

One option, is to use the ACT-R command whynot-dm (or whynot_dm in Python) which is similar to the why-not command for productions. It reports on what happened with the last retrieval request which was made. Calling it with no parameters will print out the last retrieval request which was made and then for each chunk in declarative memory print the chunk, its parameters, and an indication of why it wasn't retrieved for that request. Displaying the information for all chunks however is not typically useful because there could be a lot of chunks in declarative memory which makes it difficult to find the important ones, but it is also possible to pass it the names of chunks and it will only print the details for those chunks. Since our model has stopped and the last retrieval request is the one we are interested in we can use that command like this to get the information about the chunks v1, v2, and v3:

```
? (whynot-dm v1 v2 v3)
```

```
>>> actr.whynot_dm('v1','v2','v3')
```

Those print out the following information:

```
Retrieval request made at time 0.050:
 -  RESULT NIL
    SIMPLE-VALUE-CHUNK T

V1
   RESULT  "true"
   SIMPLE-VALUE-CHUNK  T

Declarative parameters for chunk V1:
 :Activation -3.026
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V1 . 2.0))
 :Similarities ((V1 . 0.0))
 :Last-Retrieval-Activation -3.142
 :Last-Retrieval-Time  0.050

V1 matched the request
V1 was below the retrieval threshold 0.0
```

```
V2
   RESULT  "false"
   SIMPLE-VALUE-CHUNK  T

Declarative parameters for chunk V2:
 :Activation -2.437
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V2 . 2.0))
 :Similarities ((V2 . 0.0))
 :Last-Retrieval-Activation -2.725
 :Last-Retrieval-Time  0.050

V2 matched the request
V2 was below the retrieval threshold 0.0

V3
   SIMPLE-VALUE-CHUNK  T

Declarative parameters for chunk V3:
 :Activation -1.679
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V3 . 2.0))
 :Similarities ((V3 . 0.0))

V3 did not match the request
```

There we see that v1 and v2 did match the request, but they were below the retrieval threshold.

Another mechanism for investigating issues with retrievals is to turn on the activation trace in the model and run it again. That is done by setting the :act parameter to a non-nil value. In the unit texts it was set to t, but like the trace-detail parameter, it can also be set to values of high, medium, or low to control how much detail is shown. We will run the model with each value below to show the differences in information provided.

We could make that change in the model file, save the model, and then load it, but since the model itself hasn't changed we don't really need to perform those steps. Instead, we can reset the model (using either the reset command or the Reset button in the Environment) and then just call sgp from the ACT-R prompt to change the parameter value before running it or from the Python interface the function set_parameter_value could be used which takes a string with the name of the parameter and a string of the value in this case:

```
? (sgp :act medium)
```

```
>>> actr.set_parameter_value(':act','medium')
```

Changing the parameters interactively like that can be a convenient way to debug a model, but may not always be possible; particularly if there is additional code involved which is responsible for resetting and running the model. Here is the activation trace with a value of t (which is the same as a value of high and could also be set using True through the Python command):

```
     0.050   DECLARATIVE            start-retrieval
Computing activation for chunk V1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9634798
Total base-level: -2.9634798
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: "true"
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.17835411
Adding permanent noise 0.0
Chunk V1 has an activation of: -3.1418338
Chunk V1 has the current best activation -3.1418338
Computing activation for chunk V2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9634798
Total base-level: -2.9634798
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: "false"
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.23814109
Adding permanent noise 0.0
Chunk V2 has an activation of: -2.7253387
Chunk V2 is now the current best with activation -2.7253387
No chunk above the retrieval threshold: 0.0
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
```

Here is what we get when it is set to medium:

```
     0.050   DECLARATIVE            start-retrieval
Computing activation for chunk V1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
```

```
base-level value: -2.9634798
Total base-level: -2.9634798
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: "true"
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.17835411
Adding permanent noise 0.0
Chunk V1 has an activation of: -3.1418338
Chunk V1 has the current best activation -3.1418338
Computing activation for chunk V2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9634798
Total base-level: -2.9634798
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: "false"
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.23814109
Adding permanent noise 0.0
Chunk V2 has an activation of: -2.7253387
Chunk V2 is now the current best with activation -2.7253387
No chunk above the retrieval threshold: 0.0
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
```

Both of those traces display all of the details of the activation calculations for the chunks which matched the request and in this case there isn't any difference between them. The difference would show up if there were chunks in declarative memory which had the slots specified in the request but which failed on some of the constraints specified, for example, if the :recently-retrieved request parameter had been specified. In those situations the high detail trace will also include a line for each of the non-matching chunks whereas the medium detail trace will not show the chunks that didn't match the request.

Here is what we get when it is set to low:

```
     0.050   DECLARATIVE            start-retrieval
Chunk V1 has an activation of: -3.1418338
Chunk V2 has an activation of: -2.7253387
No chunk above the retrieval threshold: 0.0
```

```
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
```

When set to low only the final activation values are shown instead of all the details.

Before looking at the details of that retrieval, we will introduce another command which can also be useful for interactively debugging a model: with-parameters. Instead of changing the value of a parameter with sgp one can use the with-parameters command at the ACT-R prompt to temporarily set parameter values and evaluate some other commands (there is no equivalent to with-parameters available through the remote interface at this time). Thus, instead of setting the activation trace to low and then running the model we could have done the following to set both the activation trace and the standard trace detail to low and then run with those settings:

```
? (with-parameters (:act low :trace-detail low)
    (run 10))
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
Chunk V1 has an activation of: -3.1418338
Chunk V2 has an activation of: -2.7253387
No chunk above the retrieval threshold: 0.0
     1.050   DECLARATIVE            RETRIEVAL-FAILURE
     1.050   ------                 Stopped because no events left to process
```

After the with-parameters call is done the parameter values will be automatically returned to the values that they had previously.

As for the retrieval, regardless of which trace we look at, the critical line is the last one:

```
No chunk above the retrieval threshold: 0.0
```

No chunk is retrieved because they all have activations below the current retrieval threshold, which is also what we found from whynot-dm.

Another way we could have investigated the chunks' activations is by using the sdp command to see the current declarative parameters for each chunk. That will include the current activation as well as the activation it had the last time it was attempted to be retrieved. When the model stopped we could call sdp to print out all of those chunks and their parameters like this:

```
? (sdp v1 v2 v3)

>>> actr.sdp('v1','v2','v3')
```

Which would result in the following output:

```
Declarative parameters for chunk V1:
 :Activation -3.387
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V1 . 2.0))
 :Similarities ((V1 . 0.0))
 :Last-Retrieval-Activation -3.142
 :Last-Retrieval-Time  0.050
Declarative parameters for chunk V2:
 :Activation -2.527
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V2 . 2.0))
 :Similarities ((V2 . 0.0))
 :Last-Retrieval-Activation -2.725
 :Last-Retrieval-Time  0.050
Declarative parameters for chunk V3:
 :Activation -1.626
 :Permanent-Noise  0.000
 :Base-Level -2.964
 :Creation-Time -1500.000
 :Reference-Count  1
 :Source-Spread  0.000
 :Sjis ((V3 . 2.0))
 :Similarities ((V3 . 0.0))
```

That doesn't completely explain why things failed, but if we knew the retrieval threshold in advance and suspected that to be the problem then sdp may have been useful without having to run the model again to get the activation trace. That same information is shown in the Declarative tool of the ACT-R Environment, so that could also be used to investigate the chunks' activations, and there is a "Why not?" button that can be used in the Declarative tool as well. It is also possible to record the retrieval information for viewing after the run if desired and that will be described in a later section.

Now that we know the problem is that the chunk activations are below the retrieval threshold there are basically two ways to address that: either lower the retrieval threshold or increase the activation of those chunks in some way. Decreasing the threshold is an easy thing to do since all it takes is adding the parameter setting to the model. Changing the activation of the chunks can be accomplished in several ways, but given our constraint of not adjusting their histories limits us to essentially two options. One way to increase their activations would be to use the :blc (base-level constant) parameter to add a fixed value to all chunk activations. Another way would be to add additional information to those chunks which could provide a way for spreading activation to increase their activations.

As is usually the case, there is no one "right" answer as to how to fix this. A modeler will have to consider his or her theory as to how people are performing the task, any data which is available, and the possible implications of making the change to other components of the model. For this model we do not have a theory or data since we are not modeling a real task. The effects on other parts of the model are also not relevant at this point for the same reason. So, since we don't have any reason to pick one option over the others, for the purpose of the exercise we will set the retrieval threshold lower and then deal with any possible consequences this has later on in the model. Before doing so we will look at some of the potential issues from the other changes which one might want to consider.

If the :blc parameter is adjusted that will affect the activation of all of the chunks which are retrieved by the model. Since the time to retrieve a chunk depends on its activation, not only will setting :blc affect whether a chunk is retrieved but also how long it will take. Thus, that may then necessitate the adjustments of other parameters as well to keep the response times in line with the data if that is important. However, if response time is not important to the data being modeled, then adjusting :blc might be a simple way to help ensure that chunks exceed the retrieval threshold and are retrieved quickly (since a higher activation will mean faster retrieval).

Using spreading activation to increase the activation of the chunks might be a plausible mechanism for the task. If the knowledge is prespecified for the model, like we are doing here, then it may be easy to add some additional context to that information to facilitate spreading activation. For example, although the grouped recall example from unit 5 didn't use spreading activation it did have a chunk which represented the current list itself as a member of the group chunks which could have been used for that purpose. If the model is learning the chunks on its own however then one needs to have a way for the model to generate its own context. One way that is often done is to include the model's current goal or imaginal chunk as a slot value in the memories it creates. That way, each new goal or problem representation provides a particular context. The biggest downside to using this approach is primarily the additional complexity it requires in the model. One now has to have that context information available to spread the activation in a slot of a buffer at the time of retrieval, and it may require additional retrievals to remember past contexts as well as the needed information.

Now that we've covered some of the alternatives, we will set the threshold lower and see how that affects the model. The first question is how low do we want to set it? To really answer that we need to decide how likely we want the model to fail to retrieve a chunk when there is one which can be retrieved, and to determine that we need to know what the activation of the chunks are and how much noise there is in the activations. Knowing the activation and noise value we can compute the recall probability for a chunk using the equation presented in unit 4. However, right now for this model, we just want it to succeed every time. Thus we want to pick a value significantly lower than the activation of the chunks involved. From the activation trace we see that the chunks involved have activations of about -3.14 and -2.73. Therefore if we set the threshold to -10 that should be sufficient since with the model's activation noise set to .25 the recall probability for a chunk with an activation of -3.14 will be extremely close to 1.0 with that threshold.

Before changing the model and reloading it we can actually test that new value now. First we set the retrieval threshold to the value we want:

```
? (sgp :rt -10)

>>> actr.set_parameter_value(':rt',-10)
```

Then we can use the command simulate-retrieval-request to see what would happen if we were to make that same retrieval request now. The parameters to that command are the slots specified for the request just as they would appear in a production after the +retrieval>:

```
? (simulate-retrieval-request - result nil simple-value-chunk t)

>>> actr.simulate_retrieval_request('-','result','nil',
                                    'simple-value-chunk','t')
```

For the Python version we could also rely on the fact that the Lisp nil corresponds to None in Python and the Lisp t is equivalent to True in Python and use those values instead of strings with the Lisp values:

```
>>> actr.simulate_retrieval_request('-','result',None,
                                    'simple-value-chunk',True)
```

Those show the following output:

```
Chunk V1 has the current best activation -2.874935
Chunk V2 has activation -2.887749
Chunk V1 with activation -2.874935 is the best
```

The output from calling that command is the low detail activation trace which would result if that request were made at the current time. If we set the retrieval threshold back to 0 and try it again we can see that it indicates the retrieval failure as the trace above does:

```
Chunk V1 has the current best activation -2.874935
Chunk V2 has activation -2.887749
No chunk above the retrieval threshold: 0.0
```

So now that we know a retrieval threshold of -10 will work for the model to be able to retrieve those chunks we will add that additional setting to the sgp call in the model:

```
(sgp :esc t :v t :bll .5 :ans .25 :mas 3 :mp 10 :rt -10)
```

We need to save that change, load the model, and run it again. Here is the trace that we get now if we run for 10 seconds:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    10.000   ------                 Stopped because time limit reached
```

The model stopped because it reached the end of the 10 seconds we asked it to run and it did not retrieve the chunk in that time. There are a couple of things to consider now. First, did the model successfully retrieve the chunk? If so, why did it take at least 10 seconds to complete? Then we have to decide if that amount of time is reasonable for the model in performing this task.

As to whether or not the model successfully retrieved the chunk we have several options for testing that. First, we could just run the model some more until we find either a successful retrieval or a retrieval failure. In this case that would work just fine because there are no other productions that could fire to interfere with that. Alternatively, we could reset it and enable the activation trace so that we have the details of what happened. Again, for this model that is not a difficult option since this happens early in the run and because there is no task which is running the model it's easy to stop it where we want without having to use the Stepper tool in the ACT-R Environment. Instead of using those however we are going to introduce a new command that can also be helpful in situations like this. That command is called mp-show-queue and it allows us to look ahead in time to see what events the model is scheduled to do in the future without actually running it. It takes no parameters and would be called like this:

```
? (mp-show-queue)
>>> actr.mp_show_queue()
```

This is the output that it shows:

```
Events in the queue:
    15.312   DECLARATIVE            RETRIEVED-CHUNK V2
    15.312   PROCEDURAL             CONFLICT-RESOLUTION
```

That shows us that the model will complete the retrieval at time 15.312 and then perform another conflict-resolution action. Just because we see an event scheduled to occur at some future time with mp-show-queue however does not mean that we will necessarily see that same action in the trace if we continue to run the model. That's because things can happen to change the situation before that time arrives. Thus, looking ahead at the model's actions like that can be very useful in situations where a delayed action, like a retrieval completion, could be superseded by a new retrieval. For example, if another production were to fire and make a retrieval request at time 11.0 that would interrupt the ongoing retrieval and we would not actually see that retrieved-chunk action at time 15.312 if we were to run the model. That doesn't happen in this model, and we would have seen the same results if we were to just run it that long:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    15.312   DECLARATIVE            RETRIEVED-CHUNK V2
    15.312   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    15.312   PROCEDURAL             CONFLICT-RESOLUTION
    15.320   ------                 Stopped because time limit reached
```

Although it's not really necessary here, mp-show-queue is a useful tool to know about and can be very helpful in some situations.

Now that we know the retrieval did succeed we should consider the time that it took to do so. As shown in unit 4 the time it takes to retrieve a chunk depends on its activation by the equation:

$$Time = Fe^{-A}$$

The activation of our chunk is about -2.73 and the value of F is the latency factor parameter (:lf) which defaults to 1. Since we don't set that parameter in our model the time to retrieve the chunk should be about $e^{2.73} \approx$ 15.3 seconds, which is what we see in the trace.

If the model were performing a real task, particularly if we had data for comparison, we would want to consider if a retrieval of that length is acceptable for the model, and if not, what we should do about it. Since this model is not performing any particular task we don't really have any basis for judging the length of that retrieval time, but we can still consider how we would change it if we wanted to. Based on the equation for the retrieval time there are two things we can do to affect the time. The first would be to change the activation of the chunk, and that could be done in the same ways as were discussed previously. The other option would be to change the :lf parameter. The thing to keep in mind when changing :lf is that it will affect all of the retrievals which the model performs. As an example we will change :lf for this model to decrease the time it takes to complete retrievals by setting it to .8:

```
  (sgp :esc t :v t :bll .5 :ans .25 :mas 2 :mp 10 :rt -10 :lf .8)
```

Saving that change and then reloading the model here is what the trace looks like now with the activation trace set to low to show that while the time of the retrieval has changed the activations of the chunks are the same as they were previously:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
Chunk V1 has an activation of: -3.1418338
Chunk V2 has an activation of: -2.7253387
Chunk V2 with activation -2.7253387 is the best
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.259   DECLARATIVE            RETRIEVED-CHUNK V2
    12.259   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    12.259   PROCEDURAL             CONFLICT-RESOLUTION
    ...
```

Second Retrieval Request

The second retrieval request the model makes is very similar to the first one. Except that there are no constraints placed on the request this time:

```
  (p p2
     =goal>
       isa task-state
       state r2
     =retrieval>
   ==>
     =goal>
       state r3
     +retrieval>
       isa simple-value)
```

Assuming that we want to restrict the request to the simple-value chunks we should change that request to specify the simple-value-chunk slot before trying to run it as we did for production p1:

```
(p p2
   =goal>
     isa task-state
     state r2
   =retrieval>
 ==>
   =goal>
     state r3
   +retrieval>
     isa simple-value
     simple-value-chunk t)
```

Here is a portion of the trace from running the model for this retrieval with the activation trace enabled:

```
...
    12.309   PROCEDURAL             PRODUCTION-FIRED P2
    12.309   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    12.309   DECLARATIVE            start-retrieval
Computing activation for chunk V1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9675493
Total base-level: -2.9675493
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.08887799
Adding permanent noise 0.0
Chunk V1 has an activation of: -2.8786714
Chunk V1 has the current best activation -2.8786714
Computing activation for chunk V2
Computing base-level
Starting with blc: 0.0
Computing base-level from 2 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.2744021
Total base-level: -2.2744021
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
```

```
Adding transient noise 0.07606379
Adding permanent noise 0.0
Chunk V2 has an activation of: -2.1983383
Chunk V2 is now the current best with activation -2.1983383
Computing activation for chunk V3
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9675493
Total base-level: -2.9675493
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.026406968
Adding permanent noise 0.0
Chunk V3 has an activation of: -2.9939563
Chunk V2 with activation -2.1983383 is the best
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    19.517   DECLARATIVE            RETRIEVED-CHUNK V2
    19.517   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    19.517   PROCEDURAL             CONFLICT-RESOLUTION
...
```

We see that it retrieves chunk v2 again this time, and that this retrieval is faster than the last time requiring only a little more than 7 seconds instead of around 12. That is because the chunk has an extra reference now since it was retrieved previously and thus it has a higher base-level activation than the other two simple-value chunks.

That is the expected result of base-level learning, activation of the chunk increases with practice which makes it more likely to be retrieved and faster when it is. However, there is a potential issue that can arise with respect to base-level learning. The issue to be wary of is that when using requests to declarative memory with few constraints a single chunk may come to dominate and always be retrieved. In some situations that may be desirable, but in other situations one may not want one chunk to dominate like that.

If one does not want a single chunk to dominate, but can't provide additional constraints in the request or change the context to affect the other components of the activation equation, then one may need to take advantage of the declarative finsts which were described in unit 3 of the tutorial. They can be used to suppress the retrieval of a chunk which has been recently retrieved so that a single chunk is not retrieved repeatedly. To do so one needs to add the :recently-retrieved request parameter to the request which is made to the retrieval buffer with a value of nil. That will then remove chunks which are currently marked with a declarative finst from those considered for that request. For demonstration purposes we will make that change to the request made in p2 and see the difference. Here is the new version of p2:

```
(p p2
   =goal>
     isa task-state
```

```
        state r2
      =retrieval>
    ==>
      =goal>
        state r3
      +retrieval>
        isa simple-value
        simple-value-chunk t
        :recently-retrieved nil)
```

and here is the trace showing that retrieval now:

```
...
    12.259   PROCEDURAL             CONFLICT-RESOLUTION
    12.309   PROCEDURAL             PRODUCTION-FIRED P2
    12.309   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    12.309   DECLARATIVE            start-retrieval
Removing recently retrieved chunks:
V2
Computing activation for chunk V1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9675493
Total base-level: -2.9675493
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.08887799
Adding permanent noise 0.0
Chunk V1 has an activation of: -2.8786714
Chunk V1 has the current best activation -2.8786714
Computing activation for chunk V3
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references ()
  creation time: -1500.000 decay: 0.5  Optimized-learning: T
base-level value: -2.9675493
Total base-level: -2.9675493
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot SIMPLE-VALUE-CHUNK
  Requested: = T  Chunk's slot value: T
  similarity: 0.0
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.07606379
Adding permanent noise 0.0
Chunk V3 has an activation of: -2.8914855
Chunk V1 with activation -2.8786714 is the best
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    26.541   DECLARATIVE            RETRIEVED-CHUNK V1
    26.541   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V1
    26.541   PROCEDURAL             CONFLICT-RESOLUTION
...
```

This time chunk v1 was retrieved because chunk v2 was removed from consideration since it has a declarative finst on it from the previous retrieval. If one wants to see which chunks are currently marked with a declarative finst the print-dm-finsts command can be used. It takes no parameters:

```
? (print-dm-finsts)
```

```
>>> actr.print_dm_finsts()
```

Here is the output of that after the run shown above:

```
Chunk name     Time Stamp
-------------------------
V1                26.541
```

Note that only v1 currently has a finst on it at this time because the default duration of declarative finsts is 3 seconds and more than that amount of time has passed since v2 was retrieved. If we instead check at time 12.3, after the first retrieval has completed and just before the second request is made, we will see that v2 does have a declarative finst on it:

```
...
    12.309   DECLARATIVE            start-retrieval
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    12.500   ------                 Stopped because time limit reached

? (print-dm-finsts)

Chunk name     Time Stamp
-------------------------
V2                12.259
```

If one cannot use finsts or some other means of preventing the retrieval of the same chunk repeatedly (extra constraints in the request or spreading activation with values that would favor different chunks for example), but needs to avoid the issue of a single chunk becoming dominant there are some additional options available if one adds optional components to the ACT-R system. Distributed with the ACT-R source code are several extensions which have been developed for ACT-R. Those optional components are found in the extras directory of the distribution and each one is found in a separate subdirectory. Two of the extensions available affect the base-level learning equation and may help avoid the dominant chunk problem. Those two particular extras are in the spacing-effect and base-level-inhibition directories. Because they are not part of the standard ACT-R system we will not be describing them in the tutorial materials, but one can find details and instructions on their use in the extra files provided.

Third Retrieval Request

The next retrieval request this model makes declares the chunk-type simple-fact which is defined like this:

```
(chunk-type simple-fact item attribute)
```

The production which makes the request is this one:

```
  (p p3
     =goal>
       isa task-state
       state r3
     =retrieval>
   ==>
     =goal>
       state r4
     +retrieval>
       isa simple-fact
       attribute pink)
```

The request specifies that a chunk with an attribute slot value of pink should be retrieved from declarative memory.

Here are the simple-fact chunks which the model starts with in its declarative memory from the add-dm command in the model definition:

```
   (f1 isa simple-fact item sky attribute blue)
   (f2 isa simple-fact item rose attribute red)
   (f3 isa simple-fact item grass attribute green)
```

There are no base-level activation values set for those chunks, thus for now they will each have one reference which occurs at time 0 because that's when they are added to the model's declarative memory.

Notice that none of the chunks have an attribute which matches the request which is being made, but this model does have partial matching enabled so perhaps one of them will still be retrieved. We will run it to find out what happens, and here is the result of running the model for 30 seconds:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.259   DECLARATIVE            RETRIEVED-CHUNK V2
    12.259   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    12.259   PROCEDURAL             CONFLICT-RESOLUTION
    12.309   PROCEDURAL             PRODUCTION-FIRED P2
    12.309   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    12.309   DECLARATIVE            start-retrieval
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    26.541   DECLARATIVE            RETRIEVED-CHUNK V1
    26.541   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V1
    26.541   PROCEDURAL             CONFLICT-RESOLUTION
    26.591   PROCEDURAL             PRODUCTION-FIRED P3
    26.591   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.591   DECLARATIVE            start-retrieval
    26.591   PROCEDURAL             CONFLICT-RESOLUTION
    30.000   ------                 Stopped because time limit reached
```

It has not retrieved a chunk at that point. Instead of continuing to run, we will again look ahead with mp-show-queue:

```
Events in the queue:
 17647.763   DECLARATIVE            RETRIEVAL-FAILURE
 17647.763   PROCEDURAL             CONFLICT-RESOLUTION
```

That shows that the model has failed to retrieve a chunk as a result of that request and that it is going to take over 17000 seconds while trying. First we will look at why the model failed to retrieve a chunk and then we will consider why it takes so long when it fails.

Turning on the activation trace and running again we see these activation computations for this request:

```
    26.591   DECLARATIVE            start-retrieval
Computing activation for chunk F1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9471392
Total base-level: -0.9471392
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ATTRIBUTE
  Requested: = PINK  Chunk's slot value: BLUE
  similarity: -1.0
  effective similarity value is -10.0
Total similarity score -10.0
Adding transient noise -0.026406968
Adding permanent noise 0.0
Chunk F1 has an activation of: -10.973546
Chunk F1 has the current best activation -10.973546
Computing activation for chunk F2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9471392
Total base-level: -0.9471392
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ATTRIBUTE
  Requested: = PINK  Chunk's slot value: RED
  similarity: -1.0
  effective similarity value is -10.0
Total similarity score -10.0
Adding transient noise -0.42355087
Adding permanent noise 0.0
Chunk F2 has an activation of: -11.370689
Computing activation for chunk F3
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9471392
Total base-level: -0.9471392
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ATTRIBUTE
  Requested: = PINK  Chunk's slot value: GREEN
  similarity: -1.0
  effective similarity value is -10.0
Total similarity score -10.0
Adding transient noise 0.43639642
Adding permanent noise 0.0
Chunk F3 has an activation of: -10.510742
Chunk F3 is now the current best with activation -10.510742
No chunk above the retrieval threshold: -10.0
```

Each chunk has a base-level activation of around -.947 but then loses more activation because of the mismatch to the requested slot value. Because we have not set any similarities in this model they all default to the worst mismatch value of -1, and because the :mp parameter is set to 10 in the sgp settings for the model, each chunk then has 10 * -1 added to its activation for a total of around -10.947 before noise is added. None of the noise values are large enough to bring a chunk above the retrieval threshold which we set previously at -10. Thus, the model fails to retrieve any of them.

When there is a retrieval failure the time that it will take uses the same equation as for a successful retrieval, except that the retrieval threshold is used instead of a chunk's activation. Thus, with our current parameter settings a retrieval failure will take .8*e^{10} seconds, which is almost 5 hours of simulated time in which the model sits trying to retrieve a chunk before it finally fails. A delay of that long is likely to be unacceptable in any reasonable model, so we need to decide what to do about it. Since we don't have a task to guide us, we're free to explore many possible alternatives for how to address that. First we will consider options for making it more likely that one of the existing chunks will be retrieved, and then we will consider how we can handle things if the model does still have retrieval failures.

If we want one of these chunks to be retrieved then we need to either raise their activations or lower the retrieval threshold again. If we were to lower the threshold then the retrieval of these chunks would take as long as the retrieval failure did now, and the lower we make the threshold the worse things are with respect to the time it takes when there is a failure. So we will not consider that a good choice at this point. Instead, we will look at how we can raise the activation of these chunks.

There are three components to the activation equation: base-level, spreading activation, and partial matching and each of those provides opportunities to increase the chunk's activation. We discussed the spreading activation change with respect to the first retrieval, and again will not choose to modify the model in that way for this retrieval. Instead we will look at the options for affecting the other two components of activation.

For the chunks’ base-levels we again have the option of setting the :blc parameter to increase every chunk’s base-level. Having seen the problem of very long retrieval times for low threshold and activations, we may want to consider doing that so we can shift things to a level that produces more reasonable times, but we will come back to that in a later section. Another option which we have available for changing the base-level of these chunks is to explicitly set their base-level activations using the set-base-levels command. For the first retrieval we had considered the base-level settings of the chunks involved as fixed values, but for these chunks we will not. Thus, we could set their creation time and number of references to values that provide sufficiently large base-level activations. When choosing to modify the base-levels of chunks one should take into account what those chunks represent and what the values for creation time and references mean for the model. For example, if the chunks represent information which the model would not have learned prior to doing the task then their creation times probably shouldn’t be any earlier than when the model would have started the task, and similarly they shouldn’t have more references than would be reasonable in that time. However, if the chunks are representing background knowledge that the model had long before doing the task then a much earlier creation time and larger reference count are warranted. For background knowledge of that nature it’s often difficult to determine what would be an appropriate creation time and number of references so more arbitrary values are used to achieve an appropriate activation for the task.

Setting the base-levels of the chunks for this retrieval seems like it would be a reasonable thing to do since they are representing facts one would assume a person would have learned long before the task and which have been encountered frequently. Thus, as a first step we will give those chunks a strong history with this setting in the model:

```
(set-base-levels (f1 1000 -10000)
                 (f2 1000 -10000)
                 (f3 1000 -10000))
```

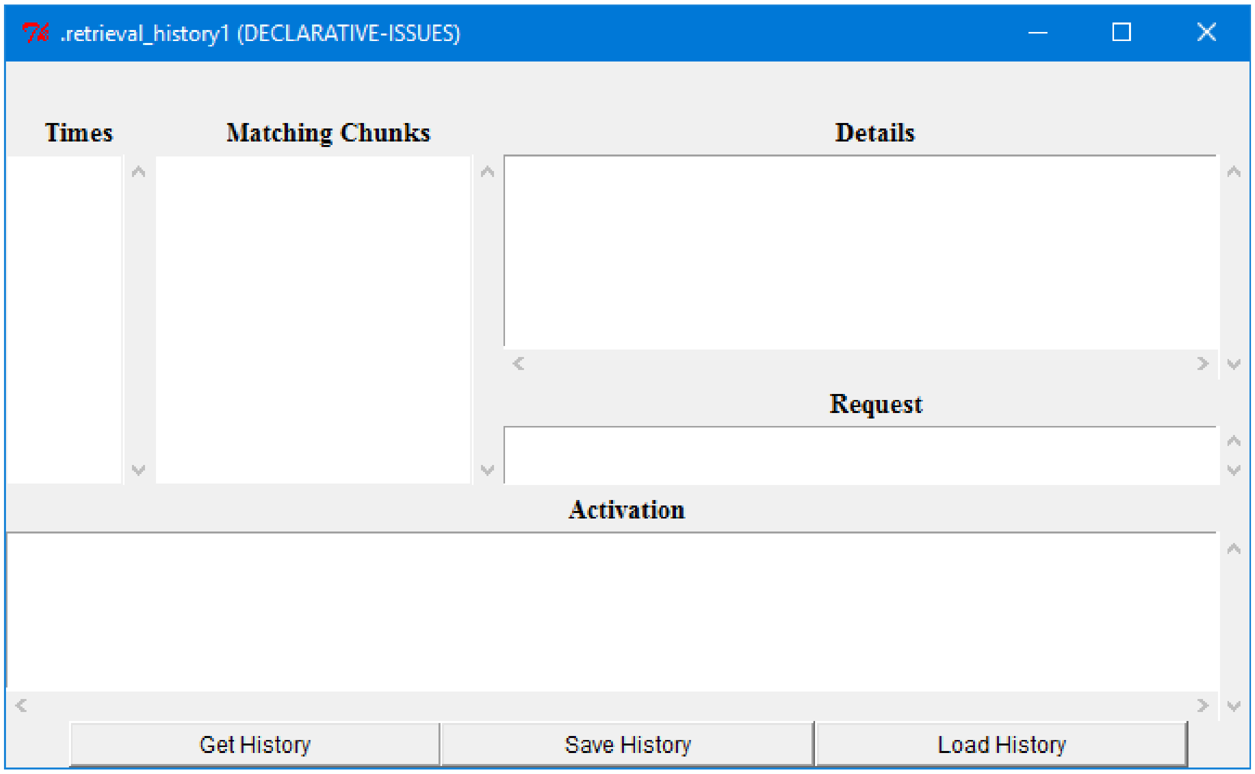
That gives the chunks a history which, while in absolute terms is probably not likely (having only learned the chunk 10000 seconds ago and having used it 1000 times since then), but may be sufficient to provide a fairly stable and relatively strong base-level activation over the course of the current task.

Running the model now we see that it will succeed in retrieving one of those chunks, but it will take a significant amount of time when checked with mp-show-queue:

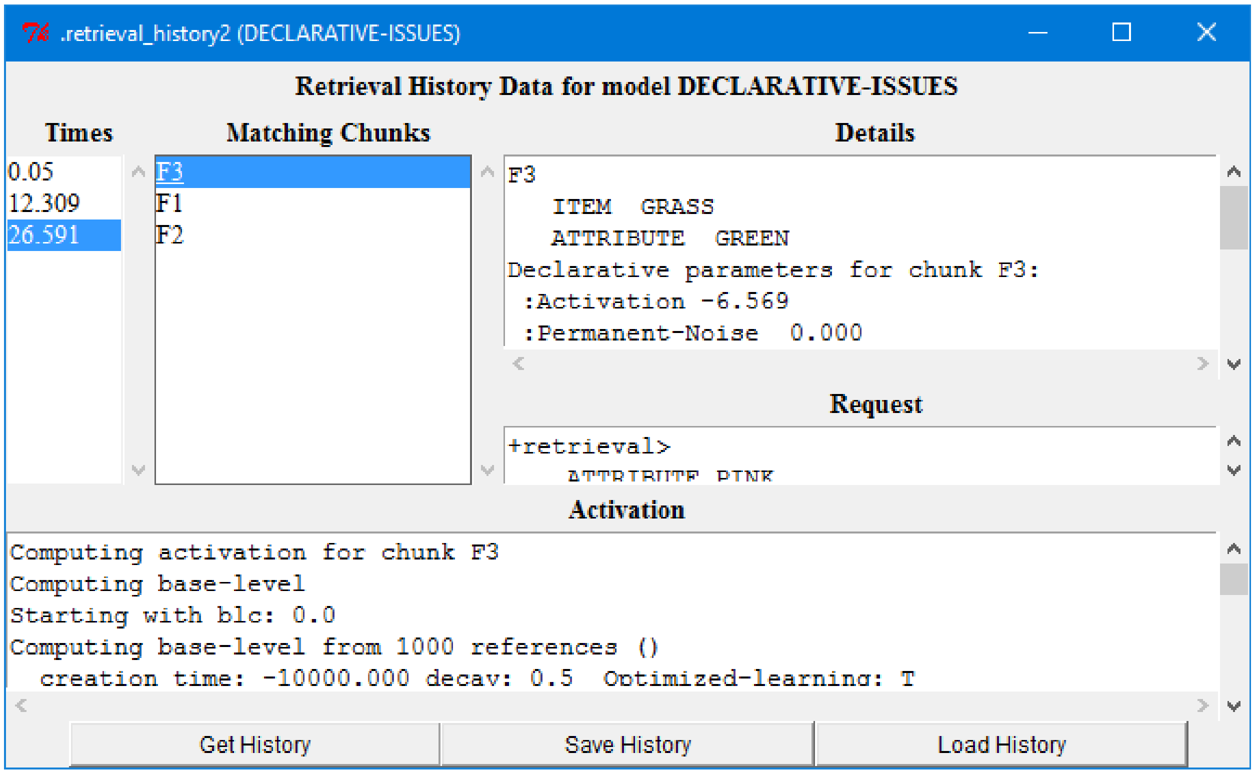
```
...
    26.591   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.591   DECLARATIVE            start-retrieval
    26.591   PROCEDURAL             CONFLICT-RESOLUTION
    28.000   ------                 Stopped because time limit reached


Events in the queue:
   596.830   DECLARATIVE            RETRIEVED-CHUNK F3
   596.830   PROCEDURAL             CONFLICT-RESOLUTION
```

We could turn the activation trace on to see why that happens, but instead we will introduce another tool available in the ACT-R Environment. If we open the "Retrieval History" tool before we start running the model it will open a window like the one shown below and start recording the activation information while the model runs.



When the model is done running, clicking the "Get History" button on the bottom left of that window will cause the "Times" section to display all the times in the model run at which a retrieval request was made. Picking one of those times will then cause all of the chunks which matched the request to be displayed in the "Matching Chunks" section and the request which was made to be shown in the "Request" section. The top chunk in the list will be the one which was selected for retrieval, or it will be the symbol :retrieval-failure if there was no matching chunk with an activation above the retrieval threshold. Picking one of those chunks will then cause the "Details" section to display the chunk and its parameters at that time and the "Activation" section to display the complete activation trace for how that chunk's activation was computed for that retrieval request at that time. Here is what we see for the chunk F3 which is the one that will be retrieved for the request made at time 26.591:



Retrieval History Data for model DECLARATIVE-ISSUES

Times	Matching Chunks
0.05	F3
12.309	F1
26.591	F2

Details

```
F3
   ITEM  GRASS
   ATTRIBUTE  GREEN
Declarative parameters for chunk F3:
 :Activation -6.569
 :Permanent-Noise  0.000
```

Request

```
+retrieval>
    ATTRIBUTE PINK
```

Activation

```
Computing activation for chunk F3
Computing base-level
Starting with blc: 0.0
Computing base-level from 1000 references ()
  creation time: -10000.000 decay: 0.5  Optimized-learning: T
```

and here is all of the information from the activation section:

```
Computing activation for chunk F3
Computing base-level
Starting with blc: 0.0
Computing base-level from 1000 references ()
  creation time: -10000.000 decay: 0.5  Optimized-learning: T
base-level value: 2.9944046
Total base-level: 2.9944046
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ATTRIBUTE
  Requested: = PINK  Chunk's slot value: GREEN
  similarity: -1.0
  effective similarity value is -10.0
Total similarity score -10.0
Adding transient noise 0.43639642
Adding permanent noise 0.0
Chunk F3 has an activation of: -6.5691986
```

Before looking at those results we will first mention that one can also get that information without using the ACT-R Environment. To enable the recording of that information either the :sact parameter must be set to t before the model runs or the record-history command must be used to indicate that the “retrieval-history” should be recorded:

```
? (record-history "retrieval-history")
```

```
>>> actr.record_history('retrieval-history')
```

To access that saved information one can get the raw data using the get-history-data command, but we will not describe the format of that data here other than to say that it is a string which contains the data in a JSON format:

```
? (get-history-data "retrieval-history")

>>> actr.get_history_data('retrieval-history')
```

Probably more useful are the commands for printing the trace information. The saved-activation-history command can be used to get all the times and chunks for which the activation information has been recorded. It returns a list of lists where each sublist starts with the time in milliseconds at which a request was made and the remainder of the list are the chunks which matched the request at that time.

```
? (saved-activation-history)
((50 V1 V2) (12309 V1 V3) (26591 F1 F2 F3))

>>> actr.saved_activation_history()
[[50, 'V1', 'V2'], [12309, 'V1', 'V3'], [26591, 'F1', 'F2', 'F3']]
```

With that information one can use the print-activation-trace or print-chunk-activation-trace command to print all of the activation trace information at a given time, or only the activation trace information for a particular chunk which would look like this to get the trace at time 26591 and the trace for the chunk f3 at that time:

```
? (print-activation-trace 26591)

>>> actr.print_activation_trace(26591)


? (print-chunk-activation-trace f3 26591)

>>> actr.print_chunk_activation_trace('f3',26591)
```

However we get that information, looking at the trace we see that the base-level of that chunk looks strong (it would take less than 50ms to retrieve a chunk with an activation of ~3 with the current parameter settings) but it takes a big negative hit from the partial matching component of the equation. The other two chunks show a similar pattern, and it's only the noise value which differentiates them making f3 the one to be retrieved. Since the base-level looks strong we will now consider adjusting the partial matching component of the activations.

For partial matching there are three things to consider: what values are requested in the slots of the retrieval request, what the similarities are between those requested values and the values in the slots of the chunks in declarative memory, and the mismatch penalty parameter (:mp) value. We will consider some of the issues related to each of those, and then decide what changes, if any, to make to the model.

Choosing how specific to make the retrieval request can be important in determining how likely that request is to succeed when using partial matching because each additional constraint in the request is an added opportunity to decrease the activation of the target chunks. That is because partial matching provides a penalty to chunks which do not match the request – it does not increase the activation of chunks which do (at least not under default and recommended parameter settings). If there is a chunk which matches the request then the specificity of the request doesn't really matter since there will be no penalty to that matching chunk. However, when the model is making requests in situations where it may not have a perfectly matching chunk (for instance in the 1-hit blackjack game from the unit 5 assignment) one will need to carefully determine what is important to put in the request. If there are too many constraints the model may fail to find any chunk which is close enough to all of the constraints to be above the threshold, but conversely if there are two few constraints put on the request it may retrieve a chunk which is not really relevant to the current situation. In the current model the chunks of interest for this request do not have a lot of slots to test and the request for a chunk based on a single constraint seems reasonable for the chunks involved. So, we will not adjust that aspect of the model.

The settings of the similarities between items and the :mp parameter are related, so we will discuss them together. By default a chunk is maximally similar to itself and it is maximally dissimilar to all other values. To have chunks related by some intermediate similarity the modeler must set those values. The question becomes how to decide what to make similar and how to set those similarities so that they provide the desired effects. In setting the similarities there are two things to consider: the magnitude of the effect a single mismatch will have on the activation and the relative similarity values among the items involved.

The total effect of similarity depends on how many mismatches there are as discussed with respect to the specificity of the request above, but the effect that each individual mismatch has is based on the similarity setting between the items involved and the setting of :mp. The default range for similarities is from -1 to 0, and the similarity value between the items is multiplied by the setting of the :mp parameter to determine the penalty to the activation. Setting similarities in the default range and then using :mp to scale them often works out well. However, because the :mp parameter is a constant used in all retrieval requests, when one needs there to be different similarity effects for different types of items it may be necessary to change the range of similarities instead of just scaling them all with :mp. To do that one can change the similarity range by setting the :md and :ms parameters (maximum difference and maximum similarity respectively). The recommendation is to always leave :ms at 0, but :md can be set to any value needed to provide an appropriate range. When changing the range, it's often best to then just set :mp to 1 so the similarity values directly reflect the effect on activation, but that's not required and one can still scale them with :mp as well if desired.

How to set the relative similarities between items depends on what sorts of effects one would like the model to show. While it is possible to set each possible similarity value explicitly in the model to produce specific results, it's usually more plausible to set them systematically. In some situations one can rely on other experimental results for guidance in how to set them, for instance research on numbers, language, or perceptual effects may provide a general equation or metric to use in those situations, but other times one may need to determine values appropriate for the current task by parameter exploration or analysis of the data. If parameter estimation is required, one thing that may be useful in determining similarity values is to look at the activations of the competing chunks at the time of the request along with the activation noise value. Assuming all the chunks are sufficiently above the retrieval threshold this equation describes the probability of chunk i being retrieved among those which are being considered (the set j):

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

Using that equation for probability of retrieval one can then determine the expected changes in retrieval probability based on the differences in similarity values. Of course, that level of analysis will not always be necessary, but sometimes it's useful to be able to investigate things in that way.

For this model we do not have any similarity values set and thus since there is no chunk which has an attribute of pink they all get the maximum penalty of -10 (:mp of 10 times the default mismatch value of -1). If we want the model to have a preference for particular items then we will need to set some similarities and we may want to adjust the :mp value as well. Based on this retrieval request and the initial declarative memory chunks, all we need are the similarities between pink and each of blue, red, and green since those are the only items involved that could be partial-matched to the request. However, if this were a task which required a richer set of information and which may involve requests for any color then we would likely want to set all of the possible similarities, and using a sim-hook function like the 1-hit blackjack model does for similarities between numbers might be a good way to do so. For the purposes of the demonstration model however we will just set one similarity value so that red is considered more similar to pink than pink is to blue or green. That way the model will be most likely to retrieve the chunk f2 when requesting a chunk with an attribute of pink, and since we are not concerned with exactly how similar the items are or exactly how much more likely it should be for this model we will just set that to a similarity of -.4 by adding this setting to the model definition:

```
(set-similarities (pink red -.4))
```

and then investigate the effect that has before considering further changes.

Here is the trace of the model run with that addition made:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.259   DECLARATIVE            RETRIEVED-CHUNK V2
    12.259   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    12.259   PROCEDURAL             CONFLICT-RESOLUTION
    12.309   PROCEDURAL             PRODUCTION-FIRED P2
    12.309   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    12.309   DECLARATIVE            start-retrieval
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    26.541   DECLARATIVE            RETRIEVED-CHUNK V1
    26.541   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V1
    26.541   PROCEDURAL             CONFLICT-RESOLUTION
    26.591   PROCEDURAL             PRODUCTION-FIRED P3
    26.591   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.591   DECLARATIVE            start-retrieval
    26.591   PROCEDURAL             CONFLICT-RESOLUTION
    29.931   DECLARATIVE            RETRIEVED-CHUNK F2
    29.931   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F2
...
```

Now we see that the chunk f2 is retrieved in about 3 seconds for this request. That is sufficient for this demonstration so we will not change anything else to affect that retrieval.

Fourth Retrieval Request

The next retrieval request this model makes is very similar to the previous one, except this time the request is for a chunk with an attribute of black:

```
  (p p4
     =goal>
       isa task-state
       state r4
     ?retrieval>
       state free
   ==>
     =goal>
       state r5
     +retrieval>
       isa simple-fact
       attribute black)
```

Like the previous request there is no matching chunk and thus this request will either fail or be satisfied by a partially matched result if there is one above the retrieval threshold. If we run this model for 30 seconds the request is made and then we can look at when the request will be satisfied with mp-show-queue:

```
...
    29.981   PROCEDURAL             PRODUCTION-FIRED P4
    29.981   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    29.981   DECLARATIVE            start-retrieval
    29.981   PROCEDURAL             CONFLICT-RESOLUTION
    30.000   ------                 Stopped because time limit reached

Events in the queue:
   261.522   DECLARATIVE            RETRIEVED-CHUNK F1
   261.522   PROCEDURAL             CONFLICT-RESOLUTION
```

It will retrieve the chunk f1 after more than a 200 second delay. That delay seems unreasonable thus we need to do something to change that, and there are multiple options available depending on what exactly we want to have happen in the model.

If we want it to retrieve some chunk in a reasonable amount of time, then we will need to do something to either change the retrieval time or activation values. The options for doing that have been discussed in previous sections, and we will just summarize them here. We can change the latency factor parameter to make the retrieval faster. We can increase the activation of the potential chunks by changing their base-level activation either by giving them a stronger history or setting the base-level constant. We can decrease the penalty for partial matching by setting similarity values or changing the :mp parameter. Finally, we can add additional context to the buffers so that spreading activation will increase the chunk's activation.

If we don't mind having the request fail, then we can increase the retrieval threshold so that the model fails to retrieve in a shorter time than that. Since the time for a failure is determined by the setting of the retrieval threshold and the latency factor the settings of those parameters effectively specify the upper bound for how long the model can take to perform any retrieval. However, changing those parameters will affect all of the retrieval requests which the model makes.

Another way to handle that would be to use the temporal module in the model so that it can monitor the time that has passed explicitly and then stop waiting once too much time has passed. That allows the model to have a flexible "failure time", but it will typically require the model having additional productions to set up and use the temporal information. Details on using the temporal module are not yet available in the tutorial, but you can find information on using it in the ACT-R reference manual. If it's possible for the model to produce the response without completing the retrieval, for example if it can also find the information by searching for it visually, then that can also be used to finish before the retrieval failure time passes. In situations like that, while the retrieval is happening the model can also be engaged in the alternate process of determining the information. If the retrieval completes before the other method then the model can stop and use the retrieved information. If the other method completes first, then the model will not have to wait until the retrieval succeeds or fails. That can work very well in a learning model as long as the result of the alternate process results in strengthening the same declarative information each time. Then, as the activation of that chunk increases the model will shift from always having to do the deliberate process to being able to rely entirely on the retrieved information. That is similar to how the zbrodoff model in unit 4 of the tutorial operated, except that it did not perform the retrieval of the information in parallel with the alternative mechanism since in that case, the other process, counting, also required the use of declarative retrievals and the model can only perform one retrieval at a time.

In this model we would like some chunk to be retrieved and there is no alternative method available for producing a result. Thus, we need to make some adjustment to change the retrieval time. We are again going to avoid using spreading activation. So that leaves us with partial matching, base-level activation, or the latency factor to be adjusted. For partial matching we could adjust the similarities as we did with the previous retrieval, but that doesn't seem as appropriate to do for black and the target colors. We could also adjust the mismatch penalty, which might work well in this situation though changing it will also affect all of the other partially matched retrievals as well. Since we have already set the chunks' base-level histories to something fairly strong, we don't want to adjust that any further. Changing the base-level constant would also affect the base-level activations, but we will again avoid doing that. If we change the latency factor that is going to affect all of the retrievals which the model makes, and while that might be useful we are not going to do so here. So, among the options available, changing the mismatch penalty is the one that we will investigate further for this retrieval.

The current setting in the model is 10 and with the default maximum dissimilarity value of -1 causes the chunks which mismatch the requested value of black in the attribute slot to have -10 added to their activations. We want the activation of those chunks to be higher so that they are retrieved faster, thus we need to decrease the penalty. If we had data to fit, that would give us a guide as to how long the retrievals should be taking and suggest a more specific change to make, but since this is just for demonstration purposes we will just set it to something lower and look at the result. If we decrease the :mp value to 2 and run the model again here is what we get:

```
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED P1
     0.050   PROCEDURAL             CLEAR-BUFFER GOAL
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK0
     0.050   DECLARATIVE            start-retrieval
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.259   DECLARATIVE            RETRIEVED-CHUNK V2
    12.259   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V2
    12.259   PROCEDURAL             CONFLICT-RESOLUTION
    12.309   PROCEDURAL             PRODUCTION-FIRED P2
    12.309   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    12.309   DECLARATIVE            start-retrieval
    12.309   PROCEDURAL             CONFLICT-RESOLUTION
    26.541   DECLARATIVE            RETRIEVED-CHUNK V1
    26.541   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL V1
    26.541   PROCEDURAL             CONFLICT-RESOLUTION
    26.591   PROCEDURAL             PRODUCTION-FIRED P3
    26.591   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.591   DECLARATIVE            start-retrieval
    26.591   PROCEDURAL             CONFLICT-RESOLUTION
    26.727   DECLARATIVE            RETRIEVED-CHUNK F2
    26.727   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F2
    26.727   PROCEDURAL             CONFLICT-RESOLUTION
    26.777   PROCEDURAL             PRODUCTION-FIRED P4
    26.777   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.777   DECLARATIVE            start-retrieval
    26.777   PROCEDURAL             CONFLICT-RESOLUTION
    26.855   DECLARATIVE            RETRIEVED-CHUNK F1
    26.855   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F1
    26.855   PROCEDURAL             CONFLICT-RESOLUTION
 ...
```

The fourth retrieval now completes in less than 200 ms as does the third retrieval which previously took more than 3 seconds. If we look closely at the activation traces we will see that without any similarities set f1 is the chunk chosen here because of noise since the other activation quantities of the potential chunks are very similar to each other, with the only difference being that f2 has a slightly higher base-level activation than f1 and f3 since it has an extra reference. We are going to consider that a reasonable time for this retrieval, but before moving on there is one other option to consider when adjusting the mismatch penalty.

If we didn't want to affect the timing of the third request but still wanted to speed up the fourth one it is possible to override the global :mp value in a request. That may be necessary in tasks where different types of information need to be retrieved and some require more specificity than others i.e. how close it must be to the requested values. That is done using the :mp-value request parameter in the retrieval request. If we set the global mp value back to 10 and change the request in production **p4** like this:

```
     +retrieval>
       isa simple-fact
       attribute black
       :mp-value 2
```

we get the following result from running it:

```
 ...
    26.541   PROCEDURAL             CONFLICT-RESOLUTION
    26.591   PROCEDURAL             PRODUCTION-FIRED P3
    26.591   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.591   DECLARATIVE            start-retrieval
    26.591   PROCEDURAL             CONFLICT-RESOLUTION
    29.931   DECLARATIVE            RETRIEVED-CHUNK F2
    29.931   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F2
    29.931   PROCEDURAL             CONFLICT-RESOLUTION
    29.981   PROCEDURAL             PRODUCTION-FIRED P4
    29.981   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    29.981   DECLARATIVE            start-retrieval
    29.981   PROCEDURAL             CONFLICT-RESOLUTION
    30.059   DECLARATIVE            RETRIEVED-CHUNK F1
    30.059   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL F1
    30.059   PROCEDURAL             CONFLICT-RESOLUTION
...
```

The third retrieval request again takes about 3 seconds to complete but the fourth one is still around 200ms.

Fifth Retrieval Request

Up to this point we have avoided the effects of spreading activation on retrievals, but for this request we will investigate issues related to using it. To do that we are going to use chunks based on the following chunk-types:

```
  (chunk-type number representation)
  (chunk-type math-fact arg1 arg2 result operator)
  (chunk-type context val1 val2 val3 goal)
```

and these initial chunks in declarative memory:

```
(zero isa number representation "0")
(one isa number representation "1")
(two isa number representation "2")
(three isa number representation "3")

(1-1 isa math-fact arg1 one arg2 one result zero operator subtract)
(2-1 isa math-fact arg1 two arg2 one result one operator subtract)
(3-2 isa math-fact arg1 three arg2 two result one operator subtract)
(1+1 isa math-fact arg1 one arg2 one result two operator add)
(1+2 isa math-fact arg1 one arg2 two result three operator add)
```

The first thing to consider when using spreading activation is which buffers the model is going to use as sources for spreading activation. By default, only the imaginal buffer is considered a source of activation, but occasionally one also will want to use the goal buffer to spread activation (note previous versions of ACT-R used the goal buffer as the only buffer to spread activation by default, but with version 7 the imaginal buffer is now the only one to spread activation by default). Doing that will require setting the :ga parameter to enable the goal buffer as a source. For this model we will only be using the imaginal buffer as a source since our goal buffer is only being used to hold state information which is not related to items in declarative memory.

The default weight for activation spreading from the imaginal buffer is 1 and that is sufficient for our purposes here so we do not need to change any parameters. The value used in setting the source spread from a buffer is often set at 1, but other values may be used and some researchers have found that adjustments to the source spread parameters can account very well for differences between individuals. For this model we will not adjust that setting, but you may want to investigate that on your own after working through the demonstrations to see how it affects things.

The only other parameter required for using spreading activation is :mas which when set to a number both enables spreading activation and specifies the value of S in the equation for S_{ji} values. In this model we have set that parameter to a value of 2 initially, but we may need to modify that as we go along.

The next request which the model makes is with this production:

```
(p p5
   =goal>
     isa task-state
     state r5
   ?imaginal>
     state free
   =retrieval>
 ==>
   =goal>
     state r6
   +imaginal>
     isa context
     val1 one
     val2 two
   +retrieval>
     isa math-fact
     - arg1 nil
     - arg2 nil
     - result nil)
```

That production makes both a request to the imaginal buffer to create a chunk with values in the val1 and val2 slots and a retrieval request for a chunk which has values in the arg1, arg2, and result slots. Running the model with the activation trace enabled produces this output for that retrieval request (with the :mp parameter set back to 2 again):

```
...
    26.905   PROCEDURAL             PRODUCTION-FIRED P5
    26.905   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    26.905   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    26.905   DECLARATIVE            start-retrieval
Computing activation for chunk 1-1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9530089
Total base-level: -0.9530089
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ZERO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.36828277
Adding permanent noise 0.0
Chunk 1-1 has an activation of: -0.5847261
Chunk 1-1 has the current best activation -0.5847261
Computing activation for chunk 2-1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9530089
Total base-level: -0.9530089
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.17789373
Adding permanent noise 0.0
```

```
Chunk 2-1 has an activation of: -1.1309026
Computing activation for chunk 3-2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9530089
Total base-level: -0.9530089
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: THREE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.33106902
Adding permanent noise 0.0
Chunk 3-2 has an activation of: -0.6219399
Computing activation for chunk 1+1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9530089
Total base-level: -0.9530089
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.38912883
Adding permanent noise 0.0
Chunk 1+1 has an activation of: -1.3421377
Computing activation for chunk 1+2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
```

```
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.9530089
Total base-level: -0.9530089
Computing activation spreading from buffers
Total spreading activation: 0.0
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: THREE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.0474016
Adding permanent noise 0.0
Chunk 1+2 has an activation of: -0.9056073
Chunk 1-1 with activation -0.5847261 is the best
    26.905   PROCEDURAL             CONFLICT-RESOLUTION
...
```

The thing to note there is that there is no spreading activation occurring even though that production made a request to create a chunk in the imaginal buffer. The reason for that is because the sources of activation are determined at the time the request is made, but it takes the imaginal module time to create the chunk. Thus, there is no chunk in the imaginal buffer at the time the retrieval request occurs from which to spread activation. Since we want to see the effects of spreading activation from that chunk we will change the model so that production p5 does not make a retrieval request and then look at the next production, p6, which makes that same request:

```
(p p5
   =goal>
     isa task-state
     state r5
   ?imaginal>
     state free
   =retrieval>
 ==>
   =goal>
     state r6
   +imaginal>
     isa context
     val1 one
     val2 two)
```

```
  (p p6
     =goal>
       isa task-state
       state r6
     =imaginal>
   ==>
     +retrieval>
       isa math-fact
       - arg1 nil
       - arg2 nil
       - result nil
     =goal>
       state r7
     =imaginal>
       val3 add)
```

Production p6 will not be selected and fire until there is a chunk in the imaginal buffer since it has a test for a chunk in the buffer on its LHS. It then modifies the chunk in the imaginal buffer along with making a retrieval request for a math-fact. Thus, there should now be some activation spreading and here is the activation trace generated from this request:

```
...
    27.155   PROCEDURAL             PRODUCTION-FIRED P6
    27.155   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    27.155   DECLARATIVE            start-retrieval
Computing activation for chunk 1-1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.13018736 from source ONE level  0.33333334 times Sji 0.39056206
    Spreading activation  0.0 from source TWO level  0.33333334 times Sji 0.0
Total spreading activation: 0.13018736
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ZERO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
```

```
Adding transient noise 0.36828277
Adding permanent noise 0.0
Chunk 1-1 has an activation of: -0.45916334
Chunk 1-1 has the current best activation -0.45916334
Computing activation for chunk 2-1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.13018736 from source ONE level  0.33333334 times Sji 0.39056206
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.19960088
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.17789373
Adding permanent noise 0.0
Chunk 2-1 has an activation of: -0.9359263
Computing activation for chunk 3-2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
#|Warning: Calculated Sji value between ONE and 3-2 is negative, but using a value of 0. |#
    Spreading activation  0.0 from source ONE level  0.33333334 times Sji 0.0
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.069413505
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: THREE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.33106902
Adding permanent noise 0.0
Chunk 3-2 has an activation of: -0.55715096
Computing activation for chunk 1+1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.20456855 from source ADD level  0.33333334 times Sji 0.61370564
    Spreading activation  0.13018736 from source ONE level  0.33333334 times Sji 0.39056206
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.4041694
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise -0.38912883
Adding permanent noise 0.0
Chunk 1+1 has an activation of: -0.94259286
Computing activation for chunk 1+2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.20456855 from source ADD level  0.33333334 times Sji 0.61370564
#|Warning: Calculated Sji value between ONE and 1+2 is negative, but using a value of 0. |#
    Spreading activation  0.0 from source ONE level  0.33333334 times Sji 0.0
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.27398205
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: THREE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.0474016
Adding permanent noise 0.0
Chunk 1+2 has an activation of: -0.6362498
Chunk 1-1 with activation -0.45916334 is the best
    27.155   PROCEDURAL             CONFLICT-RESOLUTION
    28.421   DECLARATIVE            RETRIEVED-CHUNK 1-1
    28.421   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL 1-1
...
```

Before looking at the final result of the request we will first look at what the sources of activation are. From the activation trace we see that it lists these three chunks as sources: add, one, and two for all of the chunks:

```
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
```

The thing to note is that the chunk add being in a slot of the imaginal buffer chunk was the result of a modification to the chunk made as an action in the production which makes the retrieval request. Modifications made directly by the production will always take effect before the retrieval request starts. If we enable the high detail trace and run it again that can be seen in this sequence of events following the production firing:

```
...
    27.155   PROCEDURAL             PRODUCTION-FIRED P6
    27.155   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
    27.155   PROCEDURAL             MOD-BUFFER-CHUNK IMAGINAL
    27.155   PROCEDURAL             MODULE-REQUEST RETRIEVAL
    27.155   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    27.155   DECLARATIVE            start-retrieval
...
```

The mod-buffer-chunk actions for the goal and imaginal buffer occur before the declarative module starts the retrieval. Also worth noting is that the clearing of buffers by the production will also precede the start of the declarative retrieval. Using the high detail trace can be helpful to determine why items are or are not sources when looking at other situations because to be a source the change must occur prior to the start-retrieval action of the declarative module.

Looking at the result of that retrieval we see that it retrieved the chunk 1-1:

```
    28.421   DECLARATIVE            RETRIEVED-CHUNK 1-1
    28.421   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL 1-1
```

which looks like this:

```
1-1
   RESULT  ZERO
   ARG1  ONE
   ARG2  ONE
   OPERATOR  SUBTRACT
```

That seems unusual given that we have sources of one, two, and add and there's another chunk which looks like this that seems like it should be getting more spreading activation:

```
1+2
   RESULT  THREE
   ARG1  ONE
   ARG2  TWO
   OPERATOR  ADD
```

We will look at the activation trace for those two items to see what causes the difference, and here are the relevant traces:

```
Computing activation for chunk 1-1
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.13018736 from source ONE level  0.33333334 times Sji 0.39056206
    Spreading activation  0.0 from source TWO level  0.33333334 times Sji 0.0
Total spreading activation: 0.13018736
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: ZERO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.36828277
Adding permanent noise 0.0
Chunk 1-1 has an activation of: -0.45916334
...
Computing activation for chunk 1+2
Computing base-level
Starting with blc: 0.0
Computing base-level from 1 references (0.000)
  creation time: 0.000 decay: 0.5  Optimized-learning: T
base-level value: -0.95763344
Total base-level: -0.95763344
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.20456855 from source ADD level  0.33333334 times Sji 0.61370564
#|Warning: Calculated Sji value between ONE and 1+2 is negative, but using a value of 0. |#
    Spreading activation  0.0 from source ONE level  0.33333334 times Sji 0.0
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.27398205
Computing partial matching component
  comparing slot ARG1
  Requested: - NIL  Chunk's slot value: ONE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot ARG2
  Requested: - NIL  Chunk's slot value: TWO
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
  comparing slot RESULT
  Requested: - NIL  Chunk's slot value: THREE
  similarity: -1.0
  negation test with similarity not ms has no effect
  effective similarity value is 0.0
Total similarity score 0.0
Adding transient noise 0.0474016
Adding permanent noise 0.0
Chunk 1+2 has an activation of: -0.6362498
```

Looking at those traces we see that those two chunks have the same base-level activation and chunk 1+2 does have a higher total spreading activation value. Chunk 1-1 gets a greater boost from noise than 1+2, so the first though might be that it's just an issue with noise. However, a closer look at the spreading activation calculations reveals a warning and raises some interesting questions:

```
Computing activation for chunk 1-1
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.13018736 from source ONE level  0.33333334 times Sji 0.39056206
    Spreading activation  0.0 from source TWO level  0.33333334 times Sji 0.0
Total spreading activation: 0.13018736

Computing activation for chunk 1+2
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.20456855 from source ADD level  0.33333334 times Sji 0.61370564
#|Warning: Calculated Sji value between ONE and 1+2 is negative, but using a value of 0. |#
    Spreading activation  0.0 from source ONE level  0.33333334 times Sji 0.0
    Spreading activation  0.069413505 from source TWO level  0.33333334 times Sji 0.20824051
Total spreading activation: 0.27398205
```

Ignoring the warning for now, one question is why does the S_{ji} from one to 1-1 differ from the S_{ji} from one to 1+2, and another is why is the S_{ji} from one to 1+2 zero?

The answer to the first issue has to do with how S_{ji}s are computed when there are multiple references within a chunk. The equation for S_{ji} from the main unit 5 text:

$$S_{ji} = S - \ln(fan_j)$$

is a simplification of the full calculation which is only true when there's a single link between chunks j and i, but in this case the chunk one occurs in two different slots of the chunk 1-1. The complete form of the equation for S_{ji} uses the value fan_{ji} instead of fan_j where fan_{ji} is defined as:

$$fan_{ji} = \frac{1 + slots_j}{slotsof_{ji}}$$

***slots$_j$*:** the number of slots in which *j* is the value across all chunks in declarative memory

***slotsof$_{ji}$*:** the number of slots in chunk *i* which have *j* as the value (plus 1 if chunk *i* is chunk *j*)

In this case, j is the chunk one and i is the chunk 1-1. Chunk one is a value in nine slots of the chunks in declarative memory, so that is $slots_j$, and it occurs in two slots of chunk 1-1, so that is the value for $slots_{ji}$. Combining that with the value of 2 for S as was set in the model we get:

$$S_{ji} = 2 - \ln\left(\frac{1+9}{2}\right) = 2 - 1.609438 = 0.39056206$$

which is what we see in the trace. For the S_{ji} between the chunk one and the chunk 1+2 the equation is:

$$S_{ji} = 2 - \ln\left(\frac{1+9}{1}\right) = 2 - 2.3025851 = \text{-}0.30258512$$

which is actually a negative spreading of activation. The warning before that calculation:

```
#|Warning: Calculated Sji value between ONE and 1+2 is negative, but using a value of 0. |#
```

indicates that a negative activation spread is treated as 0 by default. This is a safety test that is enabled by default to prevent negative associations since they would be inhibiting the retrieval of related information instead of supporting it. The easy way to fix that is to make sure that the S value is set high enough to avoid the negative value. Occasionally situations occur where one may want that inhibitory behavior, and in those situations it's still advised to set S high enough that items don't automatically get negative S_{ji} values. Instead, the recommendation is to set those desired negative associations explicitly with the add-sji command.

To fix the issue with negative associations in the model we will set our S value to 4 which should be sufficient to keep all S_{ji} values positive (as long as fan_{ji} is less than 54 it will be positive). When we run the model after making that change we see that the model does retrieve the chunk we would expect it to:

```
...
    27.155   PROCEDURAL             PRODUCTION-FIRED P6
    27.155   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    27.155   DECLARATIVE            start-retrieval
    27.155   PROCEDURAL             CONFLICT-RESOLUTION
    27.381   DECLARATIVE            RETRIEVED-CHUNK 1+2
```

However, to be sure things are doing what we expect we should look at the activation trace to make sure, and here is the trace with the base-level and similarity sections removed since those are identical among these chunks:

```
Computing activation for chunk 1-1
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.796854 from source ONE level  0.33333334 times Sji 2.390562
    Spreading activation  0.0 from source TWO level  0.33333334 times Sji 0.0
Total spreading activation: 0.796854
...
Adding transient noise 0.36828277
Adding permanent noise 0.0
Chunk 1-1 has an activation of: 0.20750335

Computing activation for chunk 2-1
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.796854 from source ONE level  0.33333334 times Sji 2.390562
    Spreading activation  0.73608017 from source TWO level  0.33333334 times Sji 2.2082405
Total spreading activation: 1.5329342
...
Adding transient noise -0.17789373
Adding permanent noise 0.0
Chunk 2-1 has an activation of: 0.39740703

Computing activation for chunk 3-2
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.0 from source ADD level  0.33333334 times Sji 0.0
    Spreading activation  0.56580496 from source ONE level  0.33333334 times Sji 1.6974149
    Spreading activation  0.73608017 from source TWO level  0.33333334 times Sji 2.2082405
Total spreading activation: 1.3018851
...
Adding transient noise 0.33106902
Adding permanent noise 0.0
Chunk 3-2 has an activation of: 0.67532074

Computing activation for chunk 1+1
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.87123525 from source ADD level  0.33333334 times Sji 2.6137056
    Spreading activation  0.796854 from source ONE level  0.33333334 times Sji 2.390562
    Spreading activation  0.73608017 from source TWO level  0.33333334 times Sji 2.2082405
Total spreading activation: 2.4041696
...
Adding transient noise -0.38912883
Adding permanent noise 0.0
```

```
Chunk 1+1 has an activation of: 1.0574073

Computing activation for chunk 1+2
...
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK1-0
    sources of activation are: (ADD ONE TWO)
    Spreading activation  0.87123525 from source ADD level  0.33333334 times Sji 2.6137056
    Spreading activation  0.56580496 from source ONE level  0.33333334 times Sji 1.6974149
    Spreading activation  0.73608017 from source TWO level  0.33333334 times Sji 2.2082405
Total spreading activation: 2.1731205
...
Adding transient noise 0.0474016
Adding permanent noise 0.0
Chunk 1+2 has an activation of: 1.2628886
Chunk 1+2 is now the current best with activation 1.2628886
```

Looking over the spreading activation values shows that in fact the chunk 1+1 gets more spreading activation than chunk 1+2 and it is only because of noise that we retrieved 1+2 this time. The reason for that is because the chunk 1+1 is also receiving activation spread from each of the sources since it also contains the chunk two in its result slot, and since it has two occurrences of the chunk one it has a greater S_{ji} from one than the chunk 1+2 does.

This highlights a big distinction between spreading activation and partial matching. Spreading activation is a bottom-up mechanism which increases the activation of chunks relative to how well they match the current context without regard for the specific structure of that information in the target chunks. Whereas partial matching is a top-down process which penalizes those chunks which do not match the specific pattern provided in the request. In models of simple tasks often only one effect or the other is desired and to keep things simple only that particular mechanism is enabled, as was the case for the tutorial unit models, but in more complex models both effects may be desirable in which case one has to be more careful about both maintaining an appropriate context and making appropriate requests to achieve the desired results.

If we want this model to be relatively certain of retrieving the fact associated with adding one and two we will need to add that pattern of information into the request. We will not make that change to the model as part of the demonstration, but you should feel free to try that and see how the activations change. You may also want to try changing the similarities between the number chunks to see how that affects things as well because the current model does not set any similarities between the number chunks.

Last two productions

The final issue we will look at does not involve a retrieval. Instead we will look at a potential issue which can arise when creating chunks that will be merged into declarative memory. In most situations chunks will merge and strengthen existing declarative chunks as one would expect, but there is a situation which can sometimes occur which is worth discussing here because it can be confusing. The issue can arise when a model creates a chunk which has slot values that are chunks which are not in declarative memory (typically because they reference a chunk currently in a buffer) and then merges that chunk into declarative memory.

For the example we will investigate what happens when these two productions fire:

```
(p p7
     =goal>
       isa task-state
       state r7
     =imaginal>
     =retrieval>
   ==>
     =goal>
       state r8
     =imaginal>
       goal =goal)

  (p p8
     =goal>
       isa task-state
       state r8
   ==>
     +goal>
     -imaginal>)
```

in the context of these chunks which are in declarative memory:

```
   (g1 isa task-state state r8)
   (old-context isa context val1 one val2 two val3 add goal g1)
```

Before running the model we will look at what we might expect to happen. Production p7 waits for the previous retrieval to complete and then modifies the chunks in the goal and imaginal buffers. The goal chunk is modified such that it now looks just like g1 and the imaginal chunk has that current goal buffer chunk placed into its goal slot. Production p8 fires next since the goal buffer state matches and then it performs two actions. It makes a request to the goal module to create a new chunk, which will implicitly clear the current chunk from the goal buffer, and it clears the chunk from the imaginal buffer. What we might expect to happen here is that the goal buffer's chunk will merge with chunk g1 and then the imaginal buffer's chunk will merge with the chunk old-context.

After running the model here is what we see in declarative memory with respect to chunks with a state slot (task-state chunks) and chunks with a goal slot (the context chunks) which are found using the sdm command here (with a mix of Lisp and Python calls for examples), but could also be found using the filter at the top of the Declarative viewer of the ACT-R Environment which allows one to restrict the display to contain only chunks which have the particular set of slots chosen as the filter:

```
...
    27.431   PROCEDURAL             PRODUCTION-FIRED P7
    27.431   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
    27.431   PROCEDURAL             CONFLICT-RESOLUTION
    27.481   PROCEDURAL             PRODUCTION-FIRED P8
    27.481   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    27.481   PROCEDURAL              CLEAR-BUFFER GOAL
    27.481   GOAL                    SET-BUFFER-CHUNK GOAL CHUNK2
    27.481   PROCEDURAL              CONFLICT-RESOLUTION
    27.481   ------                  Stopped because no events left to process
```

```
? (sdm - state nil)
G1
   STATE  R8
```

```
>>> actr.sdm('-','goal','nil')
OLD-CONTEXT
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  G1
CHUNK1-0
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  G1
```

We see that there is only one task-state chunk as we expected, but there are two apparently identical context chunks. One way to see why that happens would be to step through the actions which occur as a result of that production firing and inspect things carefully after each event. If you would like to do that you can do so, but here we will just inspect the actions in the trace and explain the outcome.

Before doing that, there is something else which we can do that might make things clearer. If we turn off the :ncnar parameter the model will not automatically normalize the chunk names when chunks are merged and that might also help to see what has happened. Before running the model again we need to turn that off by adding this to the sgp call in the model:

```
  (sgp ... :ncnar nil)
```

Now, after we run the model this is what is shown for the task state chunks in declarative memory:

```
G1
   STATE  R8
```

There is still only one chunk named g1. For the context chunks in declarative memory there are again two, but they look different now:

```
OLD-CONTEXT
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  G1

CHUNK1-0
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  CHUNK0-0
```

However, although they appear to have different values in their goal slots there is no difference between them in terms of the model's operation because both of those names are referencing the same chunk. If we print out those two chunks we see that chunk0-0 actually names the same chunk g1 as indicated by that name in parentheses:

```
? (pprint-chunks g1 chunk0-0)
G1
   STATE  R8

CHUNK0-0 (G1)
   STATE  R8
```

Changing the :ncnar parameter only changes how the information is provided to the modeler – when :ncnar is true it will always show the "true name" for a chunk which appears in a slot of other chunks regardless of any other names for that chunk which may exist. The true name is the name of the chunk which is in the model's declarative memory with which the other chunk(s) have merged.

Here are the events from the high detail trace for production **p8** firing, which may suggest what has happened, but we will still go over the details:

```
     27.481   PROCEDURAL             PRODUCTION-FIRED P8
     27.481   PROCEDURAL             MODULE-REQUEST GOAL
     27.481   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     27.481   PROCEDURAL             CLEAR-BUFFER GOAL
     27.481   GOAL                   CREATE-NEW-BUFFER-CHUNK GOAL
     27.481   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK2
```

Although all of those events are listed as occurring at the same time, as we've seen using the stepper, each is executed individually in the order that they are shown. Thus, first the production fires, then the request is made to the goal buffer, then the imaginal buffer gets cleared, and finally the goal buffer gets cleared.

The important question is then how does declarative memory handle merging chunks? The answer is that it only attempts to merge chunks immediately upon their being cleared from a buffer, and it will only merge chunks if all of their contents are perfect matches. When the slot values are chunk names a perfect match means that they must refer to the same chunk (note however that that doesn't mean that the slot values must have the same chunk name because merged chunks can still be referenced by either name).

Thus, when the imaginal buffer gets cleared the chunk in it looks like this:

```
CHUNK1-0
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  CHUNK0-0
```

At that time the chunk chunk0-0 is still in the goal buffer and has not been merged with chunk g1 in declarative memory. Because of that the chunk chunk1-0 is not a perfect match to the chunk old-context which is in declarative memory since the value of their goal slots, chunk0-0 and g1, are different chunks. That means chunk1-0 must be added to declarative memory as a new chunk. Then, the goal buffer gets cleared. Because chunk chunk0-0 is a perfect match to the chunk g1 those two chunks are then merged. The merging of those two chunks does not make the declarative module retroactively merge the chunks old-context and chunk1-0. Thus, declarative memory still has two context chunks; one with a value of g1 in the goal slot and one with a value of chunk0-0 in the goal slot, but both of those values now reference a single chunk.

That may seem like a problem with how merging works, but there are good reasons for having it work sequentially like that. One of those reasons is that it allows the modeler to control what happens – sometimes one might want separate chunks instead of having them merged. If we do not want separate chunks, then we have to ensure that all the chunks in the slots of the chunk we want to merge into declarative memory are merged into declarative memory first (in this case the chunk in the goal buffer must be merged into declarative memory before the chunk in the imaginal buffer since that goal buffer chunk is in a slot of the chunk in the imaginal buffer). If we want that to happen within a single production, then this becomes one of the rare situations where controlling the order in which a production's actions occur matters.

Generally, the order in which a production performs its actions does not matter since they are all happening at the same time and there are usually no interactions among them. However, since the simulation has to perform the actions sequentially, in situations like this one the modeler may need to make sure some things happen in a particular order, but the modeler cannot arbitrarily order a production's actions. A production will always perform its actions in the following order: all user actions (!eval!, !bind!, and !output!), all buffer modifications, all requests, then all buffer clearing actions. Within a particular type of action it will perform the explicit actions in the order provided in the production followed by any implicit actions of that type (like clearing the buffer due to strict harvesting or as a result of a request) in no particular order. Thus, if we want the goal buffer to be cleared prior to the imaginal buffer we will have to explicitly perform that action in the production instead of letting it happen implicitly, and it will have to be placed before the imaginal buffer clearing.

Here is a modified version of p8 which adds an explicit clearing of the goal buffer before the clearing of the imaginal buffer:

```
(p p8
   =goal>
     isa task-state
     state r8
   ==>
   +goal>
   -goal>
   -imaginal>)
```

When we run the model after saving that change and reloading we get the following result for the context chunks in declarative memory:

```
? (sdm - goal nil)
OLD-CONTEXT
   VAL1  ONE
   VAL2  TWO
   VAL3  ADD
   GOAL  G1
```

which shows that there is only one chunk now thus the imaginal chunk has been merged with the old-context chunk. If we wanted to investigate further we could make sure that that chunk has two references by looking at the details in the declarative viewer or by using the sdp command to check its parameters, and if we do so we find that it does have a value of two for its reference-count:

```
>>> actr.sdp('old-context')
Declarative parameters for chunk OLD-CONTEXT:
 :Activation  0.190
 :Permanent-Noise  0.000
 :Base-Level -0.270
 :Creation-Time 0.000
 :Reference-Count  2
 :Source-Spread  0.000
 :Sjis ((OLD-CONTEXT . 4.0) (ONE . 1.6974149) (TWO . 2.2082405) (ADD . 2.6137056)
 (G1 . 3.3068528))
 :Similarities ((OLD-CONTEXT . 0.0))
```

Unit 6: Selecting Productions on the Basis of Their Utilities and Learning these Utilities

Occasionally, we have had cause to set parameters of productions so that one production will be preferred over another in the conflict resolution process. Now we will examine how production utilities are computed and used in conflict resolution. We will also look at how these utilities are learned.

6.1 The Utility Theory

Each production has a utility associated with it which can be set directly as we have seen in some of the previous units. Like activations, utilities have noise added to them. The noise is controlled by the utility noise parameter s which is set with the parameter :egs. The noise is distributed according to a logistic distribution with a mean of 0 and a variance of

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

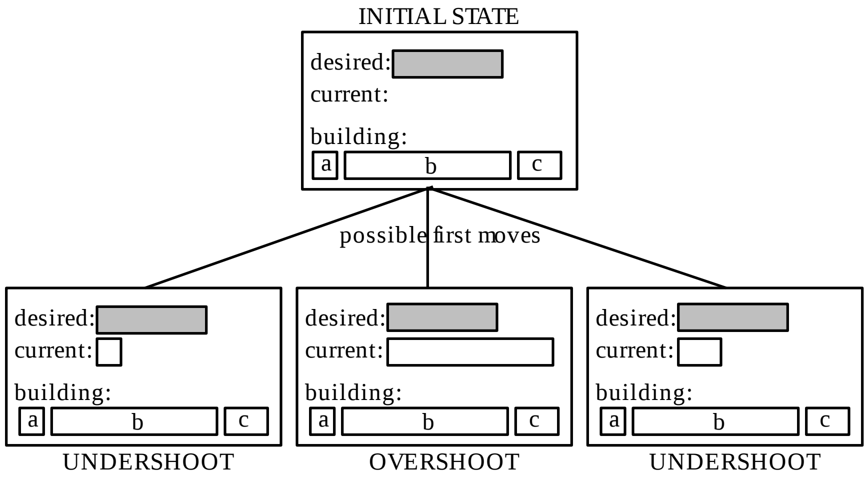
If there are a number of productions competing with expected utility values U_j the probability of choosing production i is described by the formula

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

where the summation j is over all the productions which currently have their conditions satisfied. Note however that that equation only serves to describe the production selection process. It is not actually computed by the system. The production with the highest utility (after noise is added) will be the one chosen to fire.

6.2 Building Sticks Example

We will illustrate these ideas with an example from problem solving. Lovett (1998) looked at participants solving the building-sticks problem illustrated in the figure below. This is an isomorph of Luchins waterjug problem that has a number of experimental advantages. Participants are given an unlimited supply of building sticks of three lengths and are told that their objective is to create a target stick of a particular length. There are two basic strategies they can select – they can either start with a stick smaller than the desired length and add sticks (like the addition strategy in Luchins waterjugs) or they can start with a stick that is too long and "saw off" lengths equal to various sticks until they reach the desired length (like the subtraction strategy). We will call the first of those the undershoot strategy and the second the overshoot strategy. Subjects show a strong tendency to hillclimb and choose as their first stick a stick that will get them closest to the target stick.



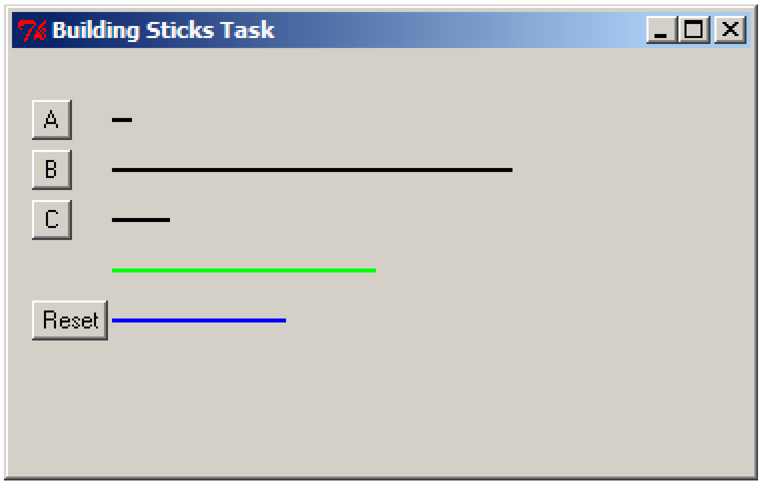
You can go through a version of this task that was written to work with ACT-R models by loading the bst.lisp file in Lisp or importing the bst file in Python. The function bst-test in Lisp or test in the bst module in Python takes one parameter indicating how many sample pairs of problems to present and an optional second parameter which indicates whether a person or model is performing the task. The default is to run the model, so to run yourself through one pair of problems you would use one of these:

```
? (bst-test 1 t)
```

```
>>> bst.test(1,True)
```

The return value will be a list of two numbers indicating how many times you used the overshoot strategy on the first and second problem from the pair respectively.

The experiment will look something like this:



To do the task you will see four lines initially. The top three are black and correspond to the building sticks you have available. The fourth line is green and that is the target length you are attempting to build. The current stick you have built so far will be blue and below the target stick. You will build the current stick by pressing the button to the left of a stick you would like to use next. If your current line is shorter than the target the new stick will be added to the current stick, and if your current line is longer than the target the new stick will be subtracted from the current stick. When you have successfully matched the target length the word "Done" will appear below the current stick and you will progress to the next trial. At any time you can hit the button labeled Reset to clear the current stick and start over.

As it turns out, both of the problems presented in that test set can only be solved by the overshoot strategy. However, the first one looks like it can be solved more easily by the undershoot strategy. The exact lengths of the sticks in pixels for that problem are:

```
A = 15  B = 200  C = 41 Goal = 103
```

The difference between B and the goal is 97 pixels while the difference between C and the goal is only 62 pixels – a 35 pixel difference of differences. However, the only solution to the problem is B – 2C – A. The same solution holds for the second problem:

```
A = 10  B = 200 C = 29 Goal = 132
```

But in this case the difference between B and the goal is 68 pixels while the difference between C and the goal is 103 pixels – a 35 pixel difference of differences in the other direction. You can run the model on these problems and it will tend to choose undershoot for the first about 75% of the time and overshoot for the second about 75% of the time. You can run the model multiple times using the bst-test or test function with the number of pairs to run the model through to see the results for yourself. If you only run the model through one pair it will perform the task with a visible window so that you can watch it, but if you run more than one pair it will use a virtual window to complete the task faster.

The model for the task involves many productions for encoding the screen and selecting sticks. However, the critical behavior of the model is controlled by four productions that make the decision as to whether to apply the overshoot or the undershoot strategy.

```
(p decide-over
   =goal>
      isa       try-strategy
      state     choose-strategy
      strategy  nil
   =imaginal>
      isa       encoding
      under     =under
      over      =over
   !eval! (< =over (- =under 25))
  ==>
   =imaginal>
   =goal>
      state     prepare-mouse
      strategy  over
   +visual-location>
      isa       visual-location
      kind      oval
      value     "b")


(p force-over
   =goal>
      isa       try-strategy
      state     choose-strategy
    - strategy  over
  ==>
   =goal>
      state     prepare-mouse
      strategy  over
   +visual-location>
      isa       visual-location
      kind      oval
      value     "b")



(p decide-under
   =goal>
      isa       try-strategy
      state     choose-strategy
      strategy  nil
   =imaginal>
      isa       encoding
      over      =over
      under     =under
   !eval! (< =under (- =over 25))
  ==>
   =imaginal>
   =goal>
      state     prepare-mouse
      strategy  under
   +visual-location>
      isa       visual-location
      kind      oval
      value     "c")


(p force-under
   =goal>
      isa       try-strategy
      state     choose-strategy
    - strategy  under
  ==>
   =goal>
      state     prepare-mouse
      strategy  under
   +visual-location>
      isa       visual-location
      kind      oval
      value     "c")
```

The key information is in the over and under slots of the chunk in the **imaginal** buffer. The over slot encodes the pixel difference between stick b and the target stick, and the under slot encodes the difference between the target stick and stick c. These values have been computed by prior productions that encode the problem. If one of these differences appears to get the model much closer to the target (more than 25 pixels closer than the other) then the decide-under or decide-over productions can fire to choose the strategy. In all situations, the other two productions, force-under and force-over, can apply. Thus, if there is a clear difference in how close the two sticks are to the target stick there will be three productions (one decide, two force) that can apply and if there is not then just the two force productions can apply. The choice among the productions is determined by their relative utilities which we can see using the Procedural tool in the ACT-R Environment, or by using the spp command:

```
? (spp force-over force-under decide-over decide-under)

>>> actr.spp('force-over','force-under','decide-over','decide-under')
```

The output from that will look like this if the model has not yet performed the task:

```
Parameters for production FORCE-OVER:
 :utility    NIL
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production FORCE-UNDER:
 :utility    NIL
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-OVER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-UNDER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

The productions' current utility values, labeled :u, are set in the model using the spp command:

```
(spp decide-over :u 13)
(spp decide-under :u 13)
(spp force-over :u 10)
(spp force-under :u 10)
```

The :u parameters are set to 10 for the force productions and to 13 for the decide productions since making the decision based on which one looks closer should be preferred to just guessing, at least initially. The :utility value shown in the output from spp indicates the last computed utility value for the production during a conflict-resolution event and includes the utility noise. The value of nil in the output above before the model runs indicates that the production has not yet been used. If you run the model through the task and then check the parameters you will see something like this which shows the noisy utility values for the productions which matched the state and could possibly have been selected during conflict-resolution:

```
Parameters for production FORCE-OVER:
 :utility  8.893
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production FORCE-UNDER:
 :utility 15.080
 :u  10.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-OVER:
 :utility 15.434
 :u  13.372
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production DECIDE-UNDER:
 :utility    NIL
 :u  13.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

Let us consider how these productions apply in the case of the two problems in the model. Since the difference between the under and over differences is 35 pixels, there will be one decide and two force productions that match for both problems. Let us consider the probability of choosing each production using the equation shown above, and the fact that the noise parameter, *s,* is set to 3 in the model.

First, consider the probability of the decide production:

$$\Pr obability(decide) = \frac{e^{13/4.24}}{e^{13/4.24}+e^{10/4.24}+e^{10/4.24}}$$
$$= \frac{e^{3/4.24}}{e^{3/4.24}+e^{0}+e^{0}} = .504$$

Similarly, the probability of the two force productions can be shown to be .248. Thus, there is a .248 probability that a force production will fire that has the model try to solve the problem in the direction other than it appears.

6.3 Utility Learning

So far we have only considered the situation where the production parameters are static. The utilities of productions can also be learned as the model runs based on rewards that are received by the model. When utility learning is enabled, the productions' utilities are updated according to a simple integrator model (e.g. see Bush & Mosteller, 1955). If U_i(n-1) is the utility of a production *i* after its n-1st application and R_i(n) is the reward the production receives for its nth application, then its utility U_i(n) after its nth application will be:

$$U_i(n) = U_i(n-1) + \alpha\left[R_i(n) - U_i(n-1)\right]$$

where α is the learning rate and is typically set at .2 (this can be changed by adjusting the

:alpha parameter in the model). This is also basically the Rescorla-Wagner learning rule (Rescorla & Wagner, 1972). According to this equation the utility of a production will be gradually adjusted until it matches the average reward that the production receives.

There are a couple of things to mention about the rewards. The rewards can occur at any time, and are not necessarily associated with any particular production. Also, a number of productions may have fired before a reward is delivered. The reward R_i(n) that production *i* will receive will be the external reward received minus the time from production *i*'s selection to the reward. This serves to give less reward to more distant productions. This is like the temporal discounting in reinforcement learning but proves to be more robust within the ACT-R architecture (not suggesting it is generally more robust). This reinforcement goes back to all of the productions which have been selected between the current reward and the previous reward.

There are two ways to provide rewards to a model: at any time the trigger-reward command can be used to provide a reward or rewards can be attached to productions and those rewards will be applied after the corresponding production fires. Attaching rewards to productions can be the more convenient way to provide rewards to a model when they correspond to situations which the model will explicitly process. For instance, in the building sticks task the rewards are provided when the model successfully completes a problem and when it has to reset and start over, and there are productions which handle those situations: read-done detects that it has completed the problem and pick-another-strategy is responsible for choosing again after resetting. One can associate rewards with these outcomes by setting the reward values of those productions:

```
(spp read-done :reward 20)
(spp pick-another-strategy :reward 0)
```

When read-done fires it will propagate a reward of 20 back to the previous productions which have been fired. Of course, productions earlier in the chain will receive smaller values because the time to the reward is subtracted from the reward. If pick-another-strategy fires, a reward of 0 will be propagated back – which means that previous productions will actually receive a negative reward because of the time that passed. Consider what happens when a sequence of productions leads to a dead end, pick-another-strategy fires, another sequence of productions fire that leads to a solution, and then read-done fires. The reward associated with read-done will propagate back only to the production which fired after pick-another-strategy and no further because the reward only goes back as far as the last reward. Note that the production read-done will receive its own reward, but pick-another-strategy will not receive any of read-done's reward since it will have received the reward from its own firing.

6.4 Learning in the Building Sticks Task

The following are the lengths of the sticks and the percent choice of overshoot for each of the problems in the testing set from an experiment with a building sticks task reported in Lovett & Anderson (1996):

a	b	c	Goal	%OVERSHOOT
15	250	55	125	20
10	155	22	101	67
14	200	37	112	20
22	200	32	114	47
10	243	37	159	87
22	175	40	73	20
15	250	49	137	80
10	179	32	105	93
20	213	42	104	83
14	237	51	116	13
12	149	30	72	29
14	237	51	121	27
22	200	32	114	80
14	200	37	112	73
15	250	55	125	53

The majority of these problems look like they can be solved by undershoot and in some cases the pixel difference is greater than 25. However, the majority of the problems can only be solved by overshoot. The first and last problems are interesting because they are identical and look strongly like they are undershoot problems. It is the only problem that can be solved either by overshoot or undershoot. Only 20% of the participants solve the first problem by overshoot but when presented with the same problem at the end of the experiment 53% use overshoot.

The model which ran the test trials above can also perform the whole experiment, and will show performance similar to the data through the utility learning mechanism which is enabled in the model by setting the :ul parameter to t. The bst-experiment function in Lisp and the experiment function in the bst module for Python will run the model through the experiment multiple times averaging the results. The following shows the performance of the model averaged over 100 iterations of the experiment:

```
CORRELATION:  0.803
MEAN DEVIATION: 17.129

Trial 1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
     23.0 60.0 59.0 70.0 91.0 42.0 80.0 86.0 59.0 34.0 33.0 22.0 54.0 72.0 56.0

DECIDE-OVER : 13.1506
DECIDE-UNDER: 11.1510
FORCE-OVER  : 12.1525
FORCE-UNDER : 6.5943
```

Also printed out are the average values of the utility parameters for the critical productions after each run through the experiment. As can be seen, the two over productions have increased their utility while the under productions have had a drop off. On average, the **force-over** production has a slightly higher value than the **decide-under** production. It is this change in utility values that creates the increased tendency to choose the overshoot strategy.

This model also turns on the utility learning trace, the :ult parameter, which works similar to the activation trace shown in the previous unit. If you enable the trace in the model by setting the :v parameter to **t** then every time there is a reward given to the model the trace will show the utility changes for all of the productions affected by that reward. Here is an example from a run showing the positive reward for successfully completing a trial:

```
     5.062   UTILITY                PROPAGATE-REWARD 20
 Utility updates with Reward = 20.0   alpha = 0.2
  Updating utility of production START-TRIAL
   U(n-1) = 0.0   R(n) = 14.938 [20.0 - 5.062 seconds since selection]
   U(n) = 2.9876
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 0.0   R(n) = 14.988 [20.0 - 5.012 seconds since selection]
   U(n) = 2.9976
  Updating utility of production ATTEND-LINE
   U(n-1) = 0.0   R(n) = 15.038 [20.0 - 4.962 seconds since selection]
   U(n) = 3.0076
  Updating utility of production ENCODE-LINE-A
   U(n-1) = 0.0   R(n) = 15.173 [20.0 - 4.827 seconds since selection]
   U(n) = 3.0346
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 2.9976   R(n) = 15.223 [20.0 - 4.777 seconds since selection]
   U(n) = 5.44268
  Updating utility of production ATTEND-LINE
   U(n-1) = 3.0076   R(n) = 15.273 [20.0 - 4.727 seconds since selection]
   U(n) = 5.46068
  Updating utility of production ENCODE-LINE-B
   U(n-1) = 0.0   R(n) = 15.423 [20.0 - 4.577 seconds since selection]
   U(n) = 3.0846002
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 5.44268   R(n) = 15.473 [20.0 - 4.527 seconds since selection]
   U(n) = 7.448744
  Updating utility of production ATTEND-LINE
   U(n-1) = 5.46068   R(n) = 15.523 [20.0 - 4.477 seconds since selection]
   U(n) = 7.473144
  Updating utility of production ENCODE-LINE-C
   U(n-1) = 0.0   R(n) = 15.658 [20.0 - 4.342 seconds since selection]
   U(n) = 3.1316001
  Updating utility of production FIND-NEXT-LINE
   U(n-1) = 7.448744   R(n) = 15.708 [20.0 - 4.292 seconds since selection]
   U(n) = 9.100595
  Updating utility of production ATTEND-LINE
   U(n-1) = 7.473144   R(n) = 15.757999 [20.0 - 4.242 seconds since selection]
   U(n) = 9.1301155
  Updating utility of production ENCODE-LINE-GOAL
   U(n-1) = 0.0   R(n) = 15.893 [20.0 - 4.107 seconds since selection]
   U(n) = 3.1786
  Updating utility of production ENCODE-UNDER
   U(n-1) = 0.0   R(n) = 16.028 [20.0 - 3.972 seconds since selection]
   U(n) = 3.2056
  Updating utility of production ENCODE-OVER
   U(n-1) = 0.0   R(n) = 16.163 [20.0 - 3.837 seconds since selection]
   U(n) = 3.2326
  Updating utility of production FORCE-UNDER
   U(n-1) = 10.0   R(n) = 16.213 [20.0 - 3.787 seconds since selection]
   U(n) = 11.2425995
  Updating utility of production MOVE-MOUSE
   U(n-1) = 0.0   R(n) = 16.263 [20.0 - 3.737 seconds since selection]
   U(n) = 3.2526002
  Updating utility of production CLICK-MOUSE
   U(n-1) = 0.0   R(n) = 16.713 [20.0 - 3.287 seconds since selection]
   U(n) = 3.3425999
  Updating utility of production LOOK-FOR-CURRENT
   U(n-1) = 0.0   R(n) = 17.063 [20.0 - 2.937 seconds since selection]
   U(n) = 3.4126
  Updating utility of production ATTEND-LINE
   U(n-1) = 9.1301155   R(n) = 17.112999 [20.0 - 2.887 seconds since selection]
   U(n) = 10.726692
  Updating utility of production ENCODE-LINE-CURRENT
   U(n-1) = 0.0   R(n) = 17.248 [20.0 - 2.752 seconds since selection]
   U(n) = 3.4496
  Updating utility of production CALCULATE-DIFFERENCE
   U(n-1) = 0.0   R(n) = 17.383 [20.0 - 2.617 seconds since selection]
   U(n) = 3.4766
  Updating utility of production CONSIDER-C
   U(n-1) = 0.0   R(n) = 17.433 [20.0 - 2.567 seconds since selection]
   U(n) = 3.4866002
  Updating utility of production CHOOSE-C
   U(n-1) = 0.0   R(n) = 17.568 [20.0 - 2.432 seconds since selection]
   U(n) = 3.5136
  Updating utility of production MOVE-MOUSE
   U(n-1) = 3.2526002   R(n) = 17.618 [20.0 - 2.382 seconds since selection]
   U(n) = 6.12568
  Updating utility of production CLICK-MOUSE
   U(n-1) = 3.3425999   R(n) = 17.668 [20.0 - 2.332 seconds since selection]
   U(n) = 6.2076797
  Updating utility of production LOOK-FOR-CURRENT
   U(n-1) = 3.4126   R(n) = 17.868 [20.0 - 2.132 seconds since selection]
   U(n) = 6.3036804
  Updating utility of production ATTEND-LINE
   U(n-1) = 10.726692   R(n) = 17.918 [20.0 - 2.082 seconds since selection]
   U(n) = 12.164953
  Updating utility of production ENCODE-LINE-CURRENT
   U(n-1) = 3.4496   R(n) = 18.053 [20.0 - 1.947 seconds since selection]
   U(n) = 6.37028
  Updating utility of production CALCULATE-DIFFERENCE
   U(n-1) = 3.4766   R(n) = 18.188 [20.0 - 1.812 seconds since selection]
   U(n) = 6.41888
  Updating utility of production CONSIDER-C
   U(n-1) = 3.4866002   R(n) = 18.238 [20.0 - 1.762 seconds since selection]
   U(n) = 6.43688
  Updating utility of production CONSIDER-A
   U(n-1) = 0.0   R(n) = 18.373 [20.0 - 1.627 seconds since selection]
   U(n) = 3.6746
  Updating utility of production CHOOSE-A
   U(n-1) = 0.0   R(n) = 18.508 [20.0 - 1.492 seconds since selection]
   U(n) = 3.7015998
  Updating utility of production MOVE-MOUSE
   U(n-1) = 6.12568   R(n) = 18.558 [20.0 - 1.442 seconds since selection]
   U(n) = 8.612144
  Updating utility of production CLICK-MOUSE
   U(n-1) = 6.2076797   R(n) = 19.045 [20.0 - 0.955 seconds since selection]
   U(n) = 8.775144
  Updating utility of production LOOK-FOR-CURRENT
   U(n-1) = 6.3036804   R(n) = 19.395 [20.0 - 0.605 seconds since selection]
   U(n) = 8.921945
  Updating utility of production ATTEND-LINE
   U(n-1) = 12.164953   R(n) = 19.445 [20.0 - 0.555 seconds since selection]
   U(n) = 13.620962
  Updating utility of production ENCODE-LINE-CURRENT
   U(n-1) = 6.37028   R(n) = 19.58 [20.0 - 0.42 seconds since selection]
   U(n) = 9.012224
  Updating utility of production CALCULATE-DIFFERENCE
```

```
   U(n-1) = 6.41888   R(n) = 19.715 [20.0 - 0.285 seconds since selection]
   U(n) = 9.078104
  Updating utility of production CHECK-FOR-DONE
   U(n-1) = 0.0   R(n) = 19.765 [20.0 - 0.235 seconds since selection]
   U(n) = 3.9529998
  Updating utility of production FIND-DONE
   U(n-1) = 0.0   R(n) = 19.815 [20.0 - 0.185 seconds since selection]
   U(n) = 3.963
  Updating utility of production READ-DONE
   U(n-1) = 0.0   R(n) = 19.95 [20.0 - 0.05 seconds since selection]
   U(n) = 3.9900002
```

6.5 Additional Chunk-type Capabilities

Before discussing the assignment task for this unit we will look at a few of the productions in this model which appear to be doing things differently than previous units.

6.5.1 Default chunk-type slot values

If we look at the actions of the productions encode-line-goal and click-mouse we see that they seem to be missing the cmd slot in the requests to the **visual** and **manual** buffers which we have seen previously, and instead are only declaring a chunk-type with an isa:

```
(p encode-line-goal
...
  ==>
...
   +visual>
      isa        move-attention
      screen-pos =c)


(p click-mouse
...
  ==>
...
   +manual>
      isa    click-mouse)
```

Up to this point it has been stated that the isa declarations are optional and not a part of a request or condition. If that is true then how are those requests doing the right thing since we've also said that **visual** and **manual** requests require the cmd slot be specified to indicate the action to perform?

The answer to that question is that there is an additional process which happens when declaring a chunk-type with isa both in creating chunks and in specifying the conditions and actions of a production. That additional process relies on the ability to indicate default values for a slot when creating a chunk-type. Up until now when we have created chunk-types we have only specified the set of slots which it can include, but in addition to that one can specify default initial values for specific slots. If a chunk-type indicates that a slot should have a value by default, then when that chunk-type is declared any slots with default values that are not specified in the chunk definition or production statement will automatically be included with their default values.

To specify a default value for a slot in a chunk-type one needs to specify a list of two items for the slot where the first item is the slot name and the second is the default value, instead of just a slot name. Here are some example chunk-types for reference:

```
(chunk-type a slot)
(chunk-type b (slot 1))
(chunk-type c slot (slot2 t))
```

The chunk-type b has a default value of 1 for the slot named slot and chunk-type c has a default value of t for the slot named slot2. If we create some chunks specifying those chunk-types, some of which include a value for the slots with default values and others which do not, we can see how the default values are filled in for the chunks that do not specify those slots:

```
? (define-chunks (c1 isa a)
                 (c2 isa b)
                 (c3 isa b slot 3)
                 (c4 isa c)
                 (c5 isa c slot2 nil))


? (pprint-chunks C1 C2 C3 C4 C5)
C1

C2
   SLOT  1

C3
   SLOT  3

C4
   SLOT2  T

C5
```

For the slots with default values the chunks which specify the slot get the value specified, whereas the ones that do not get the default value. The same process applies to conditions and actions in a production. The chunk-types move-attention and click-mouse are created by the vision and motor modules and include default values for the cmd slot essentially like this (there is more to the real chunk-type specifications which will be discussed in the next section):

```
(chunk-type move-attention (cmd move-attention) screen-pos)
```

```
(chunk-type click-mouse (cmd click-mouse))
```

Therefore, this request:

```
   +visual>
      isa        move-attention
      screen-pos =c
```

is equivalent to this:

```
   +visual>
      cmd        move-attention
      screen-pos =c
```

and we can see that by looking at the actual representation of the production which the procedural module has for encode-line-goal using the Procedural viewer in the ACT-R Environment or by using the pp (print production) command:

```
? (pp encode-line-goal)
(P ENCODE-LINE-GOAL
   =GOAL>
       STATE ATTENDING
   =IMAGINAL>
       C-LOC =C
       GOAL-LOC NIL
   =VISUAL>
       SCREEN-POS =POS
       WIDTH =LENGTH
       LINE T
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       GOAL-LOC =POS
       LENGTH =LENGTH
   =GOAL>
       STATE ENCODE-UNDER
   +VISUAL>
       SCREEN-POS =C
       CMD MOVE-ATTENTION
)
```

Notice how none of the isa declarations from the original definition are part of the actual production representation but the cmd slot has been included in the request to the **visual** buffer.

The same thing holds for the manual request in the click-mouse production:

```
(P CLICK-MOUSE
   =GOAL>
       STATE MOVE-MOUSE
   ?MANUAL>
       STATE FREE
 ==>
   =GOAL>
       STATE WAIT-FOR-CLICK
   +MANUAL>
       CMD CLICK-MOUSE
)
```

6.5.2 Chunk-type hierarchy

Now we will look at some of the isa declarations on the LHSs of the encode-line-goal and read-done productions.

```
(p encode-line-goal
   =goal>
      isa        try-strategy
      state      attending
   =imaginal>
      isa        encoding
      c-loc      =c
      goal-loc   nil
   =visual>
      isa        line
      screen-pos =pos
      width      =length
   ?visual>
      state      free
  ==>
   =imaginal>
      goal-loc   =pos
      length     =length
   =goal>
      state      encode-under
   +visual>
      isa        move-attention
      screen-pos =c)

(p read-done
   =goal>
```

```
      isa    try-strategy
      state  read-done
   =visual>
      isa    text
      value  "done"
  ==>
   +goal>
      isa    try-strategy
      state  start)
```

Previously in the tutorial it was stated that the chunks placed into the **visual** buffer are created with slots from the chunk-type named visual-object. However, if we look at the conditions in those productions they are specifying chunk-types of line and text respectively in the **visual** buffer conditions.

In addition to the chunk-type visual-object the vision module defines some other chunk-types for the objects which are used by the AGI experiment windows. Those other chunk-types, like line and text, are actually subtypes of the type visual-object. A subtype contains all of the slots that its parent chunk-type contains, but may also contain additional slots or different default slot values for the slots which it has. When the vision module encodes a feature from an AGI window into a chunk it actually uses the slots of one of the specific types, like line or text to create the chunk.

In general one can create an arbitrary hierarchy of chunk-types with each subtype inheriting the slots and default values of its parent chunk-type (or even multiple parent chunk-types). A hierarchy of chunk-types can be helpful to the modeler in specifying chunks and productions, but other than through the inclusion of default slot values, such a hierarchy has no effect on the actual chunks or productions in the model.

To create a chunk-type which is a subtype of another chunk-type that parent type must be specified in the definition of the subtype. That is done by using a list to specify the name of the subtype which includes a list which has the keyword :include and the name of a parent chunk-type for each parent type to be included. These chunk-type definitions create chunk-types a and b, and then a chunk-type named c which is a subtype of chunk-type a and a chunk-type d which is a subtype of both a and b:

```
(chunk-type a slot1)
(chunk-type b slot2)
(chunk-type (c (:include a)) slot3)
(chunk-type (d (:include a) (:include b)) slot4)
```

One might think these definitions would be equivalent to those shown above:

```
(chunk-type a slot1)
(chunk-type b slot2)
(chunk-type c slot1 slot3)
```

```
(chunk-type d slot1 slot2 slot4)
```

However they are not because the subtyping mechanism is also extending the slots which are valid for the parent types as well. In addition to a subtype inheriting the slots from its parent types, the parent types also gain access to all of the slots from their children. Thus, with the definitions using the hierarchy it is acceptable to specify a parent type and use slots defined in its subtypes, but that is not valid for chunks which just happen to have the same slot names. Thus, with the first set of chunk-type definitions this is acceptable:

```
(define-chunks (isa a slot3 t slot4 10))
```

because chunk-types c and d are subtypes of chunk-type a and have slots named slot3 and slot4, but with the second set that would result in warnings for invalid slots in the chunk definition.

Now, for the line and text chunk-types used in these productions here are the relevant chunk-type definitions which the vision module has:

```
(chunk-type visual-object screen-pos value status color height width)
(chunk-type (text (:include visual-object)) (text t))
(chunk-type (line (:include visual-object)) (line t) end1-x end1-y
            end2-x end2-y)
```

The text and line chunk-types are subtypes of the visual-object type and each includes a new slot with the same name as the type and a default value of t. Because of that default slot value, declaring the buffer tests with the types text and line in these productions adds that additional condition to the productions as we can see when we look at the actual representation of them:

```
 (P ENCODE-LINE-GOAL
   =GOAL>
       STATE ATTENDING
   =IMAGINAL>
       C-LOC =C
       GOAL-LOC NIL
   =VISUAL>
       SCREEN-POS =POS
       WIDTH =LENGTH
       LINE T
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       GOAL-LOC =POS
       LENGTH =LENGTH
   =GOAL>
       STATE ENCODE-UNDER
   +VISUAL>
       SCREEN-POS =C
       CMD MOVE-ATTENTION
)
(P READ-DONE
   =GOAL>
       STATE READ-DONE
   =VISUAL>
       VALUE "done"
       TEXT T
 ==>
   +GOAL>
       STATE START
)
```

Those slot tests could have been specified directly, but declaring the type can be more readable and is consistent with the way chunk-types were used in older versions of ACT-R (those prior to version 6.1).

In the default slot section above it showed chunk-type definitions for the move-attention and click-mouse actions to indicate they had default slots, but the actual definitions of those chunk-types also include parent types which could be used in the requests as well:

```
(chunk-type vision-command cmd)
(chunk-type (move-attention (:include vision-command))
            (cmd move-attention) screen-pos scale)

(chunk-type motor-command (cmd "motor action"))
(chunk-type (click-mouse (:include motor-command)) (cmd click-mouse))
```

6.6 Learning in a Probability Choice Experiment

Your assignment is to develop a model for a "probability matching" experiment run by Friedman et al (1964). However, unlike the assignments for previous units, you are not provided with the code that implements the experiment this time. Therefore you will need to first write the experiment, and then develop the model, which more closely represents the typical modeling situation. The experiment to be implemented is very simple. Here is the basic procedure which is repeated for 48 trials:

1. The participant is presented with a screen saying "Choose"

2. The participant either presses the ‘h’ key for heads or the ‘t’ key for tails

3. When the key is pressed the screen is cleared and the feedback indicating the correct answer, either "Heads" or "Tails", is displayed.

4. That feedback stays on the screen for exactly 1 second before the next trial is presented.

Friedman et al arranged it so that heads was the correct choice on 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% of the trials (independent of what the participant had done). For your experiment you will only be concerned with the 90% condition. Thus, your experiment will be 48 trials long and “Heads” will be the correct answer 90% of the time. We have averaged together the data from the 10% and 90% conditions (flipping responses) to get an average proportion of choice of the dominant answer in blocks of 12 trials. These proportions are 0.664, 0.778, 0.804, and 0.818. This is the data that your model is to fit. It is important to note that this is the **proportion of choice for heads**, not the proportion of correct responses – the correctness of the response does not matter.

Your model must begin with a 50% chance of saying heads, then based on the feedback from the experiment it must adjust its choice through utility learning so that it averages responding heads close to 66% over the first block of 12 trials, and increases to about 82% by the final block. You will run the model through the experiment many times (resetting before each experiment) and average the data of those runs for comparison. As an aspiration level, this is the performance of the model that I wrote, averaged over 100 runs:

```
CORRELATION:  0.994
MEAN DEVIATION:  0.016
 Original     Current
   0.664       0.646
   0.778       0.771
   0.804       0.826
   0.818       0.830
```

In achieving this, the parameters I worked with were the noise in the utilities (set by the :egs parameter) and the rewards associated with successful and unsuccessful responses.

The starting model for this task, found in the choice-model.lisp file of the unit, only contains the calls to clear ACT-R to its initial state and create a model named choice which has no content. You will need to write the whole model.

The initial code for this task can be found in the choice.lisp and choice.py files. It contains the call to load the starting model, specifies the experimental data, a global variable for collecting a response, and includes two functions. The first function is used as a monitor for the output-key action and just sets a global variable to record the response (as was done in many of the previous tasks). The other function, called choice-person in the Lisp version and person in the Python version, is a correct implementation of the task described for running a person through a single trial, and it returns the key which was pressed.

You should write a similar function to run the model through one trial, which should be named **choice-model** (in Lisp) or **model** (in Python). You will also need to write a function that takes one parameter and runs the whole experiment that many times and prints out the average results of the runs and the correlation and deviation of the average data to the experimental data. That function should be named **choice-data** (in Lisp) and **data** (in Python). That function does not have to be able to run a person through the task. It only needs to be able to run the model.

My suggestion would be to first write the single trial function making sure that it correctly represents the experiment described, including the timing. Then write a model that is able to perform that task correctly. Next write a function to run a block of 12 trials and test that to make sure the model works correctly when going from trial to trial. Then write a function to iterate over 4 blocks for running one pass of the experiment and test that. After that is working write the function to run the experiment multiple times. Only then should you be concerned with actually fitting the model to the data, once you are sure everything else works.

To write the experiment for the model you will need to use some of the ACT-R commands that were discussed in the previous units' code texts in addition to those already used in the function for running a person. The necessary commands will be described again here briefly, and the experiments you have seen up to this point should provide plenty of examples of their use.

The **reset** function initializes ACT-R. It returns the model to the initial state as specified in the model file. It is the programmatic equivalent of pressing the "Reset" button in the ACT-R Environment.

The **run** function can be used to run the model until either it has nothing to do or the specified amount of time has passed. It has one required parameter, which is the maximum amount of time to run the model in seconds.

The **run-full-time/run_full_time** function can be used to run the model for a specific amount of time. It takes one parameter which is the amount of time to run the model in seconds.

The **install-device/install_device** function takes one parameter which specifies a device the model will interact with and the value returned from opening the experiment window is the device of interest for this task.

In addition to those functions you will also want to use the **correlation** and **mean-deviation/mean_deviation** functions. Those calculate the correlation and mean-deviation between two lists of numbers.

6.6.1 Example experiment functions

The paired associate task from unit 4 is a good example to look at for creating your experiment, and the do-experiment function in the Lisp version and the do_experiment function in the Python version have a very similar structure to what you will need for presenting a trial of this choice experiment. The paired associate functions are a little more complicated than the function you will need for this assignment because they can run either a person or the model and are also recording response times and averaging the data over multiple runs which your single trial function will not need to do. They also call **reset** which you should not do in your single trial function because you want the model to continue to learn from trial to trial. You should only call **reset** at the start of each pass through the whole experiment. Ignoring those complications, it performs a similar sequence of operations to those necessary to do this experiment: open a window, tell the model to interact with that window, present an item of text, run the model, clear the screen, display another item of text, and then run the model again. Those functions are copied here and the relevant operations are highlighted in green.

Lisp

```
 (defun do-experiment (size trials human)

  (if (and human (not (visible-virtuals-available?)))
      (print-warning "Cannot run the task as a person without a visible window available.")
    (progn
      (reset)

      (let* ((result nil)
             (model (not human))
             (window (open-exp-window "Paired-Associate Experiment" :visible human)))

        (when model
          (install-device window))

        (dotimes (i trials)
          (let ((score 0.0)
                (time 0.0))

            (dolist (x (permute-list (subseq *pairs* (- 20 size))))

              (clear-exp-window window)
              (add-text-to-exp-window window (first x) :x 150 :y 150)

              (setf *response* nil)
              (let ((start (get-time model)))

                (if model
                    (run-full-time 5)
                  (while (< (- (get-time nil) start) 5000)
                    (process-events)))

                (when (equal *response* (second x))
                  (incf score 1.0)
                  (incf time (- *response-time* start)))

                (clear-exp-window window)
                (add-text-to-exp-window window (second x) :x 150 :y 150)
                (setf start (get-time model))

                (if model
                    (run-full-time 5)
                  (while (< (- (get-time nil) start) 5000)
                    (process-events)))))

            (push (list (/ score size) (if (> score 0) (/ time score 1000.0) 0)) result)))

        (reverse result)))))
```

Python

```
def do_experiment(size, trials, human):

    if human and not(actr.visible_virtuals_available()):
        actr.print_warning("Cannot run the task as a person without a visible window.")
    else:

        actr.reset()

        result = []
        model = not(human)
        window = actr.open_exp_window("Paired-Associate Experiment",visible=human)

        if model:
            actr.install_device(window)

        for i in range(trials):
            score = 0
            time = 0

            for prompt,associate in actr.permute_list(pairs[20 - size:]):

                actr.clear_exp_window(window)
                actr.add_text_to_exp_window (window, prompt, x=150 , y=150)

                global response
                response = ''
                start = actr.get_time(model)

                if model:
                    actr.run_full_time(5)
                else:
                    while (actr.get_time(False) - start) < 5000:
                        actr.process_events()

                if response == associate:
                    score += 1
                    time += response_time - start

                actr.clear_exp_window(window)
                actr.add_text_to_exp_window (window, associate, x=150 , y=150)
                start = actr.get_time(model)

                if model:
                    actr.run_full_time(5)
                else:
                    while (actr.get_time(False) - start) < 5000:
                        actr.process_events()

            if score > 0:
                average_time = time / score / 1000.0
            else:
                average_time = 0

            result.append((score/size,average_time))

        return result
```

In the choice files provided, the function for running a person provides the general structure for performing a trial in this task: it opens a window, creates a monitor for recording the key press, adds the choose prompt to the screen, clears that prompt, and then displays the feedback. However, instead of running a model it waits for a person to press the key and waits for 1 second of real time to pass.

Lisp

```
(defun choice-person ()
  ~~(when (visible-virtuals-available?)~~
    (let ((window (open-exp-window "Choice Experiment" :visible t)))

      (add-act-r-command "choice-response" 'respond-to-key-press "Choice task key response")
      (monitor-act-r-command "output-key" "choice-response")

      (add-text-to-exp-window window "choose" :x 50 :y 100)

      (setf *response* nil)

      (while (null *response*)
        (process-events))

      (clear-exp-window window)

       (add-text-to-exp-window window (if (< (act-r-random 1.0) .9) "heads" "tails") :x 50 :y
100)

      (let ((start (get-time nil)))

        (while (< (- (get-time nil) start) 1000)
          (process-events)))

      (remove-act-r-command-monitor "output-key" "choice-response")
      (remove-act-r-command "choice-response")

      *response*)))
```

Python

```
def person():
    global response

    ~~if actr.visible_virtuals_available():~~
        window = actr.open_exp_window("Choice Experiment",visible=True)

        actr.add_command("choice-response",respond_to_key_press,"Choice task key response")
        actr.monitor_command("output-key","choice-response")

        actr.add_text_to_exp_window (window, 'choose', x=50, y=100)
```

```
response = ''

while response == '':
    actr.process_events()

actr.clear_exp_window(window)

if actr.random(1.0) < .9:
    answer = 'heads'
else:
    answer = 'tails'

actr.add_text_to_exp_window (window, answer, x=50, y=100)

start = actr.get_time(False)

while (actr.get_time(False) - start) < 1000:
    actr.process_events()

actr.remove_command_monitor("output-key","choice-response")
actr.remove_command("choice-response")

return response
```

What you must do is write the corresponding function that has the appropriate interaction for the ACT-R model to perform the task. The code with a line through it is only a safety test for running a person and should **not** be included in your model running version. The code colored red above handles the interaction for a person doing the task, and that is **not** the same as what will be needed to run a model. It should be **replaced** with the appropriate code for the model to interact with the task, which will be similar to the green code from the paired example above (however the timing for the paired task is not the same as it is in this task so you will have to adjust that appropriately).

Something else to think about is that the exact placement of the choose prompt and the feedback of heads and tails is not specified in the description of the task. Therefore, your model should not assume anything about their locations i.e. your model should still be able to do the task regardless of where on the screen the choose prompt and the feedback occur. In the given function for running a person the answer is presented in the same location as the word "choose", but your model should also be able to perform the task if they are presented in different locations.

One final thing to note is that the paired associate task is an example of the "trial at a time" approach to building an experiment as discussed in the unit 4 code documentation, and that is probably the easiest way to approach this task. However, it is also possible to write this experiment using the "event-driven" style which was also discussed in the unit 4 code documentation. If you want to use that approach it will require a little more work to program because it does not analogize as neatly to one of the previous units' tasks. If you would like to try to write the experiment in that way you should look at the zbrodoff experiment as an example instead of the paired associate experiment. In fact, the different ways to write the experiment can actually have an effect on the data fitting for this model because they will likely have slightly different timing on the events which will affect the rewards received by the productions. For the paired associate task the style of the experiment was not an issue because the lengths of the trials were fixed, but in this case, because the trials are supposed to transition when the key press occurs, an event-driven experiment will provide a more veridical timing sequence because the events of the experiment will not be affected by components of the model other than its response. However, since you will be fitting the parameters in your model to the task you have written, that difference should not really matter, and either approach is acceptable for the assignment.

References

Bush, R. R., & Mosteller, F. (1955). *Stochastic Models for Learning*. New York: Wiley.

Friedman, M. P., Burke, C. J., Cole, M., Keller, L., Millward, R. B., & Estes, W. K., (1964). Two-choice behavior under extended training with shifting probabilities of reinforcement. In R. C. Atkinson (Ed.), *Studies in mathematical psychology* (pp. 250-316). Stanford, CA: Stanford University Press.

Lovett, M. C., & Anderson, J. R. (1996). History of success and current context in problem solving: Combined influences on operator selection. *Cognitive Psychology, 31,* 168-217.

Rescorla, R. A., & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations on the effectiveness of reinforcement and nonreinforcement. In A. H. Black & W. R. Prokasy (eds.), *Classical Conditioning: II. Current Research and Theory* (pp. 64-99). New York: Appleton-Century-Crofts.

Unit 6 Code Description

The assignment for this unit is to write a complete experiment and model essentially from scratch. The demonstration experiment for this unit is much more complicated than the one needed for the assignment task. While it does provide information on creating more involved experiments for models, the models from earlier units (like the paired associate task in unit 4) will be more useful examples in completing the assignment for this unit.

In the Building Sticks Task (BST) there is not a simple response collected from the user, but instead an ongoing interaction that only ends when the correct results are achieved. This requires some new experiment generation functions and it is written in a mixture of the "trial at a time" and event-driven styles. Each BST problem presented is a trial which is run as an event-driven experiment, and the model is iterated over those trials one at a time. The reason for using the event-driven approach for each problem is because the task is performed by pressing buttons and a button can have a function associated with it to call when it is pressed. The changes to the display can be performed by those functions as the model runs instead of having to stop it on each press, update the display, and then start running the model again. In this situation it is actually less complicated to write the event-driven task than it would be to build something that runs the model for each single action performed.

Lisp

Start by loading the model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit6;bst-model.lisp")
```

Create some global variables to hold the experiment information: the length of the target stick, the length of the current stick, the screen object for the current line, a flag for whether the task is complete, whether the person started with the over- or under- shoot strategy, the window for the experiment, and whether the window should be visible or virtual.

```
(defvar *target*)
(defvar *current-stick*)
(defvar *current-line*)
(defvar *done*)
(defvar *choice*)
(defvar *window* nil)
(defvar *visible* nil)
```

Some more global variables to hold the experimental data on choice of overshoot, the lengths of the sticks for the full experiment, and the lengths of the sticks for the 2 trial tests.

```
(defvar *bst-exp-data* '(20.0 67.0 20.0 47.0 87.0 20.0 80.0 93.0
                         83.0 13.0 29.0 27.0 80.0 73.0 53.0))

(defvar *exp-stims* '((15  250  55  125)(10  155  22  101)
                      (14  200  37  112)(22  200  32  114)
                      (10  243  37  159)(22  175  40  73)
                      (15  250  49  137)(10  179  32  105)
                      (20  213  42  104)(14  237  51  116)
                      (12  149  30  72)
                      (14  237  51  121)(22  200  32  114)
                      (14  200  37  112)(15  250  55  125)))
```

```
(defvar *no-learn-stims* '((15  200  41 103)(10  200 29 132)))
```

The build-display function takes 4 parameters which are the lengths of the three given sticks and the length of the goal stick. It sets the global variables appropriately, opens a window, and then draws the starting information for the task.

```
(defun build-display (a b c goal)

  (setf *target* goal)
  (setf *current-stick* 0)
  (setf *done* nil)
  (setf *choice* nil)
  (setf *current-line* nil)
  (setf *window* (open-exp-window "Building Sticks Task" :visible *visible*
                                  :width 600 :height 400))
```

Add-button-to-exp-window is a new function for this unit and will be described in more detail at the end of this document. For now, the important thing to know is that when the button is created it can be associated with a command to call when it is pressed. The action parameter to the function can be a string which names a command or a list of a string which names a command and additional values. If it is just a command then that command will be called with no parameters when the button is pressed, and if it is a list then the command specified will be passed the other values in the list when the button is pressed. Thus, with the buttons created below, when the button labeled “Reset” is pressed it will call the “bst-reset-button-pressed” command with no parameters and when the button labeled “A” is pressed it will call the “bst-button-pressed” command with two parameters, the first being the length held in the variable a and the other being the symbol under.

```
  (add-button-to-exp-window *window* :text "A" :x 5 :y 23
                            :action (list "bst-button-pressed" a "under") :height 24 :width 40)
  (add-button-to-exp-window *window* :text "B" :x 5 :y 48
                            :action (list "bst-button-pressed" b "over") :height 24 :width 40)
  (add-button-to-exp-window *window* :text "C" :x 5 :y 73
                            :action (list "bst-button-pressed" c "under") :height 24 :width 40)
  (add-button-to-exp-window *window* :text "Reset" :x 5 :y 123
                            :action "bst-reset-button-pressed" :height 24 :width 65)

  (add-line-to-exp-window *window* (list 75 35)  (list (+ a 75) 35) 'black)
  (add-line-to-exp-window *window* (list 75 60)  (list (+ b 75) 60) 'black)
  (add-line-to-exp-window *window* (list 75 85)  (list (+ c 75) 85) 'black)
  (add-line-to-exp-window *window* (list 75 110) (list (+ goal 75) 110) 'green))
```

The button-pressed function is associated with the A, B, and C buttons created above. It will be called when those buttons are pressed and passed the length of the corresponding stick and whether that button represents the over- or under- shoot strategy if it is the first one chosen.

```
(defun button-pressed (len dir)
```

If a previous strategy choice hasn't been set, then set it to the current strategy.

```
  (unless *choice*
    (setf *choice* dir))
```

If the trial isn't over yet then update the current stick using the chosen one.

```
  (unless *done*
    (if (> *current-stick* *target*)
        (decf *current-stick* len)
      (incf *current-stick* len))
    (update-current-line)))
```

The reset-display function gets called when the reset button is pressed and if the trial isn't over yet it will remove the current stick so the participant can start over.

```
(defun reset-display ()
  (unless *done*
    (setf *current-stick* 0)
    (update-current-line)))
```

Here we add the commands for the previous two functions so that they can be called as button actions.

```
(add-act-r-command "bst-button-pressed" 'button-pressed
                   "Choice button action for the Building Sticks Task.  Do not call directly")
(add-act-r-command "bst-reset-button-pressed" 'reset-display
                   "Reset button action for the Building Sticks Task.  Do not call directly")
```

Update-current-line redraws the current line and updates the global variables which hold the current state as necessary.

```
(defun update-current-line ()
```

If the current stick has the target length the task is over. Set the done flag, modify the line to show its new length, and display the done prompt.

```
  (cond ((= *current-stick* *target*)
         (setf *done* t)
         (modify-line-for-exp-window *current-line* (list 75 135) (list (+ *target* 75) 135))
         (add-text-to-exp-window *window* "Done" :x 180 :y 200))
```

If the current stick is returned to length 0 then remove it from the screen and clear the variable holding the line.

```
        ((zerop *current-stick*)
         (when *current-line*
           (remove-items-from-exp-window *window* *current-line*)
           (setf *current-line* nil)))
```

If there is a current stick then update it to the new length.

```
        (*current-line*
          (modify-line-for-exp-window *current-line* (list 75 135)
                                      (list (+ *current-stick* 75) 135)))
```

Otherwise, draw a new current stick of the appropriate length and save that item.

```
        (t
         (setf *current-line* (add-line-to-exp-window *window* (list 75 135)
                                                      (list (+ *current-stick* 75) 135)
                                                      'blue)))))
```

The do-experiment function takes a list of lengths which represent the sticks to present on a trial and an optional parameter indicating whether a person will be performing the task. It calls build-display to create the window and then either waits for a person to finish the task or runs the model for up to 60 seconds to perform the task (in real time if the window is visible for the model). When running the model, since this task requires using the mouse it first uses start-hand-at-mouse to put the model's hand on the virtual mouse instead of the virtual keyboard since the keyboard is the default location for the hands when using the experiment window device.

```
(defun do-experiment (sticks &optional human)
  (apply 'build-display sticks)
  (if human
      (when (visible-virtuals-available?)
        (wait-for-human))
    (progn
      (install-device *window*)
      (start-hand-at-mouse)
      (run 60 *visible*))))
```

The wait-for-human function loops until the task is over and then waits one second more after displaying the done prompt.

```
(defun wait-for-human ()
  (while (not *done*)
    (process-events))
  ;; wait for 1 second to pass after done
  (let ((start-time (get-time nil)))
    (while (< (- (get-time nil) start-time) 1000)
      (process-events))))
```

The bst-set function takes three required parameters which indicate whether a person is doing the task, whether the window should be visible, and a list of stick length lists for the trials to present. It also takes an optional parameter which indicates whether the model should be learning from trial to trial or not. It loops over the trials provided, resetting the model if it isn't supposed to learn from trial to trial, and records the initial strategy choice which was made on each one. It returns the list of strategy choices.

```
(defun bst-set (human visible stims &optional (learn t))
  (setf *visible* visible)
  (let ((result nil))
    (dolist (stim stims)
      (when (and (not human) (not learn))
        (reset))
      (do-experiment stim human)
      (push *choice* result))
    (reverse result)))
```

The bst-test function takes one required parameter which is the number of two-trial test runs to perform and an optional parameter indicating whether a person will be performing the task. It runs the required number of trials and counts the number of times over-shoot is used to solve them returning the list of those counts.

```
(defun bst-test (n &optional human)
  (let ((stims *no-learn-stims*))

    (let ((result (make-list (length stims) :initial-element 0)))
      (dotimes (i n result)
        (setf result (mapcar '+
                       result
                       (mapcar (lambda (x)
                                 (if (string-equal x "over") 1 0))
                         (bst-set human (or human (= n 1)) stims nil))))))))
```

The bst-experiment function takes one required parameter which is the number of times to perform the whole experiment and an optional parameter indicating whether a person will be performing the task. It runs the required number of experiments and computes the average proportion of times over-shoot is chosen for each trial and the average utility value of the critical productions in the model after each experiment. Those results are printed along with the correlation and mean-deviation from the original experimental data.

```
(defun bst-experiment (n &optional human)
  (let ((stims *exp-stims*))

    (let ((result (make-list (length stims) :initial-element 0))
          (p-values (list '(decide-over 0) '(decide-under 0) '(force-over 0) '(force-under 0))))
      (dotimes (i n result)
        (reset)
        (setf result (mapcar '+ result
                       (mapcar (lambda (x)
                                 (if (string-equal x "over") 1 0))
                         (bst-set human human stims))))
        (no-output
         (setf p-values (mapcar (lambda (x)
                                  (list (car x) (+ (second x) (production-u-value (car x)))))
                          p-values))))

      (setf result (mapcar (lambda (x) (* 100.0 (/ x n))) result))

      (when (= (length result) (length *bst-exp-data*))
        (correlation result *bst-exp-data*)
        (mean-deviation result *bst-exp-data*))

      (format t "~%Trial ")

      (dotimes (i (length result))
        (format t "~8s" (1+ i)))

      (format t "~%  ~{~8,2f~}~%~%" result)

      (dolist (x p-values)
        (format t "~12s: ~6,4f~%" (car x) (/ (second x) n))))))
```

The production-u-value function takes one parameter which should be the name of a production. It returns the current value of the :u parameter for that production.

```
(defun production-u-value (prod)
   (caar (spp-fct (list prod :u))))
```

Python

Start by importing the actr interface module and loading the model which can perform the task.

```
import actr

actr.load_act_r_model ("ACT-R:tutorial;unit6;bst-model.lisp")
```

Create some global variables to hold the experiment information: the length of the target stick, the length of the current stick, the screen object for the current line, a flag for whether the task is complete, whether the person started with the over- or under- shoot strategy, the window for the experiment, and whether the window should be visible or virtual.

```
target = None
current_stick = None
current_line = None
done = False
choice = None
window = None
visible = False
```

Some more global variables to hold the experimental data on choice of overshoot, the lengths of the sticks for the full experiment, and the lengths of the sticks for the 2 trial tests.

```
exp_data = [20, 67, 20, 47, 87, 20, 80, 93, 83, 13, 29, 27, 80, 73, 53]
exp_stims = [[15,250,55,125],[10,155,22,101],[14,200,37,112],
             [22,200,32,114],[10,243,37,159],[22,175,40,73],
             [15,250,49,137],[10,179,32,105],[20,213,42,104],
             [14,237,51,116],[12,149,30,72],[14,237,51,121],
             [22,200,32,114],[14,200,37,112],[15,250,55,125]]

no_learn_stims = [[15,200,41,103],[10,200,29,132]]
```

The build_display function takes 4 parameters which are the lengths of the three given sticks and the length of the goal stick. It sets the global variables appropriately, opens a window, and then draws the starting information for the task.

```
def build_display (a,b,c,goal):
    global window,target,current_stick,done,current_line,choice

    target = goal
    current_stick = 0
    done = False
    choice = None
    current_line = None
    window = actr.open_exp_window("Building Sticks Task",visible=visible,
                                  width=600,height=400)
```

Add_button_to_exp_window is a new function for this unit and will be described in more detail at the end of this document. For now, the important thing to know is that when the button is created it can be associated with a command to call when it is pressed. The action parameter to the function can be a string which names a command or a list of a string which names a command and additional values. If it is just a command then that command will be called with no parameters when the button is pressed, and if it is a list then the command specified will be passed the other values in the list when the button is pressed. Thus, with the buttons created below, when the button labeled “Reset” is pressed it will call the “bst-reset-button-pressed” command with no parameters and when the button labeled “A” is pressed it will call the “bst-button-pressed” command with two parameters, the first being the length held in the variable a and the other being the string under.

```
    actr.add_button_to_exp_window(window, text="A", x=5, y=23,
                                  action=["bst-button-pressed",a,"under"],
                                  height=24, width=40)
    actr.add_button_to_exp_window(window, text="B", x=5, y=48,
                                  action=["bst-button-pressed",b,"over"],
                                  height=24, width=40)
    actr.add_button_to_exp_window(window, text="C", x=5, y=73,
                                  action=["bst-button-pressed",c,"under"],
                                  height=24, width=40)
    actr.add_button_to_exp_window(window, text="Reset", x=5, y=123,
                                  action="bst-reset-button-pressed",
                                  height=24, width=65)

    actr.add_line_to_exp_window(window,[75,35],[a + 75,35],"black")
    actr.add_line_to_exp_window(window,[75,60],[b + 75,60],"black")
    actr.add_line_to_exp_window(window,[75,85],[c + 75,85],"black")
    actr.add_line_to_exp_window(window,[75,110],[goal + 75,110],"green")
```

The button_pressed function is associated with the A, B, and C buttons created above. It will be called when those buttons are pressed and passed the length of the corresponding stick and whether that button represents the over- or under- shoot strategy if it is the first one chosen.

```
def button_pressed(len,dir):
    global choice,current_stick
```

If a previous strategy choice hasn't been set, then set it to the current strategy.

```
    if not(choice):
        choice = dir
```

If the trial isn't over yet then update the current stick using the chosen one.

```
    if not(done):
        if current_stick > target:
            current_stick -= len
        else:
            current_stick += len

        update_current_line()
```

The reset_display function gets called when the reset button is pressed and if the trial isn't over yet it will remove the current stick so the participant can start over.

```
def reset_display():
    global current_stick

    if not(done):
        current_stick = 0
        update_current_line()
```

Here we add the commands for the previous two functions so that they can be called as button actions.

```
actr.add_command("bst-button-pressed",button_pressed,
                 "Choice button action for the Building Sticks Task.  Do not call directly")
actr.add_command("bst-reset-button-pressed",reset_display,
                 "Reset button action for the Building Sticks Task.  Do not call directly")
```

Update_current_line redraws the current line and updates the global variables which hold the current state as necessary.

```
def update_current_line():
    global current_line,done
```

If the current stick has the target length the task is over. Set the done flag, modify the line to show its new length, and display the done prompt.

```
if current_stick == target:
    done = True
    actr.modify_line_for_exp_window(current_line,[75,135],[target + 75,135])
    actr.add_text_to_exp_window(window,"Done", x=180, y=200)
```

If the current stick is returned to length 0 then remove it from the screen and clear the variable holding the line.

```
elif current_stick == 0:
    if current_line:
        actr.remove_items_from_exp_window(window,current_line)
        current_line = None
```

If there is a current stick then update it to the new length.

```
elif current_line:
    actr.modify_line_for_exp_window(current_line,[75,135],[current_stick + 75,135])
```

Otherwise, draw a new current stick of the appropriate length and save that item.

```
else:
    current_line = actr.add_line_to_exp_window(window,[75,135],
                                                 [current_stick + 75,135],"blue")
```

The do_experiment function takes a list of lengths which represent the sticks to present on a trial and an optional parameter indicating whether a person will be performing the task. It calls build_display to create the window and then either waits for a person to finish the task or runs the model for up to 60 seconds to perform the task (in real time if the window is visible for the model). When running the model, since this task requires using the mouse it first uses start_hand_at_mouse to put the model's hand on the virtual mouse instead of the virtual keyboard because the keyboard is the default location for the hands when using the experiment window device.

```
def do_experiment(sticks, human=False):
    build_display(*sticks)

    if human:
        if actr.visible_virtuals_available():
            wait_for_human()
    else:
        actr.install_device(window)
        actr.start_hand_at_mouse()
        actr.run(60,visible)
```

The wait_for_human function loops until the task is over and then waits one second more after displaying the done prompt.

```
def wait_for_human ():
    while not(done):
        actr.process_events()

    start = actr.get_time(False)
    while (actr.get_time(False) - start) < 1000:
        actr.process_events()
```

The bst_set function takes three required parameters which indicate whether a person is doing the task, whether the window should be visible, and a list of stick length lists for the trials to present. It also takes an optional parameter which indicates whether the model should be learning from trial to trial or not. It loops over the trials provided, resetting the model if it isn't supposed to learn from trial to trial, and records the initial strategy choice which was made on each one. It returns the list of strategy choices.

```
def bst_set(human,vis,stims,learn=True):
    global visible

    result = []

    visible = vis

    for stim in stims:
        if not(learn) and not(human):
            actr.reset()
        do_experiment(stim,human)
        result.append(choice)

    return result
```

The test function takes one required parameter which is the number of two-trial test runs to perform and an optional parameter indicating whether a person will be performing the task. It runs the required number of trials and counts the number of times over-shoot is used to solve them returning the list of those counts.

```
def test(n,human=False):

    l = len(no_learn_stims)

    result = [0]*l

    if human or (n == 1):
        v = True
    else:
        v = False

    for i in range(n):

        d = bst_set(human,v,no_learn_stims,False)

        for j in range(l):
            if d[j] == "over":
                result[j] += 1

    return result
```

The experiment function takes one required parameter which is the number of times to perform the whole experiment and an optional parameter indicating whether a person will be performing the task. It runs the required number of experiments and computes the average proportion of times over-shoot is chosen for each trial and the average utility value of the critical productions in the model after each experiment. Those results are printed along with the correlation and mean-deviation from the original experimental data.

```
def experiment(n,human=False):
    l = len(exp_stims)

    result = [0] * l
    p_values = [["decide-over",0],["decide-under",0],["force-over",0],["force-under",0]]
    for i in range(n):
        actr.reset()
        d = bst_set(human,human,exp_stims)
        for j in range(l):
            if d[j] == "over":
                result[j] += 1

        actr.hide_output()

        for p in p_values:
            p[1]+=production_u_value(p[0])

        actr.unhide_output()

    result = list(map(lambda x: 100 * x / n,result))

    if len(result) == len(exp_data):
        actr.correlation(result,exp_data)
        actr.mean_deviation(result,exp_data)

    print()
    print("Trial ",end="")
    for i in range(l):
        print("%-8d"%(i + 1),end="")
    print()

    print("  ",end="")
    for i in range(l):
        print("%8.2f"%result[i],end="")

    print()
    print()

    for p in p_values:
        print("%-12s: %6.4f"%(p[0],p[1]/n))
```

The production_u_value function takes one parameter which should be the name of a production. It returns the current value of the :u parameter for that production.

```
def production_u_value(prod):
    return actr.spp(prod,":u")[0][0]
```

New Commands

Creating Buttons

add-button-to-exp-window and **add_button_to_exp_window –** these functions are similar to the add-text-to-exp-window function that you have seen many times before, but for creating a button object which can be pressed by a person or the model. It has one required parameter which is the window in which to display the button, and several keyword parameters for specifying the text to display on the button, the x and y pixel coordinates of the upper-left corner of the button, an action to perform when the button is pressed, the height and width of the button in pixels, and a color for the background of the button. The action can be a string which names a valid command in ACT-R, a list with the name of a command and the parameters to pass to that command, or nil (Lisp)/None(Python). If no command is provided then pressing the button will result in a warning being printed when it is pressed to indicate the time that the press occurred. It returns an identifier for the button item.

Removing items

remove-items-from-exp-window and **remove_items_from_exp_window –** these functions take one required parameter and any number of additional parameters. The required parameter is an identifier for an experiment window. The remaining values should be the identifiers returned from items that were added to that window. Those items are then removed from that window.

Modifying lines

modify-line-for-exp-window and **modify_line_for_exp_window –** these functions take three required parameters. The first is the identifier for a line item that has been created. The next two are the lists with the x,y coordinates for the end points of the line. It has one optional parameter which should be the name of a color. The line object which is provided is updated so that its end points and color have the new values provided instead of the values they had previously. If a value of nil (Lisp) or None (Python) is provided for an end point then that item is not changed and remains as it was.

Using the mouse

By default, the experiment window device provides a keyboard and mouse for the model and its hands start on the keyboard. If you want the model's right hand to start on the mouse you can use the **start-hand-at-mouse** or **start_hand_at_mouse** function before you run the model. It is also possible for the model to move its hand explicitly to the mouse with a manual request using the cmd value hand-to-mouse and then to move it back to the keyboard with the cmd value hand-to-home.

Hiding ACT-R output

As you have seen throughout the tutorial, many of the ACT-R commands often print out information when they are used. Sometimes that output is not necessary, like in this task where we want to get the value of productions' utilities but don't care about seeing them each time. To avoid that unnecessary output there are some ways to turn it off or hide it, but there are some things to be careful about when doing so.

One option is to set the parameter :cmdt to the value nil. That parameter is similar to the :v parameter but it controls the output of the commands (it's the command trace parameter). The downside of turning that off is that it turns off all of the command output for the model regardless of the source of the command, which isn't really a concern when working through the tasks in the tutorial where you are the only one interacting with the system, but in the more general case that could be an issue if there are multiple tasks/interfaces connected some of which need to see command output and others which do not.

The best option is currently only available for code called from the ACT-R prompt i.e. Lisp. That is the **no-output** macro. It can be wrapped around any number of calls to ACT-R code and will suppress the output of those calls without affecting any commands being evaluated elsewhere e.g. from some other connected client. That is possible because of how macros work in Lisp and because it will be executing all of the ACT-R commands within a single thread thus it is able to make the change without interfering with other commands. That sort of temporary and local disabling is not possible through the remote interface because of how the remote commands are processed with respect to the current output mechanisms, but it is something which is being investigated for improvement in future versions.

When using the Python interface provided with the tutorial there are two pairs of functions that can be used to suppress and restore the output in the Python interface (note however that any other connected client will still get the output). The pair used in this task is **hide_output** and **unhide_output**. Those will suppress and then restore the printing of all the output generated by ACT-R in the Python interface – regardless of where that output is generated i.e. they will also disable warning messages which may be generated from elsewhere, for example those generated by a module while the model is running. These functions don't stop the Python interface from monitoring for output, just whether or not it is printed when it is received, and are recommended when one just wants to disable the output briefly and then enable it again. The other pair of functions are **stop_output** and **resume_output**. Those command stop monitoring for ACT-R output entirely and then restore the monitoring of output respectively. Those are costly operations and thus not the sort of thing one would want to do repeatedly, but could be useful when running a long task for which no output is needed where they will only be called once at the start and end of that task.

Providing rewards

Although it wasn't used in the task for this unit the main text mentioned the trigger-reward command. If one wants to provide a reward to a model for utility learning purposes independently of the productions then the **trigger-reward** (Lisp) or **trigger_reward** (Python) function can be used to do so. It takes one required parameter which is the amount of reward to provide and that reward will be given to the model at the current time. The reward can be a number, in which case the utilities are updated as described in the main unit text, or any other value which is not "null" (nil in Lisp or False/None in Python). If a non-numeric reward is provided that value is displayed in the model trace and serves to mark the last point of reward, but does not cause any updating of the utilities – it only serves to set the stopping point for how far back the next numeric reward will be applied.

Using values returned from calling ACT-R commands in productions

In the previous unit the !eval! production operator was described to show how one can call commands from within productions. It is also possible to use the value returned from such a call within the production. To do that the !bind! operator is used. It works very similarly to the !eval! operator, but it also requires specifying a variable name in which to store the result of evaluating the command. The encode-over and encode-under productions in the Building Sticks model use a !bind! to compute the difference between stick lengths and store the result in a slot. Here is the encode-under production:

```
(p encode-under
   =goal>
      isa        try-strategy
      state      encode-under
   =imaginal>
      isa        encoding
      b-loc      =b
      length     =goal-len
   =visual>
      isa        line
      width      =c-len
   ?visual>
      state      free
  ==>
   !bind! =val   (- =goal-len =c-len)

   =imaginal>
      under      =val
   =goal>
      state      encode-over
   +visual>
      isa        move-attention
      screen-pos =b)
```

That line of the production is using a Lisp expression to be evaluated, but like !eval! it can also evaluate an ACT-R command given the string of its name. Here is an example which would store the result of evaluating "test-command" passed the value of the =number variable into the =result variable:

```
   !bind! =result ("test-command" =number)
```

On the LHS of a production a !bind! specifies a condition that must be met before the production can be selected just like all the other items on the LHS. The value returned by the evaluation of a LHS !bind! must be true for the production to be selected (where true is technically anything that is not nil in Lisp and if the command is implemented in some other language the values must not be the equivalent of nil e.g. False and None in Python are equivalent to nil in Lisp).

On the RHS of a production if the evaluated expression from a !bind! returns a nil result it will generate a warning from ACT-R and a value of t will be used instead since a variable in a production cannot be bound to nil since that indicates the absence of a value.

Like !eval!, !bind! can be a powerful tool which can easily be abused. It is not required, and should not be used, when writing any of the productions for the assignments in the tutorial (although in unit 7 some of the starting productions for the assignment do use a !bind!).

Unit 7: Production Rule Learning

In this unit we will discuss how new production rules are learned. As we will see, a model can acquire new production rules by collapsing two production rules that apply in succession into a single rule. Through this process the model will transform knowledge that is stored declaratively into a procedural form. We call this process of forming new production rules **production compilation** and refer to the specific act of combining two productions as a composition.

7.1 The Basic Idea

A good pair of productions for illustrating production compilation is the two that fire in succession to retrieve a paired associate in the **paired** model from Unit 4:

```
(p read-probe
    =goal>
      isa      goal
      state    attending-probe
    =visual>
      isa      visual-object
      value    =val
    ?imaginal>
      state    free
   ==>
    +imaginal>
      isa      pair
      probe    =val
    +retrieval>
      isa      pair
      probe    =val
    =goal>
      state    testing)
```

```
(p recall
    =goal>
      isa      goal
      state    testing
    =retrieval>
      isa      pair
      answer   =ans
    ?manual>
      state    free
    ?visual>
      state    free
   ==>
    +manual>
      cmd      press-key
      key      =ans
    =goal>
      state    read-study-item
    +visual>
      cmd      clear)
```

If these two productions fired and retrieved the chunk for the pair of zinc & 9, production compilation would compose these two rules into the following single production:

```
(P PRODUCTION0
  "READ-PROBE & RECALL - CHUNK0-0"
   =GOAL>
```

```
        STATE ATTENDING-PROBE
    =VISUAL>
        VALUE "zinc"
    ?IMAGINAL>
        STATE FREE
    ?MANUAL>
        STATE FREE
    ?VISUAL>
        STATE FREE
  ==>
    =GOAL>
        STATE READ-STUDY-ITEM
    +VISUAL>
        CMD CLEAR
    +MANUAL>
        CMD PRESS-KEY
        KEY "9"
    +IMAGINAL>
        PROBE "zinc"
)
```

This production combines the work of the two productions and has built into it the paired associate components. In the next two subsections we will describe the general principles used for composing two production rules together and the factors that control how these productions compete in the conflict resolution process.

7.2 Forming a New Production

The basic idea behind forming a new production is to combine the tests in the two conditions into a single set of tests that will recognize when the pair of productions will apply and combine the two sets of actions into a single set of actions that has the same overall effect. Since the conditions consist of a set of buffer tests and the actions consist of a set of buffer transformations (either direct modifications or new requests) this can be done largely on a buffer-by-buffer basis. The complications occur when there is a buffer transformation in the action of the first production and either a test of that buffer in the condition of the second production or another transformation of the same buffer in the action of the second production. The productions above illustrate both complications with respect to the **goal** buffer. Because the change to the state slot of the **goal** buffer chunk in the first production is tested as a condition in the second production that test can be omitted from the tests of the composed production. Then, because that state slot is changed again by the second production, and the composed production only needs to reproduce the final state of the original two productions, that first goal change can also be omitted from the actions of the compiled production. The result of the overlap in the **goal** buffer is just a simplification of the production rule but in other cases other responses are necessary.

Because different modules use their buffers in different ways the production compilation process needs to be sensitive to those differences. For instance, in the above production we see that the **retrieval** buffer request was omitted from the newly formed production, but the **imaginal** buffer request was not. The production compilation mechanism is built around sets of rules that we call styles and each buffer is handled using the rules for the style which best categorizes its use. For each style the rules specify when two productions that use that same buffer can be combined through compilation and how to compose the uses of that buffer. By default there are five styles to which the buffers of ACT-R are assigned and we will describe the mechanisms used for those styles in the following sections. It is possible to add new styles and also to adjust the assignment of buffers to styles, but that is beyond the scope of the tutorial.

7.2.1 Motor Style Buffers

Let us first consider the compilation policy for the motor style buffers. The buffers that fit this style are the **manual** and **vocal** buffers. The main distinction of these buffers is that they never hold a chunk. They are used for requesting actions of a module which can only process one request at a time and they are only tested through queries. If the first production makes a request of one of these buffers then it is not possible to compose it with a second production if that production also makes a request of that buffer or queries the buffer for anything other than state busy. If both productions make a request, then there is a danger of jamming if both requests were to occur at the same time, and any queries that are not checking to see if the module is busy in the second production are probably there to prevent jamming in the future, and thus also block the composition.

7.2.2 Perceptual Style Buffers

Now let us consider the compilation policy for the perceptual buffers. These are the buffers in the perceptual style: **visual-location**, **visual**, **aural-location**, and **aural.** These buffers will hold chunks generated by their modules. The important characteristic about them is that those chunks are based on information in the external world, and thus are not guaranteed to result in the same request generating the same result at another time. First, like the motor style buffers, it is not possible to compose two productions which both make requests of the same perceptual style buffer or if the first production makes a request and the second production makes a query for other than state busy of that buffer because of the possibility of jamming. In addition, if the first production makes a request of one of these buffers then it is not possible to compose it with the second production if that production tests the contents of the buffer. This is because of the unpredictable nature of such requests – one does not want to create productions that encapsulate information that is based on external information which may not be valid ever again (at least not for most modeling purposes). The idea is that we only want to create new productions that are “safe”, and by safe we mean that the new production can only match if the productions that it was generated from would match and that its actions are the same as those of its parent productions. Basically, for the default mechanism, we do not want composed productions to be generated that introduce new errors into the model.

Thus, points where a request is made of a perceptual or motor style buffer are points where there are natural breaks in the compilation process. The standard production compilation mechanisms will not compose a production that makes such a request with a following production that operates on the same buffer.

7.2.3 Retrieval Style Buffer

Next let us consider the compilation policy for the **retrieval** buffer (the only buffer of the retrieval style). Because declarative memory is an internal mechanism (i.e., not subject to the whims of the outside world) it is more predictable and thus offers an opportunity for economy. The interesting opportunity for economy occurs when the first production requests a retrieval and the second tests the result of that retrieval. In this case, one can delete the request and test and instead specialize the composed production. Any variables specified in the retrieval request and bound in the harvesting of the chunk can be replaced with the specific values based on the chunk that was actually retrieved. This was what happened in the example production above where the retrieved paired-associate for zinc & 9 was built into the new production0. There is one case, however, which blocks the composition of a production which makes a retrieval request with a subsequent production. That is when the second production has a query for a retrieval failure. This cannot be composed because declarative memory grows monotonically and it is not safe to predict that in the future there will be a retrieval failure. This suggests that it is preferable, if possible, to write production rules that do not depend on retrieval failures.

7.2.4 Goal and Imaginal Style Buffers

The **goal** and **imaginal** buffers are also internal buffers allowing economies to be achieved. The mechanisms used for these two buffers are very similar, and thus will be described together. The difference between them arises from the fact that the **imaginal** buffer requests take time to complete, and that difference will be described below. First, we will analyze the process for which they are the same and that is broken down into cases based on whether the first production involves a request to the buffer or not.

7.2.4.a First production does not make a request

Let C1 and C2 be the conditions for the buffer in the first and second production and A1 and A2 be the corresponding productions' buffer modification actions for that buffer. Then, the buffer test for the composed production is C1+(C2-A1) where (C2-A1) specifies those things tested in C2 that were not created in A1. The modification for the combined production is A2+(A1~A2) where (A1~A2) indicates the modifications that were in A1 that are not undone by A2. If the second production makes a request, then that request can just be included in the composed production.

7.2.4.b First production makes a request

This case breaks down into two subcases depending on whether the second production also makes a request.

The second production does not also make a request. In this case the second production's buffer test can be deleted since its satisfaction is guaranteed by the first production. Let C1 be the buffer condition of the first production, A1 be the buffer modification action of the first production, N1 be the new request in the first production, and A2 be the buffer modification of the second production. Then the buffer test of the composed production is just C1, the goal modification is just A1, and the new request is A2+(N1~A2).

The second production also makes a request. In this case the two productions cannot be composed because this would require either skipping over the intermediate request, which would result in a chunk not being created in the new production which was generated by the initial two productions, or making two requests to the buffer in one production which could lead to jamming the module.

7.2.4.c Difference between goal and imaginal

The difference between the two comes down to the use of queries. Because the imaginal module's requests take time one typically needs to make queries to test whether it is free or busy whereas the goal module's requests complete immediately and thus the state is never busy. So, for goal style buffers production compilation is blocked by any queries in either production because that represents an unusual situation and is thus deemed unsafe. For imaginal style buffers productions which have queries are allowed when the queries and actions between the productions are "consistent". All of the rules for composition consistency with the imaginal style buffers are a little too complex to describe here, but generally speaking if the first production has a request then the second must either test the buffer for state busy and not also make a request (like motor style buffers) or explicitly test that the buffer is free and make only modifications to the buffer.

For more details, there is a spreadsheet in the docs directory called compilation.xls which contains a matrix for each buffer style indicating what combinations of usages of a buffer between the two productions can be composed.

7.3 Utility of newly created productions

So far we have discussed how new production rules are created but not how they are used. When a new production is composed it enters the procedural module and is treated just like the productions which are specified in the model definition. They are matched against the current state along with all the rest of the productions and among all the productions which match the current state the one with the highest utility is selected and fired. When a new production, which we will call **New**, is composed from old productions, which we will call **Old1** and **Old2** and which fired in that order, it is the case that whenever **New** could apply **Old1** could also apply. (Note however because **New** might be specialized it does not follow that whenever **Old1** could apply **New** could also apply.) The choice between **New**, **Old1**, and any other productions which might also apply will be determined by their utilities as was discussed in the previous unit.

A newly learned production **New** will initially receive a utility of zero by default (that can be changed with the :nu parameter). Assuming **Old1** has a positive utility value, this means that **New** will almost always lose in conflict resolution with **Old1**. However, each time **New** is recreated from **Old1** and **Old2**, its utility is updated with a reward equal to the current utility of **Old1**, using the same learning equation as discussed in the previous unit:

$$U_i(n) = U_i(n-1) + \alpha\left[R_i(n) - U_i(n-1)\right]$$ **Difference Learning Equation**

As a consequence, even though **New** may not fire initially, its utility will gradually approach the utility of **Old1**. Once the utility of **New** and **Old1** are close enough, **New** will occasionally be selected because of the noise in utilities. Once **New** is selected it will receive a reward like any other production which fires, and its utility can surpass **Old1**'s utility if it is better (it is usually a little better because it typically leads to rewards faster since it saves a production rule firing and often a retrieval from declarative memory).

7.4 Learning from Instruction

Generally, production compilation allows a problem to be solved with fewer productions over time and therefore performed faster. In addition to this speed-up, production compilation results in the drop-out of declarative retrieval as part of the task performance. As we saw in the example in the first section, production rules are produced that just "do it" and do not bother retrieving the intervening information. The classic case of where this applies in experimental psychology is in the learning of experimental instructions. These instructions are told to the participant and initially the participant needs to interpret these declarative instructions. However, with practice the participant will come to embed these instructions into productions that directly perform the task. These productions will be like the productions we normally write to model participant performance in the task. Essentially these are productions that participants learn in the warm-up phase of the experiment. The **paired-learning** model for this assignment contains an example of a system that interprets instructions about how to perform a paired associate task and learns the productions that do the task directly.

In the model we use the following chunks to represent the understanding of the instructions for the paired associate task (in some of our work we have built productions that read the instructions from the screen and build these chunks but we are skipping that step here to focus on the mechanisms of this unit):

1. (op1 isa operator pre start action read arg1 create post stimulus-read)
2. (op2 isa operator pre stimulus-read action associate arg1 filled arg2 fill post recalled)
3. (op3 isa operator pre recalled action test-arg2 arg1 respond arg2 wait)
4. (op4 isa operator pre respond action type arg2 response post wait)
5. (op5 isa operator pre wait action read arg2 fill post new-trial)
6. (op6 isa operator pre new-trial action complete-task post start)

These are represented as operators that indicate what to do in various states during the course of a paired-associate trial. They consist of a statement of what that state is in the **pre** slot and what state will occur after the action in the **post** slot. In addition, there is an **action** slot to specify the action to perform and two slots, **arg1** and **arg2**, for holding possible arguments needed during the task execution. So to loosely translate the six operators above:

1. At the start read the word and create an encoding of it as the stimulus
2. After reading the stimulus try to retrieve an associate to the stimulus
3. Test whether an item has been recalled and if it has not then just wait
4. If an item has been recalled type it and then wait
5. Store the response you read with the stimulus
6. This trial is complete so start the next one

The model uses a chunk in the **goal** buffer to maintain a current state and sub step within that state and a chunk in the **imaginal** buffer to hold the items relevant to the current state. For this task, the arguments are the stimulus and probe for a trial.

The model must retrieve operators from declarative memory which apply to the current state to determine what to do, and in this simple model we really just need one production which requests the retrieval of an operator relevant to the current state:

```
(p retrieve-operator
    =goal>
```

```
      isa   task
      state =state
      step  ready
   ==>
    +retrieval>
      isa   operator
      pre   =state
    =goal>
      step  retrieving-operator)
```

The particular actions specified in the operators (read, associate, test-arg2, type, and complete-task) are all general actions not specific to a paired associate task. We assume that the participant knows how to do these things going into the experiment. This amounts to assuming that there are productions for processing these actions. For instance, the following two productions are responsible for reading an item and creating a chunk in the **imaginal** buffer which encodes the item into the **arg1** slot of that chunk:

```
(p read-arg1
   =goal>
     isa        task
     step       retrieving-operator
   =retrieval>
     isa        operator
     action     read
     arg1       create
     post       =state
   =visual-location>
   ?visual>
     state      free
   ?imaginal>
     state      free
  ==>
   +imaginal>
     isa        args
     arg1       fill
   +visual>
     cmd        move-attention
     screen-pos =visual-location
   =goal>
     step       attending
     state      =state)


(p encode-arg1
   =goal>
     isa     task
     step    attending
   =visual>
     isa     text
     value   =val
   =imaginal>
     isa     args
     arg1    fill
   ?imaginal>
     state   free
  ==>
   *imaginal>
     arg1    =val
   =goal>
     step    ready)
```

The first production responds to the retrieval of the operator and requests a visual attention shift to an item. It also changes the state slot in the **goal** buffer to the operator's post state. The second production modifies the representation in the **imaginal** buffer with the value from the chunk in the **visual** buffer and sets the **goal** buffer's step slot to indicate that it is ready to retrieve the operator relevant to the next state.

The **paired-learning** model performs the same task as the **paired** model you used for Unit 4. However, rather than having specific productions for doing the task it interprets these operators that represent the instructions for doing this task. For reference, here is the data that is being modeled again:

Trial	Accuracy	Latency
1	.000	0.000
2	.526	2.156
3	.667	1.967
4	.798	1.762
5	.887	1.680
6	.924	1.552
7	.958	1.467
8	.954	1.402

To run the model you should first load/import the paired experiment code and then load the paired-learning.lisp model from this unit since the paired experiment code loads the unit 4 model by default. The model can be run either with production compilation on or off. To turn production compilation off, set the **:epl** parameter to **nil** in the sgp setting at the top of the model. The following shows the data from running the model through the experiment without production compilation:

```
Latency:
CORRELATION:  0.974
MEAN DEVIATION:  0.194
Trial    1       2       3       4       5       6       7       8
        0.000   2.049   1.980   1.885   1.854   1.799   1.755   1.719

Accuracy:
CORRELATION:  0.994
MEAN DEVIATION:  0.041
Trial    1       2       3       4       5       6       7       8
        0.000   0.418   0.650   0.785   0.882   0.918   0.935   0.975
```

When it is run with production compilation enabled using a value of **t** for **:epl (**which is how it is set in the given model file) the results look like this:

```
Latency:
CORRELATION:  0.992
MEAN DEVIATION:  0.083
Trial    1       2       3       4       5       6       7       8
        0.000   2.046   1.958   1.792   1.676   1.628   1.598   1.538

Accuracy:
CORRELATION:  0.995
MEAN DEVIATION:  0.036
Trial    1       2       3       4       5       6       7       8
        0.000   0.428   0.660   0.790   0.885   0.920   0.948   0.972
```

As can be seen, whether production compilation is off or on has relatively little effect on the accuracy of recall but turning it on greatly increases the speed-up over trials in the recall time. This is because we are eliminating productions and retrievals as new rules are composed.

If you set the **:pct** (production compilation trace) parameter to **t** (and you will also need to set **:v** to **t**) you will see the system print out the new productions as they are composed or the reason why two productions could not be composed. For instance, the following is a fragment of the trace when we run one paired-associate for 1 trial with production compilation turned on.

```
     0.400   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-OPERATOR
Production Compilation process started for RETRIEVE-OPERATOR
  Production ENCODE-ARG1 and RETRIEVE-OPERATOR are being composed.
  New production:

(P PRODUCTION1
  "ENCODE-ARG1 & RETRIEVE-OPERATOR"
   =GOAL>
       STEP ATTENDING
       STATE =STATE
   =IMAGINAL>
       ARG1 FILL
   =VISUAL>
       TEXT T
       VALUE =VAL
   ?IMAGINAL>
       STATE FREE
 ==>
   =GOAL>
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE =STATE
   *IMAGINAL>
       ARG1 =VAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

The production that is learned, **production1**, is a composition of these two productions:

```
(p encode-arg1
   =goal>
     isa     task
     step    attending
   =visual>
     isa     text
     value   =val
   =imaginal>
     isa     args
     arg1    fill
   ?imaginal>
     state   free
  ==>
   *imaginal>
     arg1    =val
   =goal>
     step    ready)

(p retrieve-operator
   =goal>
     isa    task
     state  =state
     step   ready
  ==>
   +retrieval>
     isa    operator
     pre    =state
   =goal>
     step   retrieving-operator)
```

The first production, **encode-arg1**, encodes the stimulus, and sets the goal to retrieve the next operator. This is followed by **retrieve-operator**, which makes the retrieval request. The composed production is particularly straight forward. Its condition is just the condition of the first production plus the check for the state slot in the goal of the second production, and its action combines the actions of the two.

It is worth understanding how the parameters of this new production are calculated. The value of the utility learning rate, α, in the model is 0.2 (set with the :alpha parameter) which is also the default value for :alpha. When the production is first created, its utility is set to zero (the default for new productions). The utilities of the initial rules in the system are all set to 5 (using the :iu parameter which sets the starting utility for all of the initial productions), so a value of zero means that it will almost certainly lose the competition with the parent production the next time it might be applicable. However, each time it is recreated, it receives a reward which is the same as the utility of the first parent production. Suppose that when the parents fire in sequence again and this same production is recreated, the first parent still has a utility of 5. The utility of the new rule is then updated:

$$\begin{aligned}U_i(n) &= U_i(n-1) + \alpha[R_i(n) - U_i(n-1)]\\ &= 0 + .2(5-0)\\ &= 1\end{aligned}$$

A utility of 1 is still not sufficient to be selected in competition with a production of utility 5 given the noise setting of .4 in this model. Thus it will need to be recreated a number of times before it will have a significant chance of being chosen in conflict resolution. The speed of this learning is determined by the setting of α. If it is set to 1, productions will typically get very good values immediately and likely be tried on the first opportunity. If you do that and run several trials of a single item you will discover in just a few trials it is selecting and firing newly learned productions which are then also going through production compilation:

```
    25.486   PROCEDURAL             PRODUCTION-FIRED PRODUCTION8
Production Compilation process started for PRODUCTION8
  Production RETRIEVE-OPERATOR and PRODUCTION8 are being composed.
  New production:

(P PRODUCTION29
  "RETRIEVE-OPERATOR & PRODUCTION8 - OP6"
   =GOAL>
       STATE NEW-TRIAL
       STEP READY
 ==>
```

```
   =GOAL>
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE START
   +GOAL>
       STATE START
       STEP RETRIEVING-OPERATOR
)
```

After just a couple more trials we will start to find productions that are the composition of multiple previously composed productions and it will soon end up with a production like this:

```
(P PRODUCTION81
  "PRODUCTION58 & PRODUCTION49 - OP3"
   =GOAL>
       STATE STIMULUS-READ
       STEP ATTENDING
   =IMAGINAL>
       ARG1 FILL
   =VISUAL>
       TEXT T
       VALUE "zinc"
   ?IMAGINAL>
       STATE FREE
   ?MANUAL>
       STATE FREE
 ==>
   =GOAL>
       STATE WAIT
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE WAIT
   +MANUAL>
       CMD PRESS-KEY
       KEY "9"
   *IMAGINAL>
       ARG1 "zinc"
       ARG2 "9"
)
```

Which responds to seeing “zinc” on the screen by placing both “zinc” and “9” in the **imaginal** buffer chunk’s slots and pressing the “9” key.

7.5 Assignment

Your assignment is to make a model that learns the past tense of verbs in English. The learning process of the English past tense is characterized by the so-called U-shaped learning with irregular verbs. That is, at a certain age children inflect irregular verbs like “to break” correctly, so they say “broke” if they want to use the past tense. But at a later age, they overgeneralize, and start saying “breaked”, and then at an even later stage they again inflect irregular verbs correctly. Some people, such as Pinker and Marcus, interpret this as evidence that a rule is learned to create a regular past tense (add “ed” to the stem). According to Pinker and Marcus, after this rule has been learned, it is overgeneralized so that it will also produce regularized versions of irregular verbs.

The start of a model to learn the past tense of verbs is included with the unit in the past-tense-model.lisp file. The assignment is to make the model learn a production which represents the regular rule for making the past tense and also specific productions for producing the past tense of each verb. So eventually it should learn productions which act essentially like this:

IF the goal is to make the past tense of a verb

THEN copy that verb and add –ed

IF the goal is to make the past tense of the verb have

THEN the past tense is had

The code that is provided in the past-tense.lisp and past_tense.py files for performing this task does two things. It adds correct past tenses to declarative memory, reflecting the fact that a child hears and then encodes correct past tenses from others. It also creates goals which indicate to the model that it should generate the past tense of a verb found in the **imaginal** buffer and then runs the model to do so. The model will be given two correct past tenses for every one that it must generate.

The past tense of verbs are encoded in chunks using the slots of this chunk-type:

```
(chunk-type past-tense verb stem suffix)
```

Where the verb slot holds the base form of the verb and the combination of the stem and suffix slots forms the past tense, with the value blank in the suffix slot meaning that there is no suffix to add (which is used instead of not specifying the slot so that one can distinguish a complete past tense from one that is malformed). Here are examples of correctly formed past tenses for the irregular verb have and the regular verb use:

```
PAST-TENSE1
     verb have
     stem had
     suffix blank

PAST-TENSE234
     verb use
     stem use
     suffix ed
```

To indicate to the model that it should create a past tense the chunk in the **goal** buffer will have a state slot with a value of start and the chunk in the **imaginal** buffer will contain a chunk which has a verb slot with the base form of a verb. Here is what those buffers' contents would look like at the start of a run to form the past tense of the verb work:

```
GOAL: STARTING-GOAL-0 [STARTING-GOAL]
STARTING-GOAL-0
   STATE  START

IMAGINAL: CHUNK2-0 [CHUNK2]
CHUNK2-0
   VERB  WORK
```

The model then has to fill in the stem and suffix slots of the chunk in the **imaginal** buffer to indicate the past tense form of the verb and set the state slot of the chunk in the **goal** buffer to done to indicate that it is finished. Once the state slot is set to done, one of the three productions provided with the model should fire to simulate the final encoding and "use" of the word, each of which has a different reward. There are three possible cases:

- An irregular inflection, this is when there is a value in the stem slot and the suffix is marked explicitly as blank. This use has the highest reward, because irregular verbs tend to be short.
- A regular inflection, in which the stem slot is the same as the verb slot and the suffix slot has a value which is not blank. This has a slightly lower reward.
- Non-inflected when neither the stem nor suffix slots are set in the chunk. The non-inflection case applies when the model cannot come up with a past tense at all, either because it has no example to retrieve, no production to create it, or no strategy to come up with anything based on a retrieved past tense. The non-inflection situation receives the lowest reward because the past tense would have to be indicated by some other method, for example by adding "yesterday" or some other explicit reference to time.

An important thing to notice is that all three of those situations receive a reward. The model receives no feedback as to whether the past tenses it produces are correct – any validly formed result is considered a success and rewarded. The only feedback it receives with respect to the correct form of past tenses is the correctly constructed verbs that it hears between the attempts to generate its own.

You can run the model with the **past-tense-trials** function in Lisp or the **trials** function in the past_tense module for Python. It takes as an argument the number of past tenses you want the model to generate:

```
? (past-tense-trials 5000)

>>> past_tense.trials(5000)
```

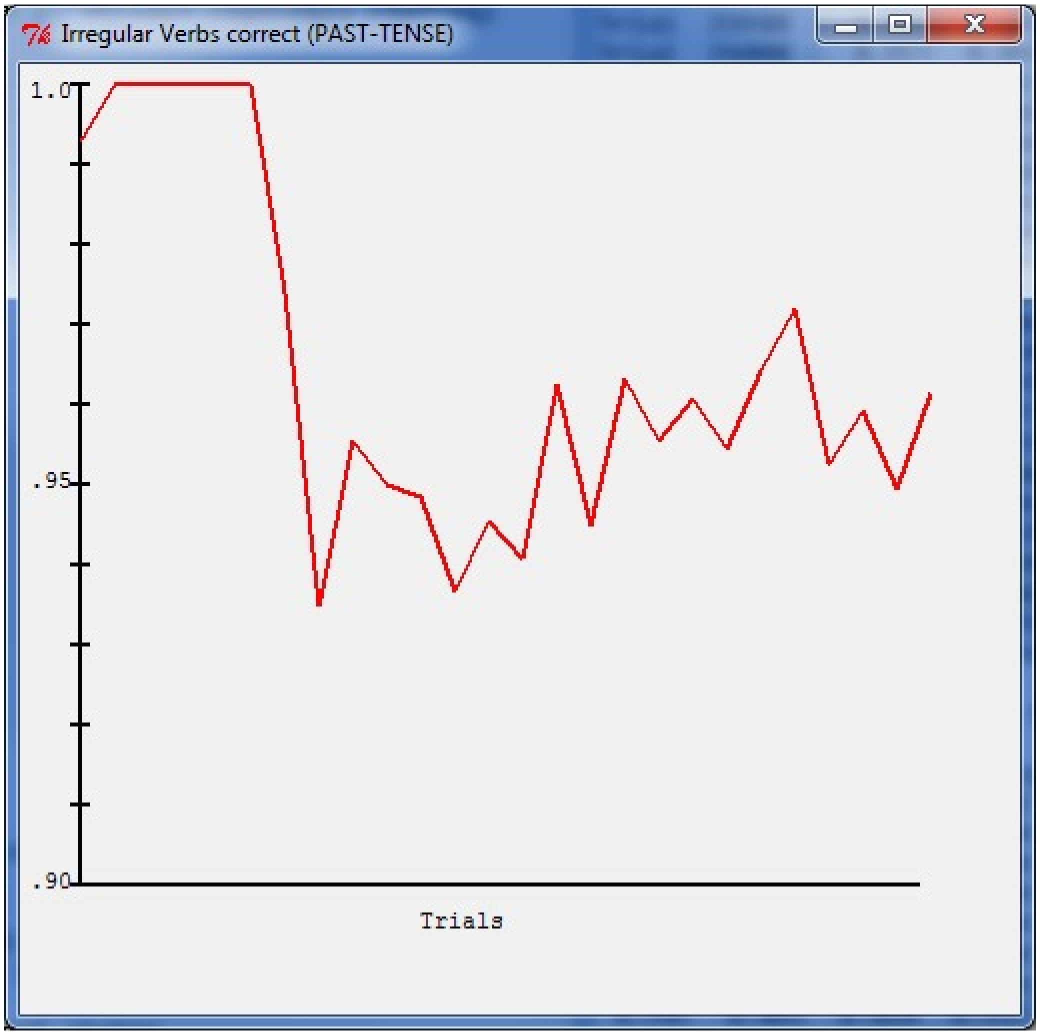
As optional parameters you can specify whether or not you want the model to continue from where it left off or to start again from the beginning, and also whether the ACT-R trace should be shown while it runs. The defaults are to start from the beginning and not show the trace. To change that for each of the options you need to specify a true value. Therefore these calls would run 100 more trials continuing from where it left off and displaying the trace:

```
? (past-tense-trials 100 t t)

>>> past_tense.trials(100,True,True)
```

As the model runs, the simulation will print out lines containing four numbers which represent the results of the last 100 verbs generated by the model. The first number is the proportion correct of irregular verbs. The second number is the proportion of irregular verbs that are inflected regularly. An increase in this number suggests that a regular rule is active i.e. irregular verbs are having “ed” added to them. The third number is the proportion of irregular verbs that are not inflected at all. The fourth number is the proportion of inflected irregular verbs that are inflected correctly (the non-inflected verbs are not counted for this measure). It is in this last column that you should see a U-shape.

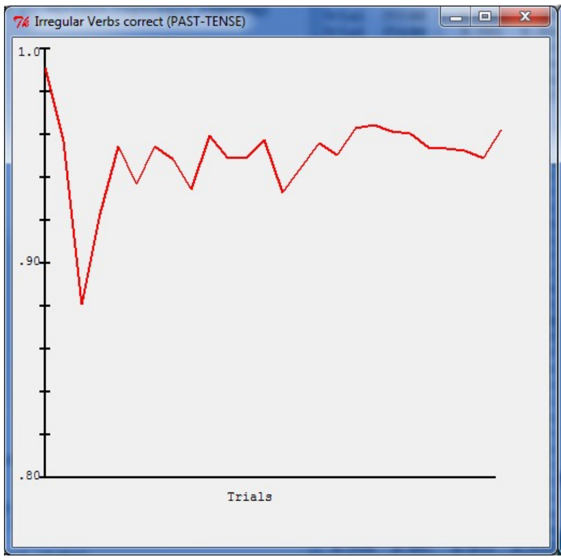
It usually requires more than 5000 trials to see the effect and often 15000~20000 trials are necessary before the entire U-shape in the learning forms. After the model has been run for at least 1000 trials, the **past-tense-results** function in Lisp or the **results** function in the past_tense module for Python will print out the results aggregated over 1000 trials at a time (the 100 trial summaries displayed while the model runs are usually too variable to easily discern the U-shape but they do allow you to see that the model is still running and roughly in what direction the results are going). By default the result of the correctly inflected irregular verbs will also be graphed to make it easier to see the U-shape, but that can be suppressed by providing a null value for the optional parameter (nil/None) to the reporting function if you just want to see the numbers. What you are looking for from the model is a graph that looks something like this:

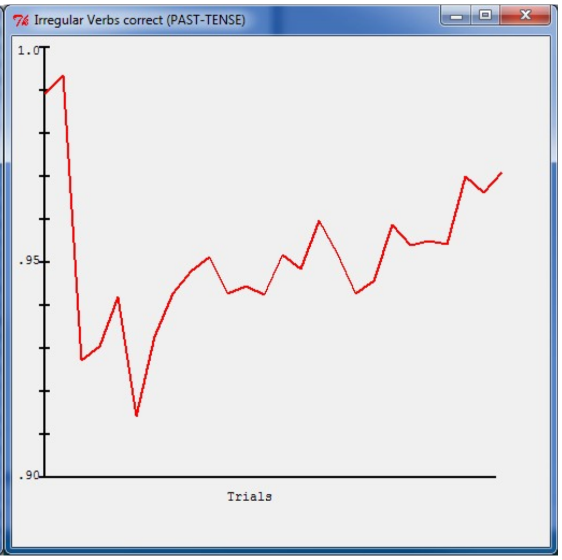


It starts out with a high percentage correct (nearly 100%), dips down, and then shows an increasing trend as it continues (this simplified version of the task will probably not get all the way back to 100% correct). That is the U-shaped learning result. An important thing to look at with the output in the graph is the vertical scale. The graph will adjust the y-intercept to display all of the results and the model should always be getting most of the trials where it does inflect the verb correct (if it's ever dropping below .75 there's probably a problem with the model).

This model differs from other models in the tutorial in that it does not model a particular experiment, but rather some long term development. This has a couple of consequences for the model. One of those is that using the perceptual and motor modules does not contribute much to the objective of the model. Thus things like actually hearing the past tenses and the generation of the word as speech are not modeled for the purpose of this exercise. They could be modeled, but it is not what the model and exercise are about and would really only serve to make things more complicated with little to no benefit. Another consequence of the nature of this task is a practical one with how long it takes to run the model to see those long term results. Because of that, we are not averaging multiple runs of the model together to report the results because the learning trajectory can vary significantly between trials and it would take a very large number of runs to reliably see the U-shape in the average results which would take a much longer time to run. Also contributing to that variability is that we are only using a very small vocabulary (though still maintaining the appropriate frequency of irregular and regular verb usage) again to keep the time needed to run it reasonable for an exercise, which leads to less data for averaging.

For reference, here are some more images of the results for the same model as shown above:





In fact, even a "correct" model may sometimes fail to show the U-shape, for example never dipping down because it learns the right inflections before it learns the regular rule or never really coming back up after dipping down because the regular rule is reinforce too much. However, on most runs (90% or more) a good model should show the U-shape in some form.

Having that sort of variability between runs is not entirely bad, as children also differ quite a bit with respect to U-shaped learning. However that does make comparing this model to data difficult, and hard data on the phenomenon are also scarce, although the phenomenon of the U-shape is reported often. For reference, data from a few children that have been followed in a longitudinal study on the topic is included with the unit in a spreadsheet (data.xls) that shows some results for comparison.

In terms of the assignment, the objective is to write a model that learns the appropriate productions for producing past tenses. There is no parameter adjustment or data fitting required. All one needs to do is write productions which can generate past tenses based on retrieving previous past tenses, and which through production compilation will over time result in new productions which directly apply the regular rule or produce the specific past tense.

The key to a successful model is to implement two different retrieval strategies in the model. The model can either try to remember the past tense for the specific verb or the model can try to generate a past tense based on retrieving any past-tense. These should be competing strategies, and only one applied on any given attempt. If the chosen strategy fails to produce a result the model should "give up" and not inflect the verb. The reason for doing that is that language generation is a rapid process and not something for which a lot of time per word can be allocated.

Additionally, the productions you write should have **no explicit reference to either of the suffixes, ed or blank**, because that is what the model is to eventually learn, i.e., you do not write a production that says add ed, but through the production compilation mechanism such a production is learned. Although the experiment code is only outfitted with a limited set of words, the frequency with which the words are presented to the model is in accordance with the frequency they appear in real life, and because of that, if your model learns appropriate productions it should generate the U-shaped learning automatically (although it won't always look the same on every run as seen above). Unlike the other models in the tutorial, for this task there is a fairly small set of "good" solutions which will result in the generation of the U-shaped learning because it is actually the model's starting productions which get composed to result in the learning of specific productions over time, and those starting productions need to allow the production compositions to happen which restricts the types of things that they can do.

There are two important things to make sure to address when writing your model with respect to performing the task. It must set the state slot of the chunk in the **goal** buffer to done on each trial, and the chunk in the **imaginal** buffer must have one of the forms described above: either a chunk with values in each of the verb, stem and suffix slots or a chunk with only a value in the verb slot. Those conditions are necessary so that one of the provided productions will fire and propagate a reward. If it does not do so, then there will be no reward propagated to promote the utility learning for that trial and the reward from a later trial will be propagated back to the productions which fired on the trial which didn't get a reward. That later reward will be very negative because there are 200 seconds between trials, and that will make it very difficult, if not impossible, to produce the U-shaped learning. Essentially this represents the model saying something every time it tries to produce a past-tense while speaking.

One final thing to note is that when the model doesn't give up it should produce a reasonable answer. One aspect of being reasonable is that the resulting chunk in the **imaginal** buffer should have the same value in the verb slot that it started with. Similarly, if there is a value in the stem slot, then it should be either that verb or the correct irregular form of that verb. For example, if it starts with "have" in the verb slot acceptable values would be "have" or "had" in the stem slot, but if it starts with a regular verb, like "use" the only acceptable value for the stem would be "use" because it shouldn't be trying to create some irregular form for a regular verb. If the model does produce an unreasonable result there will be a warning printed showing the starting verb and the result which the model created like this:

```
#|Warning: Incorrectly formed verb.  Presented CALL and produced VERB GET STEM
GOT SUFFIX BLANK. |#
```

That indicates the model was asked to produce the past tense for “call” and responded with the correct past tense for the different verb “get”. There will also be a warning printed if the model does not receive a reward on any trial. Your model should not produce any warnings when it runs.

References

Marcus, G. F., Pinker, S., Ullman, M., Hollander, M., Rosen, T. J., & Xu, F. (1992). Overregularization in Language Acquisition. *Monographs of the Society for Research in Child Development,57*(4).

Unit 7 Code Description

The paired associate model for this unit uses the same experiment code as the paired associate model from unit 4. So in this document we will only be looking at the past tense model. This is another example of an experiment and model that do not use the perceptual and motor components of ACT-R.

Lisp

Start by loading the starting model for this task.

```
(load-act-r-model "ACT-R:tutorial;unit7;past-tense-model.lisp")
```

Define some global variables to hold the collected data and generate the verbs with the appropriate frequencies.

```
(defvar *report*)
(defvar *total-count*)
(defvar *word-list*)

(defparameter *verbs* '((have    I    12458 had)
                        (do      I     4367 did)
                        (make    I     2312 made)
                        (get     I     1486 got)
                        (use     R     1016 use)
                        (look    R      910 look)
                        (seem    R      831 seem)
                        (tell    I      759 told)
                        (show    R      640 show)
                        (want    R      631 want)
                        (call    R      627 call)
                        (ask     R      612 ask)
                        (turn    R      566 turn)
                        (follow  R      540 follow)
                        (work    R      496 work)
                        (live    R      472 live)
                        (try     R      472 try)
                        (stand   I      468 stood)
                        (move    R      447 move)
                        (need    R      413 need)
                        (start   R      386 start)
                        (lose    I      274 lost)))
```

Define a function to build the word list for use in the experiment. The list for each verb will include a count for generating a random verb based on its frequency in English, the verb, and its correct past tense.

```
(defun make-word-freq-list (l)
  (let ((data nil)
        (count 0))
    (dolist (verb l)
      (incf count (third verb))
      (push-last (list count (first verb) (fourth verb)
                       (if (eq (second verb) 'i) 'blank 'ed))
            data))
    (setf *total-count* count)
    data))
```

Define a function to pick a random word from the set based on the relative frequencies.

```
(defun random-word ()
  (let ((num (act-r-random *total-count*)))
    (cdr (find-if (lambda (x) (< num (first x))) *word-list*))))
```

Make-one-goal picks a random verb from the list.

```
(defun make-one-goal ()
  (let ((word (random-word)))
```

Then it creates a chunk that only has the base form of that random verb in the verb slot and places it in the **imaginal** buffer using the set-buffer-chunk command.

```
    (set-buffer-chunk 'imaginal
                      (car (define-chunks-fct
                               (list (list 'verb (first word))))))
```

It places a copy of the starting-goal chunk into the **goal** buffer and returns the word being presented.

```
    (goal-focus starting-goal)
    word))
```

The add-past-tense-to-memory function adds a correctly formed past tense to the declarative memory of the model by placing it into the **imaginal** buffer and then clearing the buffer so that the chunk gets merged into declarative memory normally.

```
(defun add-past-tense-to-memory ()
  (let ((word (random-word)))
    (set-buffer-chunk 'imaginal
                      (car (define-chunks-fct
                               (list (mapcan (lambda (x y) (list x y))
                                       '(verb stem suffix) word)))))
    (clear-buffer 'imaginal)))
```

The print-header function just prints the column labels for the data output.

```
(defun print-header ()
  (format t "~%trials      Irregular       Regular    No inflection  Inflected correctly~%"))
```

The past-tense-results function prints out the performance of the model averaged over every 1000 trials and optionally draws a graph of the results if the optional parameter is true (which it is by default).

```
(defun past-tense-results (&optional (graph t))
  (print-header)
  (let ((data (rep-f-i 0 (length *report*) 1000)))
    (when (and graph (> (length data) 1))
      (graph-it data))
    data))
```

The graph-it function opens an experiment window and draws a graph of the model's correct performance for inflected irregular verbs (where the U-shaped learning should appear).

```
(defun graph-it (data)
  (let* ((win (open-exp-window "Irregular Verbs correct" :visible t :width 500 :height 475))
         (low (apply 'min data))
         (zoom (min .9 (/ (floor low .1) 10))))

    (clear-exp-window win)
    (add-text-to-exp-window win :text "1.0" :x 5 :y 5 :width 22)
    (add-text-to-exp-window win :text (format nil "~0,2f" zoom) :x 5 :y 400 :width 22)
    (add-text-to-exp-window win :text (format nil "~0,2f" (+ zoom (/ (- 1.0 zoom) 2)))
                            :x 5 :y 200 :width 22)

    (add-text-to-exp-window win "Trials" 200 420 100)
    (add-line-to-exp-window win '(30 10) '(30 410) 'black)
    (add-line-to-exp-window win '(450 410) '(25 410) 'black)

    (dotimes (i 10)
      (add-line-to-exp-window win (list 25 (+ (* i 40) 10)) (list 35 (+ (* i 40) 10)) 'black))

    (do* ((increment (max 1.0 (floor (/ 450.0 (length data)))))
          (range (floor 400 (- 1.0 zoom)))
          (intercept (+ range 10))
          (p1 (butlast data) (cdr p1))
          (p2 (cdr data) (cdr p2))
          (last-x 30 this-x)
          (last-y (- intercept (floor (* range (car p1))))
                  (- intercept (floor (* range (car p1)))))
          (this-x (+ last-x increment)
                  (+ last-x increment))
          (this-y (- intercept (floor (* range (car p2))))
                  (- intercept (floor (* range (car p2))))))
         ((null (cdr p1)) (add-line-to-exp-window win
                            (list last-x last-y) (list this-x this-y) 'red))
      (add-line-to-exp-window win (list last-x last-y)
                               (list this-x this-y)
                               'red))))
```

The safe-div function returns the result of n/d unless d is zero in which case it returns 0.

```
(defun safe-div (n d)
  (if (zerop d)
      0
    (/ n d)))
```

The rep-f-i function computes the average performance from a specified portion of the model's data in segments of the given length and prints out the lines of data and returns the list of the inflected irregular verb results.

```
(defun rep-f-i (start end count)
  (let ((data nil))
    (dotimes (i (ceiling (- end start) count))
      (let ((irreg 0)
            (reg 0)
            (none 0)
            (e (if (> end (+ start count (* i count)))
                   (+ start count (* i count))
                 end)))

        (dolist (x (subseq *report* (+ start (* i count)) e))
          (when (first x)
            (case (second x)
              (reg
               (incf reg))
              (irreg
               (incf irreg))
              (none
               (incf none)))))

        (let* ((total (+ irreg reg none))
               (correct (safe-div irreg (+ irreg reg))))
          (format t "~6d ~{~13,3F~^ ~}~%" e
            (list
             (safe-div irreg total)
             (safe-div reg total)
             (safe-div none total)
             correct))
          (push-last correct data))))
    data))
```

The add-to-report function takes a verb list and the name of a chunk which the model generated. It classifies the generated chunk based on how it formed the past tense and records that result in the global report. It also prints out warnings if the verb is malformed in anyway.

```
(defun add-to-report (target chunk)
  (let ((stem (chunk-slot-value-fct chunk 'stem))
        (word (chunk-slot-value-fct chunk 'verb))
        (suffix (chunk-slot-value-fct chunk 'suffix))
        (irreg (eq (third target) 'blank)))

    (if (eq (first target) word)
        (cond ((and (eq stem word) (eq suffix 'ed))
               (push-last (list irreg 'reg) *report*))
```

```
              ((and (null suffix) (null stem))
               (push-last (list irreg 'none) *report*))
              ((and (eq stem (second target)) (eq suffix 'blank))
               (push-last (list irreg 'irreg) *report*))
              (t
               (print-warning
                 (format nil "Incorrectly formed verb.  Presented ~s and produced ~{~s~^ ~}."
                             (first target)
                             (mapcan (lambda (x y) (list x y))
                               '(verb stem suffix) (list word stem suffix))))
               (push-last (list irreg 'error) *report*)))
      (progn
        (print-warning
          (format nil "Incorrectly formed verb.  Presented ~s and produced ~{~s~^ ~}."
                      (first target)
                      (mapcan (lambda (x y) (list x y))
                        '(verb stem suffix) (list word stem suffix))))
        (push-last (list irreg 'error) *report*)))))
```

Define another global variable which is used to check whether the model receives a reward, and then a function which will set that when a reward is given.

```
(defvar *reward-check*)

(defun verify-reward (&rest r)
  (declare (ignore r))
  (setf *reward-check* t))
```

The past-tense-trials function takes one required parameter and two optional ones. The required parameter specifies how many verbs the model will be given to create a past-tense. If the first optional parameter is provided as true then the model should continue from where it left off otherwise it will be reset before presenting the verbs. If the third optional parameter is given as true then the model trace will be shown while it runs, otherwise it will be turned off. It will print the averaged results every 100 trials as it runs.

```
(defun past-tense-trials (n &optional (cont nil)(v nil))
```

Add the command that will set the flag to indicate a reward was received and have it monitor the trigger-reward command.

```
  (add-act-r-command "reward-check" 'verify-reward
                     "Past tense code check for a reward each trial.")
  (monitor-act-r-command "trigger-reward" "reward-check")
```

If the cont parameter is nil or the word list has not been created then the model needs to be reset and all of the global data initialized, making sure to create chunks that represent all of the verbs if they are not already chunks.

```
  (when (or (null *word-list*) (null cont))
    (reset)
    (setf *word-list* (make-word-freq-list *verbs*))
    (let ((new nil))
      (dolist (x *word-list*)
        (mapcar (lambda (y) (pushnew y new)) (cdr x)))

      (dolist (x new)
        (unless (chunk-p-fct x)
          (define-chunks-fct (list (list x))))))

    (print-header)
    (setf *report* nil))
```

Set the value of the model's :v parameter based on the value provided to turn the trace off or on.

```
(sgp-fct (list :v v))
```

Loop over the n trials, adding two correct past tenses to the models memory before presenting a verb for it to generate. Run it for up to 100 seconds to generate the past tense, add that result to the data, clear the imaginal buffer so the current result can enter declarative memory, check whether it should print the results, and then print a warning if the model did not receive a reward or if it actually spent the entire 100 seconds trying to make the past tense.

```
(let* ((start (* 100 (floor (length *report*) 100)))
       (count (mod (length *report*) 100)))
  (dotimes (i n)
    (add-past-tense-to-memory)
    (add-past-tense-to-memory)
    (setf *reward-check* nil)
    (let ((target (make-one-goal))
          (duration (run 100)))
      (add-to-report target (buffer-read 'imaginal))
      (clear-buffer 'imaginal)
      (incf count)
      (when (= count 100)
        (rep-f-i start (+ start 100) 100)
        (setf count 0)
        (incf start 100))
      (unless *reward-check*
        (print-warning
         "Model did not receive a reward when given ~s."
         (first target)))
      (run-full-time (- 200 duration))

      (when (= duration 100)
        (print-warning
         "Model spent 100 seconds generating a past tense for ~s."
         (first target)))))
  (rep-f-i start (+ start count) 100))
```

Remove the monitor for trigger-reward when done.

```
(remove-act-r-command-monitor "trigger-reward" "reward-check")
(remove-act-r-command "reward-check")
nil)
```

Python

Start by importing the modules needed and loading the starting model for this task.

```
import actr
import math

actr.load_act_r_model("ACT-R:tutorial;unit7;past-tense-model.lisp")
```

Define some global variables to hold the collected data and generate the verbs with the appropriate frequencies.

```
report = []
total_count = 0
word_list = []

verbs = [['have','i',12458,'had'],
         ['do','i',4367,'did'],
         ['make','i',2312,'made'],
         ['get','i',1486,'got'],
         ['use','r',1016,'use'],
         ['look','r',910,'look'],
         ['seem','r',831,'seem'],
         ['tell','i',759,'told'],
         ['show','r',640,'show'],
         ['want','r',631,'want'],
         ['call','r',627,'call'],
         ['ask','r',612,'ask'],
         ['turn','r',566,'turn'],
         ['follow','r',540,'follow'],
         ['work','r',496,'work'],
         ['live','r',472,'live'],
         ['try','r',472,'try'],
         ['stand','i',468,'stood'],
         ['move','r',447,'move'],
         ['need','r',413,'need'],
         ['start','r',386,'start'],
         ['lose','i',274,'lost']]
```

Define a function to build the word list for use in the experiment. The list for each verb will include a count for generating a random verb based on its frequency in English, the verb, and its correct past tense.

```
def make_word_freq_list (l):

    data = []
    count = 0

    for verb in l:
        count += verb[2]
        if verb[1] == 'i':
            suffix = 'blank'
        else:
            suffix = 'ed'

        data.append([count,verb[0],verb[3],suffix])
    global total_count
    total_count = count
    return(data)
```

Define a function to pick a random word from the set based on the relative frequencies.

```
def random_word():

    num=actr.random(total_count)

    for i in word_list:
        if i[0] > num:
            return(i[1:])
```

Make_one_goal picks a random verb from the list.

```
def make_one_goal():

    word = random_word()
```

Then it creates a chunk that only has the base form of that random verb in the verb slot and places it in the **imaginal** buffer using the set_buffer_chunk command.

```
    actr.set_buffer_chunk('imaginal',actr.define_chunks(['verb',word[0]])[0])
```

It places a copy of the starting-goal chunk into the **goal** buffer and returns the word being presented.

```
    actr.goal_focus('starting-goal')

    return(word)
```

The add_past_tense_to_memory function adds a correctly formed past tense to the declarative memory of the model by placing it into the **imaginal** buffer and then clearing the buffer so that the chunk gets merged into declarative memory normally.

```
def add_past_tense_to_memory ():

    word = random_word()

    actr.set_buffer_chunk('imaginal',
                          actr.define_chunks(['verb',word[0],
                                              'stem',word[1],
                                              'suffix',word[2]])[0])
    actr.clear_buffer('imaginal')
```

The print_header function just prints the column labels for the data output.

```
def print_header():
    print ()
    print ( "trials      Irregular       Regular    No inflection  Inflected correctly")
```

The results function prints out the performance of the model averaged over every 1000 trials and optionally draws a graph of the results if the optional parameter is true (which it is by default).

```
def results(graph=True):

    print_header()
    data = rep_f_i(0,len(report),1000)
    if graph and len(data) > 1:
        graph_it(data)
    return(data)
```

The graph_it function opens an experiment window and draws a graph of the model's correct performance for inflected irregular verbs (where the U-shaped learning should appear).

```
def graph_it(data):

    win = actr.open_exp_window("Irregular Verbs correct", visible=True,height=500,width=475)
    low = min(data)
    zoom = min([.9,math.floor(10 * low)/10])

    actr.clear_exp_window(win)
    actr.add_text_to_exp_window(win,text="1.0",x=5,y=5,width=22)
    actr.add_text_to_exp_window(win,text="%0.2f" % zoom,x=5,y=400,width=22)
    actr.add_text_to_exp_window(win,text="%0.2f" % (zoom + ((1.0 - zoom) / 2)),
                                x=5,y=200,width=22)

    actr.add_text_to_exp_window(win,"Trials",200,420,100)
    actr.add_line_to_exp_window(win,[30,10],[30,410],'black')
    actr.add_line_to_exp_window(win,[450,410],[25,410],'black')

    for i in range(10):
        actr.add_line_to_exp_window(win,[25,10 + i*40],[35,10 + i*40],'black')

    start = data[0]
    increment = max([1.0,math.floor( 450 / len(data))])
    r = math.floor (400/ (1.0 - zoom))
    intercept = r + 10
    lastx =30
    lasty = intercept - math.floor(r * start)

    for p in data[1:]:
        x = lastx + 30
        y = intercept - math.floor(r * p)

        actr.add_line_to_exp_window(win,[lastx,lasty],[x,y],'red')
        lastx = x
        lasty = y
```

The safe_div function returns the result of n/d unless d is zero in which case it returns 0.

```
def safe_div(n, d):
    if d == 0:
        return(0)
    else:
        return (n / d)
```

The rep_f_i function computes the average performance from a specified portion of the model's data in segments of the given length and prints out the lines of data and returns the list of the inflected irregular verb results.

```
def rep_f_i(start,end,count):

    data = []

    for i in range(math.ceil( (end - start) / count)):
        irreg = 0
        reg = 0
        none = 0

        if end > (start + count + (i * count)):
            e = start + count + (i * count)
        else:
            e = end

        for x in report[(start + (i * count)):e]:
            if x[0]:
                if x[1] == 'reg':
                    reg += 1
                elif x[1] == 'irreg':
                    irreg += 1
                elif x[1] == 'none':
                    none += 1

        total = irreg + reg + none
        correct = safe_div(irreg, (irreg + reg))

        print("%6d %13.3f %13.3f %13.3f %13.3f"%
              (e,safe_div(irreg,total),safe_div(reg,total),safe_div(none,total),correct))

        data.append(correct)

    return data
```

The add_to_report function takes a verb list and the name of a chunk which the model generated. It classifies the generated chunk based on how it formed the past tense and records that result in the global report. It also prints out warnings if the verb is malformed in anyway.

```
def add_to_report(target, chunk):
    global report

    stem = actr.chunk_slot_value(chunk,"stem")
    word = actr.chunk_slot_value(chunk,"verb")
    suffix = actr.chunk_slot_value(chunk,"suffix")
    irreg = (target[2] == 'blank')
    if target[0].lower() == word.lower():
        if stem == word and suffix.lower() == 'ed':
            report.append([irreg,'reg'])
        elif stem == None and suffix == None:
            report.append([irreg,'none'])
        elif stem.lower() == target[1].lower() and suffix.lower() == 'blank':
            report.append([irreg,'irreg'])
        else:
            actr.print_warning(
              "Incorrectly formed verb. Presented %s and produced verb %s,stem %s,suffix %s."%
              (target[0],word,stem,suffix))
    else:
        actr.print_warning(
            "Incorrectly formed verb. Presented %s and produced verb %s,stem %s,suffix %s."%
            (target[0],word,stem,suffix))
        report.append([irreg,'error'])
```

Define another global variable which is used to check whether the model receives a reward, and then a function which will set that when a reward is given.

```
reward_check = False

def verify_reward(*params):
    global reward_check
    reward_check = True
```

The trials function takes one required parameter and two optional ones. The required parameter specifies how many verbs the model will be given to create a past-tense. If the first optional parameter is provided as true then the model should continue from where it left off otherwise it will be reset before presenting the verbs. If the third optional parameter is given as true then the model trace will be shown while it runs, otherwise it will be turned off. It will print the averaged results every 100 trials as it runs.

```
def trials(n,cont=False,v=False):

    global report,word_list,reward_check
```

Add the command that will set the flag to indicate a reward was received and have it monitor the trigger-reward command.

```
    actr.add_command("reward-check",verify_reward,
                     "Past tense code check for a reward each trial.")
    actr.monitor_command("trigger-reward","reward-check")
```

If the cont parameter is False or the word list has not been created then the model needs to be reset and all of the global data initialized, making sure to create chunks that represent all of the verbs if they are not already chunks.

```
    if not(cont) or not(word_list):
        actr.reset()
        word_list = make_word_freq_list(verbs)
        new = []
        for x in word_list:
            for y in x[1:]:
```

```
            if y not in new:
                new.append(y)
    for x in new:
        if not(actr.chunk_p(x)):
            actr.define_chunks([x])

    print_header()
    report = []
```

Set the value of the model's :v parameter based on the value provided to turn the trace off or on.

```
actr.set_parameter_value(":v",v)
```

Loop over the n trials, adding two correct past tenses to the models memory before presenting a verb for it to generate. Run it for up to 100 seconds to generate the past tense, add that result to the data, clear the imaginal buffer so the current result can enter declarative memory, check whether it should print the results, and then print a warning if the model did not receive a reward or if it actually spent the entire 100 seconds trying to make the past tense.

```
start = 100 * math.floor(len(report) / 100)
count = len(report) % 100

for i in range(n):
    add_past_tense_to_memory()
    add_past_tense_to_memory()
    reward_check = False
    target = make_one_goal()
    duration = actr.run(100)[0]
    add_to_report(target,actr.buffer_read('imaginal'))
    actr.clear_buffer('imaginal')
    count += 1

    if count == 100:
        rep_f_i(start, start + 100, 100)
        count = 0
        start += 100
    if not(reward_check):
        actr.print_warning("Model did not receive a reward when given %s."% target[0])

    actr.run_full_time(200 - duration)

    if duration == 100:
        actr.print_warning("Model spent 100 seconds generating a past tense for %s."%
                           target[0])

rep_f_i(start,start+count,100)
```

Remove the monitor for trigger-reward when done.

```
actr.remove_command_monitor("trigger-reward","reward-check")
actr.remove_command("reward-check")
```

New Commands

set-buffer-chunk and **set_buffer_chunk** can be used to set a buffer to hold a copy of a particular chunk. They take two parameters which must be the name of a buffer and the name of a chunk respectively. It copies that chunk into the specified buffer (clearing any chunk which may have been in the buffer at that time) and returns the name of the chunk which is now in the buffer. This acts very much like the goal-focus command, except that it works for any buffer. An important difference however is that the goal-focus command actually schedules an event which uses set-buffer-chunk to put the chunk in the goal buffer. That is important because scheduled events display in the model trace and are changes to which the model can react. There is another command which works more like goal-focus which is often more appropriate (schedule-set-buffer-chunk the details of which you can find in the reference manual), but this experiment's code doesn't do that in either place where the command is used. In the first instance it's because it is putting a chunk into the buffer and then immediately removing it without needing to run the model, and in the other instance putting a chunk into the buffer is also accompanied by a goal-focus call which generates an event to which the model can respond so it's not necessary to create another event (although it would not affect anything in this task if it did do it the "right" way by scheduling the change).

clear-buffer and **clear_buffer** can be used to clear a chunk from a buffer the same way a *–buffer>* action in a production will. It takes one parameter which should be the name of a buffer and that buffer is emptied and the chunk is merged into declarative memory. It returns the name of the chunk that was cleared. Like set-buffer-chunk, the clear-buffer command does not create an event to which the model can respond, and there is a corresponding schedule-clear-buffer which could be used to do that.

chunk-p and **chunk_p** can be used to determine if the provided name is actually the name of a chunk in the current model. They return a true result if it is (t/True) and false if not (nil/None).

push-last is a macro for Lisp which adds an item to the end of a list. It takes two parameters, any item and a list. It destructively adds the item to the end of the list. This is often more convenient than the standard Lisp function push which adds to the beginning of a list.

trigger-reward and **trigger_reward** were not actually used in the code, but it was monitored to determine if the model received a reward. This command can be called to provide a reward for utility learning to the model at any time, and it schedules an event to provide that reward at the current time. It requires one parameter which indicates the reward. If that reward value is numeric then it is used to update the utilities of those productions which have been selected since the last reward. If it is not a numeric value then it only serves to mark the last reward time for preventing further propagation of the next reward signal.

Procedural Learning Modeling Issues

This text will cover some of the issues that can arise when working with utility learning and production compilation and describe ways that the Environment tools may be used to help. Models which use the procedural learning mechanisms often do so in conjunction with declarative learning and also often require running for many trials for the learning effects to show up. Because of that, such models can be more difficult to analyze and debug since one may need to investigate both declarative and procedural issues over long runs. To better demonstrate things for this text we will be using two simple tasks which are focused only on the procedural issues involved. The mechanisms described here can be used in conjunction with those described previously for declarative memory, and we will also indicate ways of dealing with longer runs.

Utility Learning

First we will look at a model which is using utility learning in a task similar to the choice experiment from unit 6 found in the "utility-learning-issues.lisp" file. The code for the task can be found in the ul-issues.lisp and ul_issues.py files. We do not have any experimental data which we will be fitting with this model, but we do have an expectation that it will learn which choice is better. That learning should show up as a higher utility for the production which chooses the better response and we will look for that as the model runs.

The Task

In this task the model must choose one of two options, either A or B, within five seconds. Then after the five seconds have passed the model will either be presented with the correct response for this trial or informed that no answer will be provided for the trial. The feedback will be presented for two seconds and then the next trial will begin. Thus, each trial lasts exactly seven seconds. For this task, choice A will be reported as correct on 60% of the trials, 20% of the trials will indicate choice B as correct, and 20% of the trials will provide no feedback. Thus, we will expect the model to learn to choose option A more frequently than option B.

Because this task only has to run with the model it has been implemented by directly manipulating the chunks in the model's **goal** and **imaginal** buffers. The task will put a chunk in the **goal** buffer indicating that it is time to choose and then put a chunk in the **imaginal** buffer with the feedback five seconds later. Two seconds after that it will provide another **goal** chunk indicating the next choice time, and that process will repeat for as long as the model runs. The task operates by scheduling the actions to occur for the model, and does not require calling a function other than run. There is no data collected or results reported for the task, but the choice and feedback actions will be shown in the medium detail trace for reference. Because the **imaginal** buffer tests a slot that is set from code outside of the model we have used declare-buffer-usage to avoid the style warnings about that slot's usage in the productions.

The Model

Because the task is directly modifying the chunks in the buffers, the model can simply consist of five productions. Two productions respond to the **goal** buffer chunk indicating that it is time to choose, one for each choice, and there are three productions which process the feedback provided in the **imaginal** buffer. The feedback handling productions consist of one which fires when the model chose correctly, one which fires when it chose incorrectly, and one which fires when there is no feedback for the trial. We will not show the productions here, but there is nothing new or unusual about them so they should be easy to understand by looking at the model file. The only learning mechanism enabled in the model is utility learning and the model has been given some noise in utilities with these parameter settings:

```
(sgp :esc t :ul t :egs .5)
```

The utility learning rate parameter :alpha is not set so it will have the default value of .2. To allow the model to learn, the productions which fire for matching and mismatching feedback are given the following rewards:

```
  (spp response-matches :reward 4)
  (spp response-doesnt-match :reward 0)
```

The productions are not given any particular starting utilities. Therefore, they will all start with the default utility of 0.

The last line of the model definition schedules the first choose event to happen at time 0, and that starts the cycle of scheduling the events to drive the task as the model runs.

Testing the Model

Loading the ul-issues.lisp file or importing the ul_issues.py file will automatically load the model file, and when we do so there are no warnings or errors reported so we can start running it now. One thing that we could do would be to just run it for several trials and then see how the utilities have changed by that point. If the choose-a production has a higher utility than choose-b we might then consider the model done. However, as has been mentioned in the other testing texts, it is always better to start small and make sure to understand how the model is working and learning before moving on to look at the higher level results.

As a first test we should run a couple of trials and make sure the model is operating as we would expect. Here is the trace from the first trial after running for just under seven seconds:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 4
     5.050   PROCEDURAL             CONFLICT-RESOLUTION
     6.950   ------                 Stopped because time limit reached
```

We see the model choose A, the feedback presented is that A is the correct choice, then the model fires the response-matches production and a reward of 4 is applied. That looks good, but we should check a couple more trials to make sure. Here is the trace for the next two:

```
     7.000   NONE                   utility-learning-issues-choose
     7.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     7.000   PROCEDURAL             CONFLICT-RESOLUTION
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    20.950   ------                 Stopped because time limit reached
```

There we see trials with the model responding to both the lack of feedback and a trial when it responds incorrectly. Since that all looks good we should now look at how the utility learning is progressing as it runs.

Because this is a small model it should be easy to follow the learning by simply enabling the utility learning trace and watching the values change. If it were a larger model however that might not be as tractable, and we might need to use some of the Environment tools to help as will be discussed later. For now, we will just enable the utility learning trace by turning it on after resetting the model using sgp in Lisp or set_parameter_value in Python:

```
? (sgp :ult t)
```

```
>>> actr.set_parameter_value(':ult',True)
```

When we run it now we see the update to utility for the first reward:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 4
 Utility updates with Reward = 4.0   alpha = 0.2
```

```
  Updating utility of production CHOOSE-A
   U(n-1) = 0.0   R(n) = -1.0500002 [4.0 - 5.05 seconds since selection]
   U(n) = -0.21000004
  Updating utility of production RESPONSE-MATCHES
   U(n-1) = 0.0   R(n) = 3.95 [4.0 - 0.05 seconds since selection]
   U(n) = 0.79
```

Looking at the change in utility for the production choose-a on this trial indicates that there seems to be a problem. The model chose A and the feedback provided indicated that A was the correct choice which lead to a positive reward, but the utility of the choose-a production decreased from 0 to -0.21. The reason for that is because the effective reward a production receives is discounted by the time that passed between the production's selection and when the reward is received. In this task there are 5.05 seconds between the choice and the reward. Thus, with a reward of 4 being provided on a correct response we end up penalizing the production.

Generally, that is not a good situation since it means that the model would be less likely to choose A after positive feedback. If we want the reward to have a positive effect then we should make sure that it is large enough to do so considering the amount of time that passes. To fix that for this model we will adjust the reward provided for being correct to 6 instead of 4:

```
  (spp response-matches :reward 6)
```

While we are editing the model file we should also add the :ult setting so that we don't have to keep setting it each time we reset to start a new run:

```
  (sgp :esc t :ul t :egs .5 :ult t)
```

After making that change and saving the model here is the trace of the utility learning now:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 6
 Utility updates with Reward = 6.0   alpha = 0.2
  Updating utility of production CHOOSE-A
   U(n-1) = 0.0   R(n) = 0.9499998 [6.0 - 5.05 seconds since selection]
   U(n) = 0.18999997
  Updating utility of production RESPONSE-MATCHES
   U(n-1) = 0.0   R(n) = 5.95 [6.0 - 0.05 seconds since selection]
   U(n) = 1.1899999
```

We now see a positive change in the choose-a production's utility after choosing it correctly on this trial.

Had we just been looking at the model's performance over a long run we may not have noticed this oddity in the model's learning pattern. For example, had we just run the model for 100 trials from the initial state and looked at the resulting utilities we would have seen something like this if we use spp to print out the results:

```
? (spp choose-a choose-b)

>>> actr.spp('choose-a','choose-b')
```

```
Parameters for production CHOOSE-A:
 :utility -4.107
 :u  -4.405
 :at  0.050
 :reward    NIL
Parameters for production CHOOSE-B:
 :utility -5.604
 :u  -5.618
 :at  0.050
 :reward    NIL
(CHOOSE-A CHOOSE-B)
```

Production choose-a has a higher utility than choose-b which means that the model will be choosing A more often than B. So, even with a successful choice penalizing the model initially, in the long term the model still gets to the expected result since presumably the incorrect trials are penalized even more, but if we are concerned with how it gets there, which often is the reason for creating a learning model, we should pay attention to the details along the way. In this case, the negative utilities may have been an indication that there was a problem, but if instead of looking at the utilities we had been looking at response data like choice percentages we may not have noticed at all.

Now that we have the first trial operating in a reasonable manner we will look at the next trial. Here is the trace for the second trial:

```
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   ------                 Stopped because time limit reached
```

The model chooses A again but this time there is no feedback. Because the unknown-response production provides no reward there is no change to the utility of the choose-a production. So we will now look at the next trial:

```
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
```

```
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-A
   U(n-1) = 0.18999997   R(n) = -12.05 [0.0 - 12.05 seconds since selection]
   U(n) = -2.258
  Updating utility of production UNKNOWN-RESPONSE
   U(n-1) = 0.0   R(n) = -7.05 [0.0 - 7.05 seconds since selection]
   U(n) = -1.4100001
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   NONE                   utility-learning-issues-choose
    21.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    21.000   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   ------                 Stopped because time limit reached
```

This time the model chooses B, but the feedback indicates that A was the correct choice. The response-doesnt-match production fires and provides a reward of 0 which gets propagated back. However, in addition to penalizing the choose-b production as we would expect it also penalizes the choose-a production. That happens because the reward affects all productions which have fired since the previous reward, which occurred after response-matches fired on the first trial. Because there was no reward provided on the second trial when unknown-response fired this reward gets applied to the productions for that trial as well.

To prevent that from happening we will have to provide a reward on the trials without any feedback when unknown-response fires. The question becomes how much reward should we provide when there is no feedback? As always, there is no single answer to such a question and depending on the task and hypothesis behind the model, values anywhere between the positive and negative feedback may be appropriate. Alternatively, instead of picking a value, there is a special option available for the reward which we will describe and use here.

The utility learning mechanism provides the option of specifying a "null reward". Such a reward does not adjust the utilities of any productions, but it does cause the marker for when the last reward was provided to be updated. That allows the modeler to indicate that there was nothing to be learned since the last reward was provided. As with choosing which reward values to provide, the modeler will have to decide if a null reward value is appropriate for any particular situation.

Any non-numeric true value provided as a reward results in a null reward for the model. If one is providing rewards to the model automatically with the firing of productions, as is done in this example, then setting the reward value for a production to t instead of a number is how one specifies the null reward. If instead one is using the trigger-reward command to provide rewards to the model directly then any true non-numeric value can be provided to produce the null reward. The reason for allowing any value when using trigger-reward is because it will show in the trace and will be passed to monitoring commands which may be helpful when looking at the trace or developing the task code.

Here is the setting we will add to the model to provide a null reward when there is no feedback for a trial:

```
(spp unknown-response :reward t)
```

By providing a null reward when the unknown-response production fires it will stop the reward from the next trial from propagating back past that point.

After saving that change in the model and reloading it here is what we see now for the utility update on the second and third trials:

```
     7.000   PROCEDURAL             CONFLICT-RESOLUTION
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   UTILITY                PROPAGATE-REWARD T
  Non-numeric reward clears utility learning history.
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   NONE                   utility-learning-issues-choose
    21.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    21.000   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   ------                 Stopped because time limit reached
```

On the second trial it now reports that there is a null reward which clears the history and sets a new marker for the last reward given. Then on the third trial only the choose-b and response-doesnt-match productions get an update to their utilities.

After that change the model seems to be working as we would expect now – it gets a positive reward for guessing correctly, no change to rewards when there is no feedback, and a negative reward when it guesses incorrectly. If we check the utility values of the choose-a and choose-b productions now we see that the :u value for choose-a is greater than the :u value for choose-b:

```
Parameters for production CHOOSE-A:
 :utility  1.111
 :u   0.190
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production CHOOSE-B:
 :utility -1.961
 :u  -1.010
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

As a test, we can run the model for several more trials and look at the results, and we can turn off the trace using with-parameters, sgp, or the set_parameter_values functions so it runs faster, and then look at the parameter values:

```
Parameters for production CHOOSE-A:
 :utility -1.665
 :u  -0.646
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production CHOOSE-B:
 :utility -3.326
 :u  -2.982
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

We see that choose-a still has the greater U(n) value, though both are negative. Before considering why they are both negative, we will compute the probability that the model will fire choose-a instead of choose-b at this time using the equation from unit 6 of the tutorial:

$$\Pr obability(choose-a)=\frac{e^{\frac{-0.646}{.5*\sqrt{2}}}}{e^{\frac{-0.646}{.5*\sqrt{2}}}+e^{\frac{-2.982}{.5*\sqrt{2}}}}\approx .965$$

That is likely a little higher than we would want if we were trying to fit human performance, but without any explicit data to fit we will not adjust that in this model.

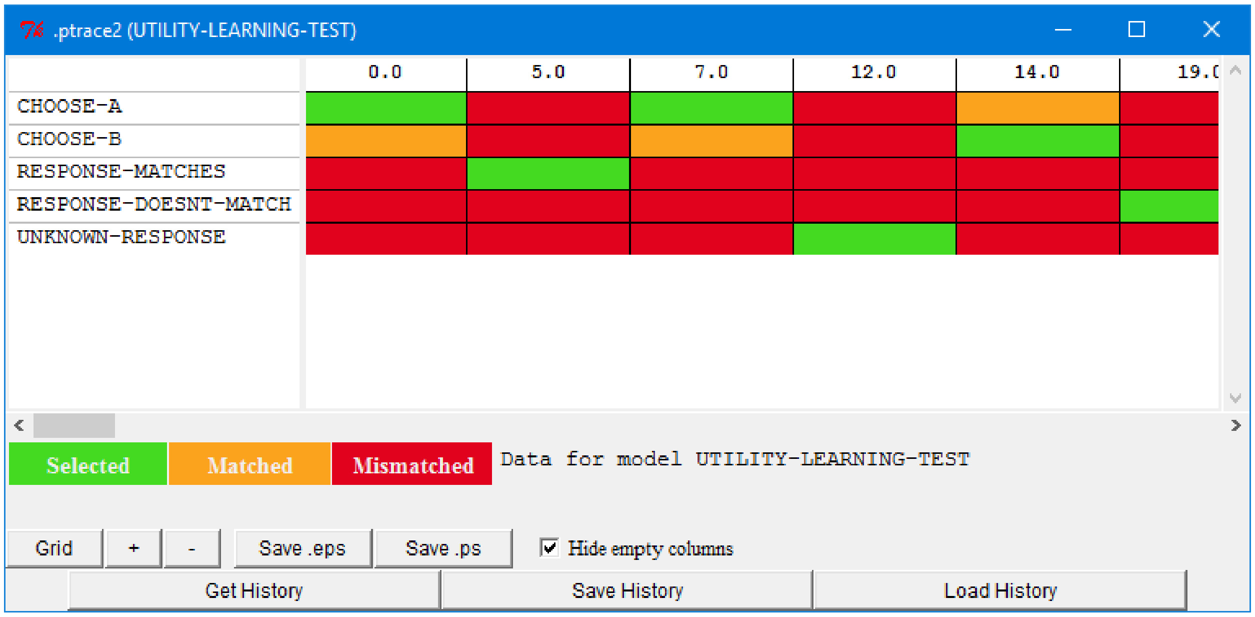
Now, as for why the values are negative, if we look back at the traces we will see that the penalty for an incorrect response is -5.05 whereas the benefit for a correct response is only +0.95. That much larger penalty for being incorrect appears to be what is driving the values negative, but we will look into that further below to make sure there is not some other issue. Because the utility values are only meaningful in comparison among competing productions, having negative values is not in and of itself a bad thing that always needs to be corrected. One situation where that might be an issue however is if one is using production compilation and has left the default utility value for newly learned productions at 0. If the original productions have negative utilities then the newly learned productions with utilities of 0 will be immediately more likely to be selected. That situation is not recommended and one would likely want to adjust the starting utilities of the original productions, adjust the initial utility for new productions, or adjust the rewards that are provided so that a more gradual introduction of the newly learned productions occurs. Since this model is not using production compilation, as long as we do not find something wrong with how it is operating we will not attempt to adjust the rewards or other parameters to eliminate the negative utilities.

Using the utility trace to investigate the changes to the utility of the productions works alright when dealing with a few trials in a small model, but if the task requires lots of trials or has lots of competing productions then reading through the trace can be difficult and time consuming. The "Production" tools in the Environment, which were introduced in the unit 3 modeling text, may help to investigate utility issues for longer runs and we will look at doing so below. Using those tools may not always explain what has happened, but when they do not they should at least help to find where problems are occurring so that a more detailed investigation can be done using more fine grain tools.

With the production grid we can get an overview of which productions are competing and which one, if any, is selected. As described in the unit 3 text, to use the tool we should open it before running the model so that it can record the data needed. When working with longer runs it can also help to have the tool hide the empty columns. That can be done by checking the "Hide empty columns" box near the bottom of the window. We will turn off the trace in the model definition for now since we will be looking at the information through the Environment:

```
(sgp :esc t :ul t :egs .5 :ult t :v nil)
```

We will save that change, reload the model, and then open the "Production" grid tool so that it records the data. We will run it for 40 trials and then press the "Get history" button to see the data. That should result in a display which looks something like this:



.ptrace2 (UTILITY-LEARNING-TEST)

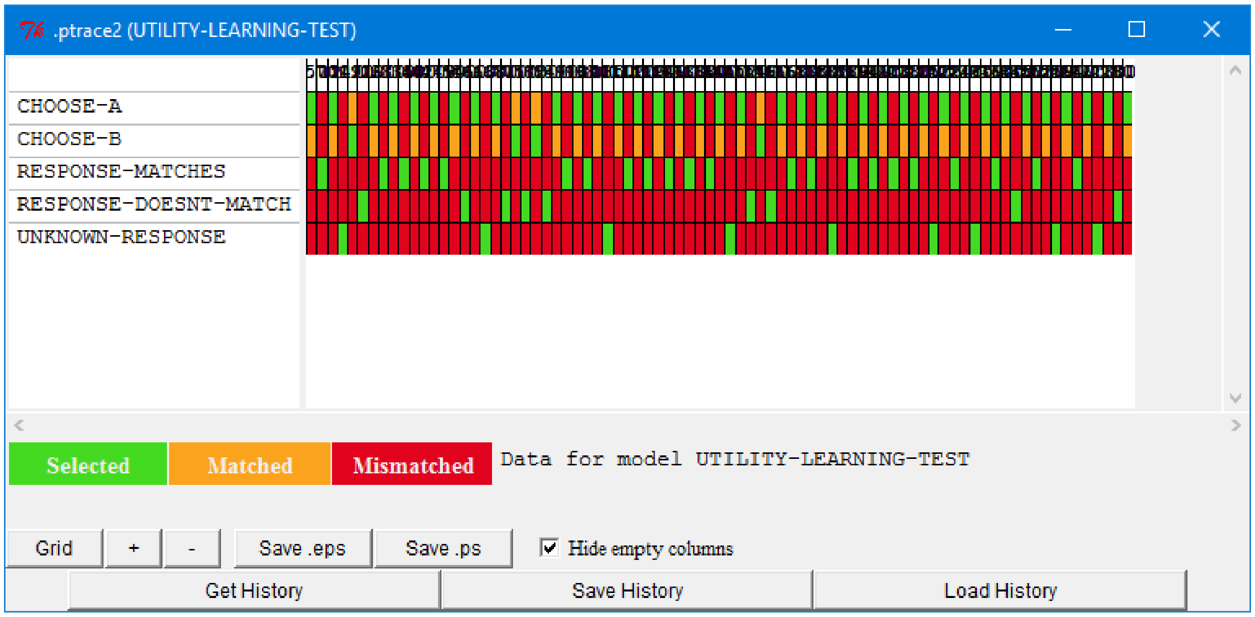
	0.0	5.0	7.0	12.0	14.0	19.
CHOOSE-A	🟩	🟥	🟩	🟥	🟧	🟥
CHOOSE-B	🟧	🟥	🟧	🟥	🟩	🟥
RESPONSE-MATCHES	🟥	🟩	🟥	🟥	🟥	🟥
RESPONSE-DOESNT-MATCH	🟥	🟥	🟥	🟥	🟥	🟩
UNKNOWN-RESPONSE	🟥	🟥	🟥	🟩	🟥	🟥

🟩 Selected 🟧 Matched 🟥 Mismatched

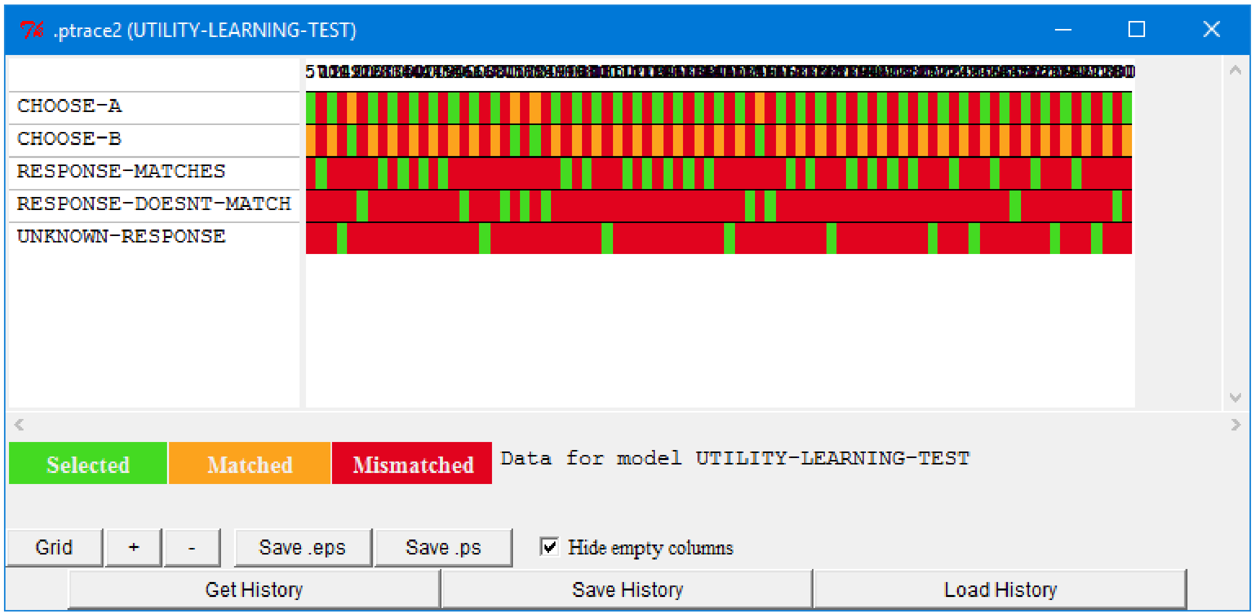
Data for model UTILITY-LEARNING-TEST

[✓ Hide empty columns]

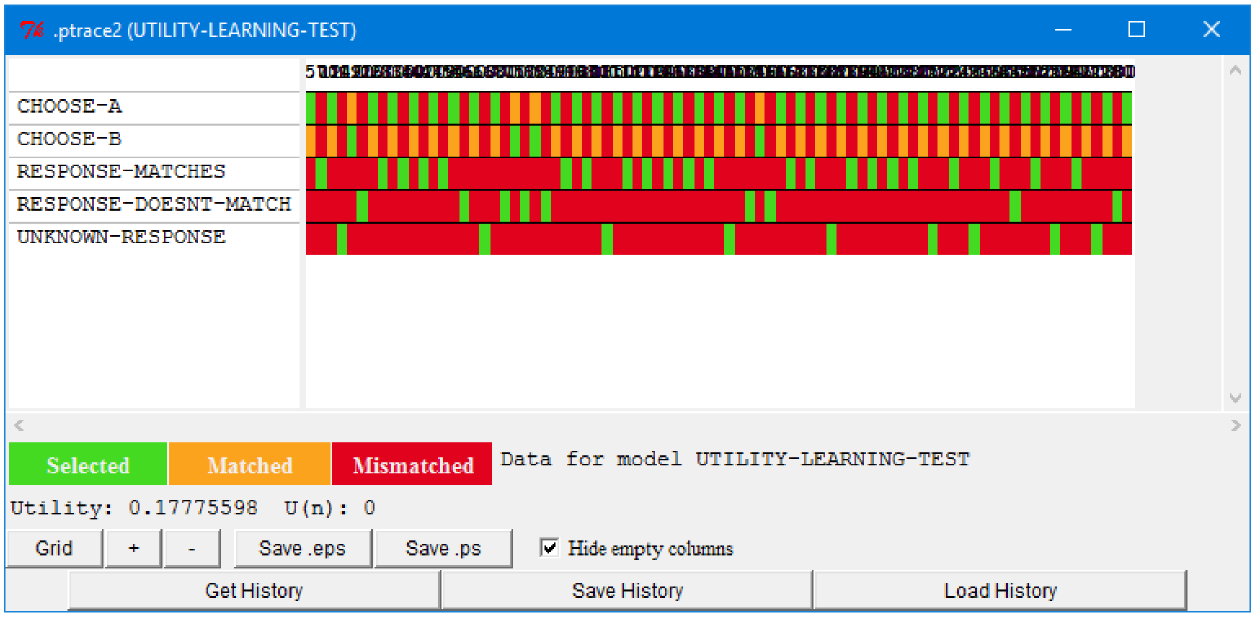
Each column is a conflict-resolution event. The green and orange productions indicate which ones matched the current state and the green one is the one that was selected. Above we can see the first three trials where the model chose A the first two times and then B on the third one. We could scroll the view horizontally to see all the trials, but looking at the whole sequence at once can often be more informative. To do that, we need to zoom out the display by pressing the "-" button at the bottom of the window. After pressing that a few times we can have the entire 40 trial sequence visible at once and that will look like this:



You may not always want to zoom out that far, but for purposes of this example we will look at the entire run. One thing that can help when zooming out with this tool is to turn off the display of the black likes separating the columns. To do that you can press the "Grid" button at the bottom and then the display will look like this which may be a little easier to look at:

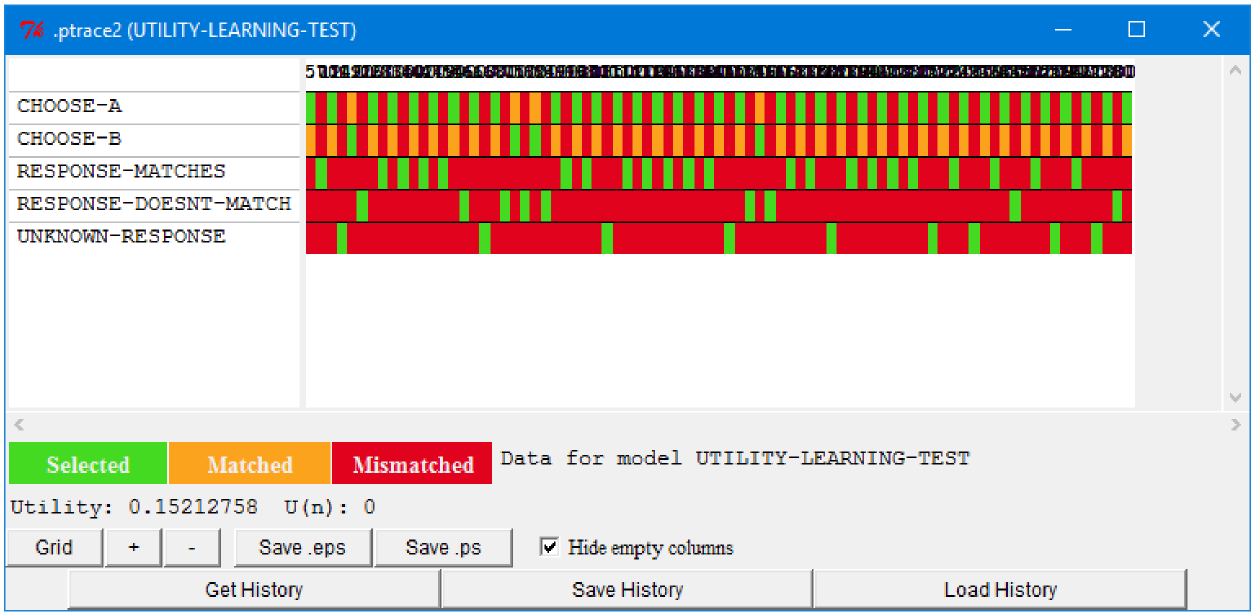


Looking at that display we can see that choose-a gets selected a lot more often than choose-b which is what we expected from the model. If we are interested in the utility values at particular times we can also see those by placing the mouse cursor over the green or orange bars in the display. Here is what it shows for the first green bar in the choose-a row:



```
Data for model UTILITY-LEARNING-TEST
Utility: 0.17775598  U(n): 0
```

It shows the noisy utility value which was used during that conflict-resolution action and the true U(n) value for the production at that time. In this case the U(n) is 0 since that is before any rewards have been applied. If we look at the first choose-b occurrence (the orange box) we see that it also has a U(n) of 0 and its utility was less than the utility of choose-a which is why choose-a was chosen at that time:

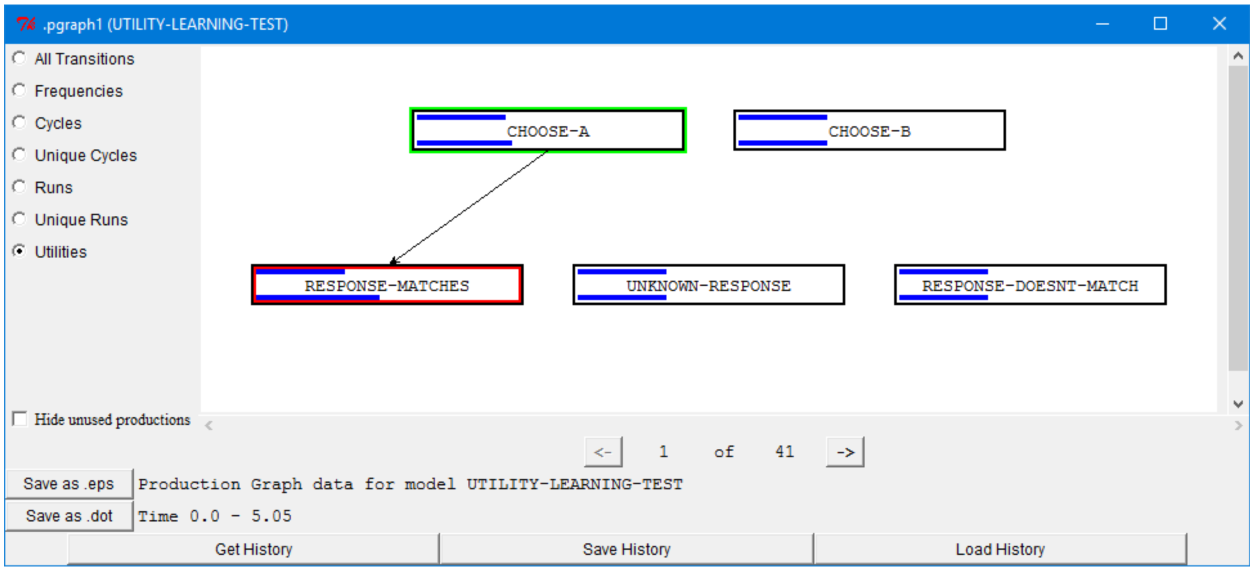


```
Data for model UTILITY-LEARNING-TEST
Utility: 0.15212758  U(n): 0
```

Using this tool we could look at the utility change for each trial to see how things are changing from trial to trial and you should feel free to investigate that. However, we will not be walking though that in this text. Instead, we will look at an alternative way to view that information using the "Production" graph tool.

The "Production" graph tool can display the sequence of production firings broken into segments based on when the model received rewards and in that view it will also show the utility changes which occurred. The "Production" graph tool relies on the same recorded data as the "Production" grid tool so since we have been using that tool the information is already available so we can just open the "Production" graph tool now and view the data without having to run the model again.

After opening a production graph display, to get the information we are interested in we need to select the "Utilities" option on the left and then press the "Get History" button. That will result in a display which looks like this after adjusting the window size to see all of the boxes:



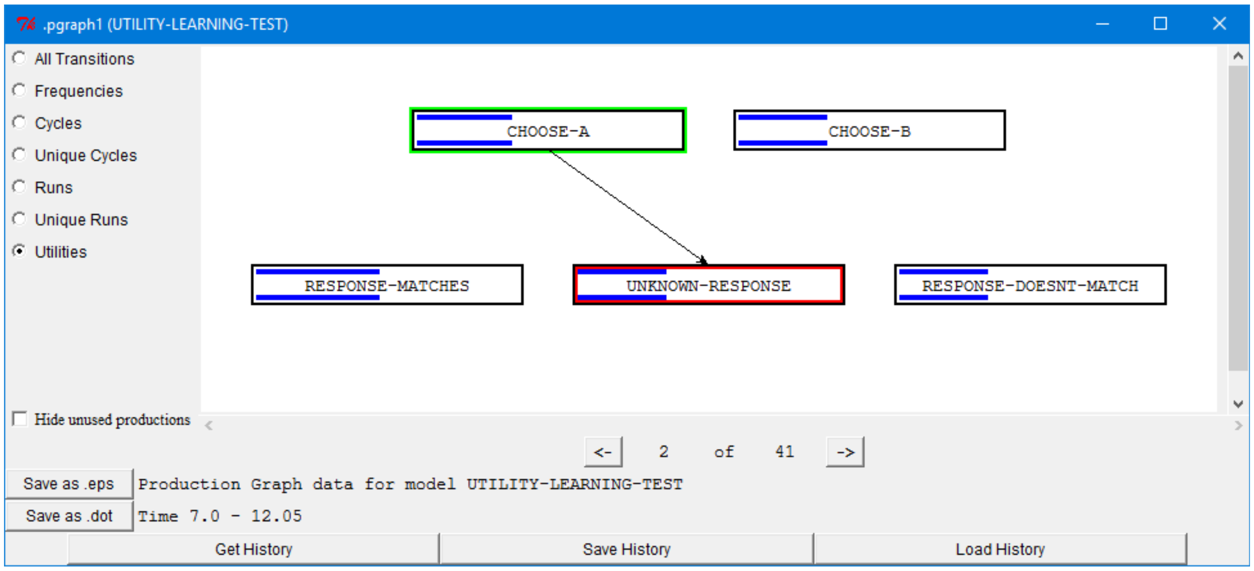
```
Production Graph data for model UTILITY-LEARNING-TEST
Time 0.0 - 5.05
```

The display is similar to the one shown earlier in the tutorial: it displays the sequence of productions which fired as a directed graph starting at the production highlighted in green and ending with a production highlighted in red. The "Utilities" display however differs in a couple of ways from those seen previously. The first is that now the run is broken up into separate graphs based on when the model receives rewards. The red highlighted productions will be the last production to fire before a reward is received (except for the last display where it might be just the final production which the model has fired whether or not it is followed by a reward). Since we have a reward provided on each trial in this task there will be one graph for each trial of this run, but the display shows that there are 41 total graphs to view. That is because the model has already selected a production at the start of the 41st trail which results in another graph to be displayed. The other difference from the previous production graph displays is that now in each production's box we see two blue lines. The one at the top represents the true utility of the production before the reward was provided and the one below represents the true utility after the reward has been propagated. The bars start at the left of the box and increase in length with the utility value. All of the productions are displayed in boxes of the same width and the utilities are scaled across all of the productions and graphs. A blue bar of length zero represents the minimum utility value that any production has across the entire run and a bar the width of the production box will be the maximum utility that occurs for any production over the entire run.

While this display does not show the actual values of the utilities, the relative changes that it does show should be sufficient to verify that things are working as desired and should be easier to go through than reading all the utility trace information. We will only show a couple of the graphs here for reference, but you may want to step through all of them on your own to make sure you understand how the model operates.

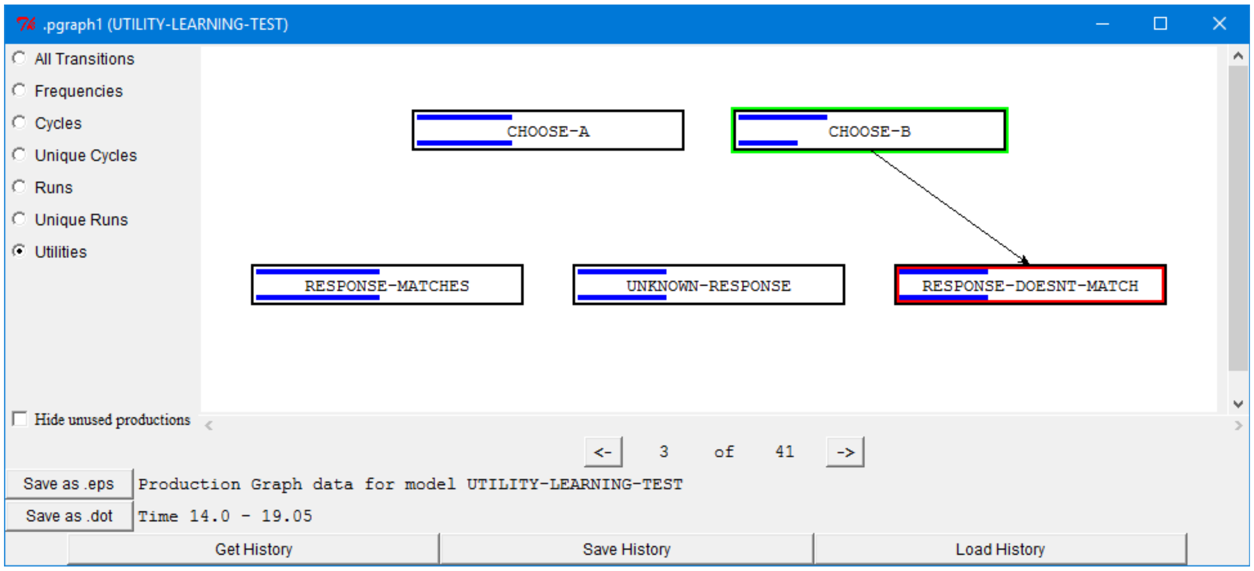
Looking at the display above we see that the model chose A on the first trail and that the response-matches production fires indicating a correct choice. When that happens we see that both the choose-a and response-matches productions had their utilities increased while the others stayed the same (which we also saw earlier in the utility trace). By using the graphic display it should be easier to look at the changes that occur on each trial than it would be to read through all of the utility traces. We will only show a couple more examples from this run below, but you may want to look at the whole sequence to verify for yourself that it works as expected.

Here is the second trial where choose-a is fired and then no feedback is provided:



Production Graph data for model UTILITY-LEARNING-TEST
Time 7.0 - 12.05

On that trail we see that none of the utilities have changed. Then on the third trial we see choose-b as the first production fired followed by response-doesnt-match:



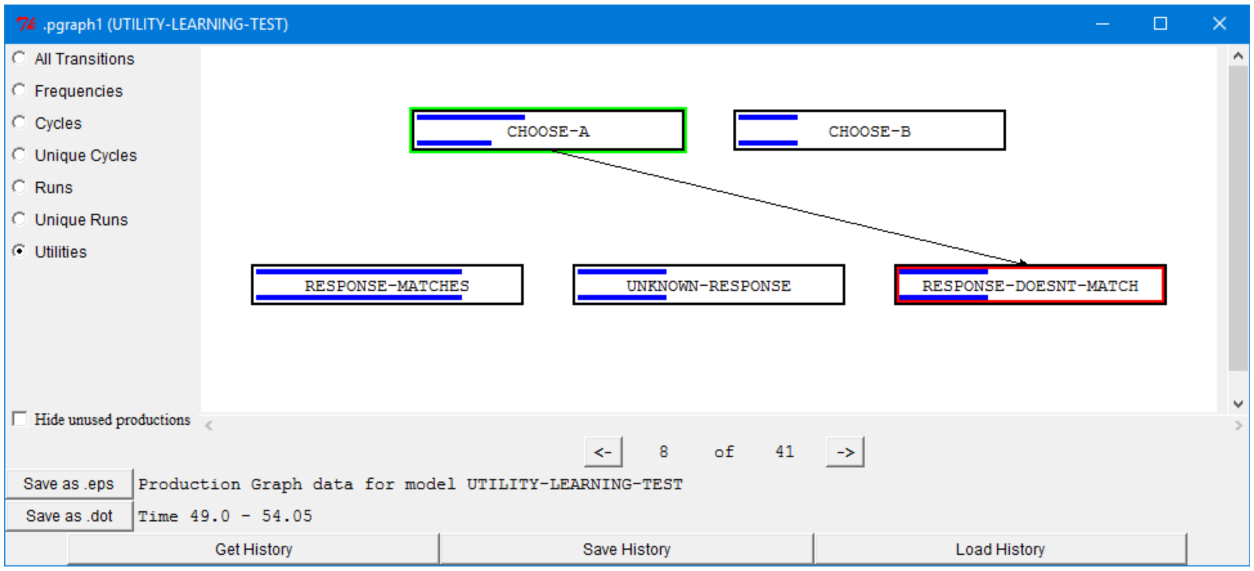
```
Production Graph data for model UTILITY-LEARNING-TEST
Time 14.0 - 19.05
```

The utility of choose-b clearly decreases, but response-doesnt-match appears to stay the same. The reason for that is because the starting utility of the productions is 0 and the reward provided by response-doesnt-match is also 0. However, it is important to note that there is still a very small change in the utility of response-doesnt-match as is shown in the utility traces displayed earlier:

```
...
    19.050   UTILITY                PROPAGATE-REWARD 0
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
```

At this time it has decreased from a utility of 0 to a utility of -0.01. That is because the effective reward for a production is the reward provided, in this case 0, minus the time since the production's selection. Since the reward is provided when that production fires, 50ms have passed since its selection and thus the effective reward it receives is -0.05 which is multiplied by the learning rate of .2. That change in utility from 0 to -0.01 is not visible in the graphic display for this task, but might be in other tasks since the changes shown are relative to the minimum and maximum utility values in the model data.

After that trial there are several which show choose-a being selected followed by response-matches and the utilities increasing. On the eighth trial we again find choose-a being selected, but this time it is followed by response-doesnt-match:



```
Production Graph data for model UTILITY-LEARNING-TEST
Time 49.0 - 54.05
```

There we see the utility of choose-a being decreased because of the incorrect guess and again, no noticeable change in response-doesnt-match.

That is the last of the utilities graphs we will describe in the text, and ends our analysis of this test model. Before going on however you may want to look at some more of the trials in the Utilities graph and perhaps experiment with the two production tools described to get a feel for how they may be useful. When you are done, you should then call the finished function to remove the new commands which were added to implement the task:

```
? (finished)
```

```
>>> ul_issues.finished()
```

Production Compilation

Models which use production compilation will almost always be using utility learning so that the newly learned productions are introduced gradually, and they will also usually involve declarative retrievals because compiling away a retrieval is one of the major benefits in compiled productions. Because of that, one will have to be sensitive to all the issues related to those mechanisms as described above and in the unit 5 text. A recommended practice when working with production compilation is to first make sure the model works as expected without turning on production compilation. That is because it will be easier to fix the basic operation of the model as well as any procedural and declarative learning issues without having to deal with newly learned productions as well. Once the model is working well at that level, then turn on production compilation and address any new issues which arise. Those new issues may still involve general utility or activation processes in addition to issues related to the learning of new productions, but having tested the model without production compilation should make it easier to locate and address the new issues. In this text we will focus specifically on preparation, testing, and debugging issues related to the production compilation aspects of an example model, but for other modeling tasks there may be other issues which will also have to be addressed.

The Task

The task the model will perform is similar to the choice and one hit blackjack tasks from previous units. Two numbers will be presented on the screen, each from 0-3, and then one of three choices must be made using the keys s, d, and f. After a key is hit, the result of that choice for the given pair of numbers will indicate whether the result was a win, loss, or draw. The spacebar must then be pressed to advance to the next trial. No information about the choices is provided in advance and the objective is to maximize the score (wins minus losses) based on the feedback provided while responding as quickly as possible. We do not have any data for the task to fit the model to, but we do expect the model to improve both its score and response time as it plays more games. We will look at the performance of the model over the course of 200 trials, averaged into blocks of 10.

To run the model through multiple 200 trial sessions and report the average results call the pcomp-issues-game function in Lisp or the game function from the pcomp_issues module in Python. It requires one parameter which is the number of games to run and average together. It also takes an optional parameter which if specified as true will print out the results of each of the individual sessions as it runs.

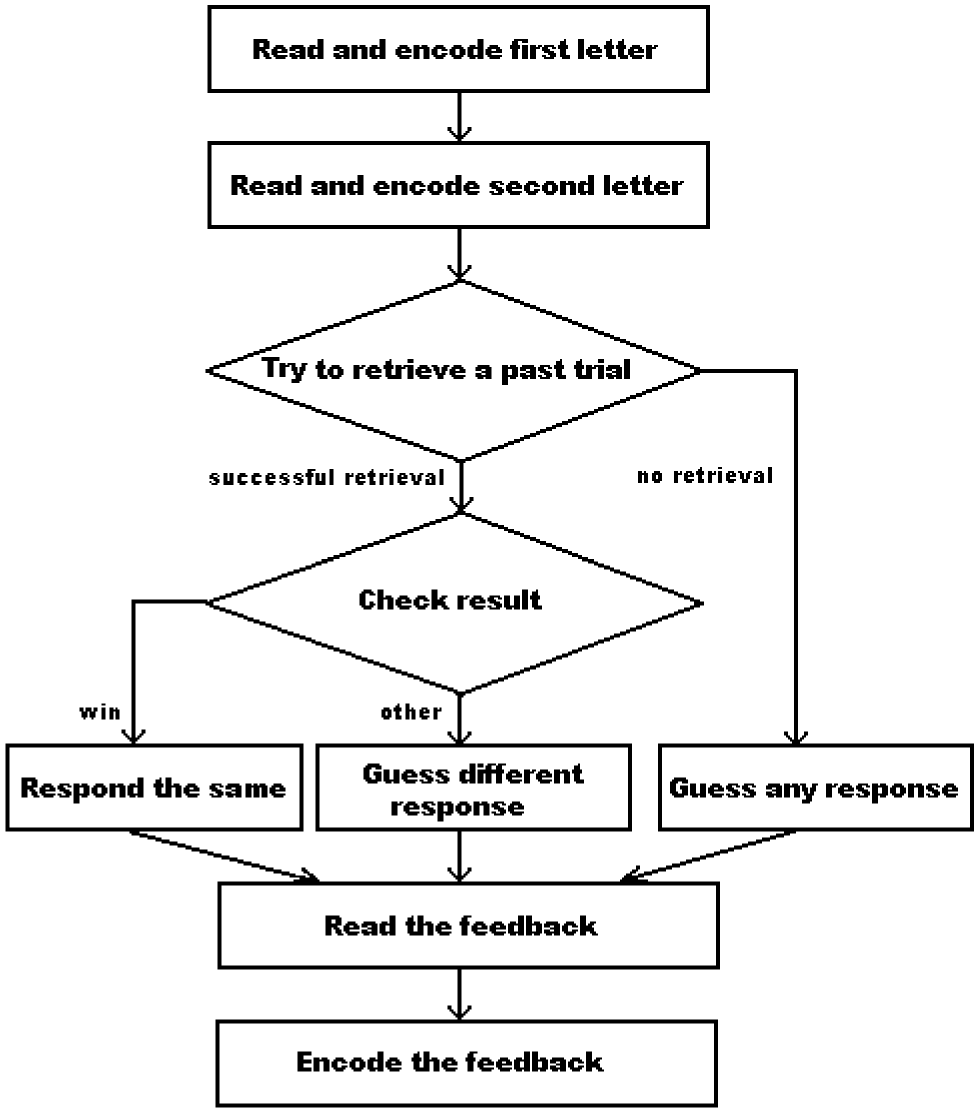
The model can also be run through fewer trials using the pcomp-issues-trials function in Lisp or the trials function from the pcomp_issues module in Python. It takes three optional parameters: the first specifies how many trials to run with the default being 200, the second indicates whether the model should be reset before the trial with the default being that it should, and the third indicates whether the scores and response times should be displayed for every 10 trials (which also defaults to being on).

The Starting Model

Before discussing anything related to production compilation, we will first describe a model which has been written to perform the task without production compilation. That model is found in the "production-compilation-issues-model.lisp" file. After that we will investigate what changes are necessary to effectively use that model with production compilation.

To model this task we have created a model which uses partial matching to retrieve a chunk stored in declarative memory from a previous trial that is similar to the current trial. This model is very similar to how the one hit blackjack model operated, and that is because this is a typical approach to use when a model must learn from experiences. For each trial of the task the model will create a chunk which includes the pair of numbers, the choice it made, and the result for that choice. Then when presented with a pair of numbers on another trial it will attempt to retrieve a chunk which indicates the winning move for the current pair and use the retrieved chunk to determine a response for this trial. The model has partial matching enabled so that it may be able to retrieve a chunk of a past trial even if this is the first time it experiences a given pair or if it has not yet found the winning move. The model also has base-level learning enabled so that the chunks which represent the trails will have their activations increased as it encounters and uses them more often which should result in a decrease in the response times over the experiment.

Here is a high-level flow chart representation of the steps which the model will be performing.



Many of those steps require multiple productions to perform, and you should be able to read through the productions in the model and follow how it works at this point. We will not describe the productions here, but we will provide some details on how the model represents the information it uses to perform the task.

Here are the chunk-types which the model specifies:

```
(chunk-type task state)
(chunk-type number visual-rep)
(chunk-type response key (is-response t))
(chunk-type trial num1 num2 result response)
```

The task chunk-type is used for the **goal** buffer to keep an explicit state marker for sequencing through the task. As has been stated in other units, doing that is not always necessary, but has been done here to make the model easier to read and follow.

The number chunk-type is needed to encode the numbers which the model will be using in its representation of the trials. That is necessary so that they can have similarities set between them. The number chunks include a slot for the visual representation so that the model can retrieve a number chunk based on the value which it gets when it attends to it on the screen. Here are the initial number chunks which the model starts with in its declarative memory:

```
  (add-dm (zero isa number visual-rep "0")
          (one isa number visual-rep "1")
          (two isa number visual-rep "2")
          (three isa number visual-rep "3"))
```

The base-level of those chunks is set to a high value so that they should always be retrieved quickly. There are also similarity values set between those items using a simple linear function based on their differences.

The response chunk-type is used to represent the possible choices which the model can make in the task. It has a slot which holds the representation of the key needed to make the **manual** response and a slot with a default value of t which is there to make it easy to retrieve a random response in the model by just indicating “isa response” in the request. Here are the chunks which the model starts with in its declarative memory:

```
  (add-dm (response-1 isa response key "s")
          (response-2 isa response key "d")
          (response-3 isa response key "f"))
```

Like the number chunks, those chunks are given a high base-level activation as an assumption that the model knows the instructions before starting the task.

The trial chunk-type is used to create the representation of a trial as the model performs the task. The num1 and num2 slots will contain number chunks for the trial presented. The result slot will contain one of the chunks: win, lose, or draw, and the response slot will contain the response chunk used on that trial. Here is an example of what such a chunk might look like:

```
CHUNK0-0
   RESULT  LOSE
   NUM1  ONE
   NUM2  ZERO
   RESPONSE  RESPONSE-2
```

Like the numbers, the result is encoded as a chunk so that similarities can be set between the choices. That way when the model attempts to retrieve a win, it may still be able to retrieve a draw or lose result for the trial. Since the model will not need to retrieve those result values, to keep the model simpler, they are encoded explicitly by productions instead of providing chunks in declarative memory and requiring a retrieval for encoding.

If you look at the similarity settings in the model between the result chunks you may find it curious that win is set to be more similar to lose than it is to draw. The reason for that is because if the model cannot retrieve a winning move, then retrieving a losing move is strategically better than retrieving a move which resulted in a draw for improving the score. Thus, the similarities are being used in this case to represent the usefulness of the information as an abstraction for a more deliberate strategy process in the model. That simplification is reasonable for this demonstration task since we are only concerned about showing learning through practice, but a more thorough model of a real task like this may require the model to account for that strategy processing.

Here are the parameter settings from the model:

```
  (sgp :esc t :lf .5 :bll .5 :mp 18 :rt -3 :ans .25)
```

Since we do not have data to fit, the parameters for the model were either set to recommended values (:bll and :ans) or simply adjusted to values which resulted in showing improvements which seemed reasonable for the demonstration.

Here are the results for the model on the task averaged over 50 runs:

```
Average Score of 50 trials
2.44 4.96 6.34 6.76 7.28 7.40 7.72 8.04 7.82 8.02 8.00 8.42 8.30 8.76 8.18 8.44 8.32 8.54 8.80 8.44
Average Response times
7.87 4.79 3.13 2.41 2.12 1.86 1.73 1.59 1.52 1.48 1.42 1.34 1.30 1.24 1.25 1.23 1.24 1.19 1.18 1.16
```

It is improving its score and getting faster over trials, which is what we expect. You may want to step through the model and perhaps explore its operation with the history and graphing tools before continuing so that you have a good understanding of how it operates.

Considerations for Production Compilation

When using production compilation, there are some things that should be considered with respect to the model for production compilation to work well. If the model was written with those considerations in mind, then the next step would be to turn production compilation on and start testing. However, if the model was not written for use with production compilation, which is the case for the starting model we described above, then just turning it on "to see what happens" is usually not going to work very well. For example, here is what happens if we run the starting model with production compilation enabled and no other changes:

```
Average Score of 50 trials
1.50 1.90 2.40 2.58 3.02 2.66 3.54 3.10 3.78 4.00 3.76 3.56 3.58 3.40 4.28 4.04 4.26 4.08 4.12 3.98
Average Response times
8.57 6.37 5.45 5.10 4.45 4.53 3.74 3.80 3.59 3.45 3.38 3.20 3.32 3.24 3.03 2.92 2.74 2.85 2.75 2.71
```

Neither the scores nor the response times look very good relative to how it ran previously. That might suggest that one should then start tracing, testing, and debugging the model, but the recommendation would be to first consider the following issues before attempting to run it with production compilation.

What is the task and how is the model run

Production compilation requires repetition to be effective because it will only show a change if the model has the opportunity to use the newly learned productions. Thus, the task must be one in which the model will be running repeatedly without being reset. Models which are already using base-level learning or utility learning will likely have that characteristic already. Other tasks however may not, for example the fan effect model and the subitizing models from the tutorial do not. Those models are being reset for each trial, and thus production compilation would not show any change in the results which they produce. If one wanted to use models like that with production compilation they would have to be changed so that they run continuously over the trials instead. The other thing to consider is whether there is enough repetition in the task to be effective. If one is looking for declarative knowledge to become encapsulated in the productions there will typically need to be multiple usages of those chunks so that the productions can be strengthened to the point of competing with the originals. For example, even if the fan experiment model were to be changed to present the trials continuously, since each test sentence is only presented once to that model, there would probably not be any use of the productions which proceduralize the declarative information. If the task is not continuous and/or does not provide any repetition then there is little reason to enable production compilation since it will not affect the operation of the model.

Utility learning

One of the most important issues with respect to production compilation is utility learning. It is the learning of utilities for the new productions which leads to their gradual introduction and whether they will end up being used in place of the original productions. Without utility learning the new productions will only ever have their initial utility value. If the model has not set the initial utilities for the existing productions or changed the :nu and :iu parameters then a newly learned production will have the same utility, 0, as all other productions and immediately compete with them, regardless of whether that production is actually useful or not. For example, in this task, that might mean that a production which always makes a losing move may be competing equally with the productions which attempt to remember a past move.

If the starting model was already using utility learning then one will want to make sure that the newly learned productions will start out with lower utilities than the original ones. If the original productions have greater than zero utilities (either because they are explicitly set or because the :iu parameter was set to greater than zero) then no immediate change would be needed. If the original productions do start with zero or negative utilities then the :nu parameter, which controls where the newly learned productions’ utilities start, should be set to a negative value so that they are lower than the original productions’ utilities. In either case those initial utility values may need to be adjusted as one starts to test the model, but it helps to have a reasonable starting point.

If the original model did not use utility learning, which is true for the starting model we have here, then one will first have to add that to it. That means that in addition to enabling the mechanism one will have to add some rewards to the model so that it has opportunities for learning. The utility values for the initial productions and starting values for the newly learned productions will also have to be set so that the new productions start below the originals (as described above).

When enabling utility learning for a model which will be using production compilation one will also want to make sure that there is some utility noise in the system so that the newly learned productions will have a chance to be selected. If there is no noise then the new productions will never exceed the utility of their parents (assuming a recommended utility learning rate of less than 1.0) and thus will never be selected. The amount of utility noise will affect the rate at which the new productions get used (how many times they will need to be recreated before they have utilities with a reasonable probability of being selected) since the noise affects the probability of selecting the productions as shown in the equation from unit 6. Assuming that one wants the productions to be introduced gradually, a low value for the noise is recommended, but what exactly constitutes a "low" value will depend on the relative utilities and the learning rate in the model.

Expected Changes

Another thing to consider for a model is what production compilation may change about the way that it operates. There are two very general things that production compilation can do: reduce sequences of production firings into fewer productions and transition knowledge from a declarative representation into a procedural one. Those can combine to produce interesting results, like the over generalization that occurs in the past-tense model, but particular effects like that usually require careful planning in the design of the model. As a first step, particularly for a model which may not have been specifically designed for production compilation, just considering the potential changes production compilation may have can be helpful before trying to use it.

If the model is being designed from the start to utilize production compilation, then knowing what effects are desired will help to shape the initial creation of the model. When looking to get a decrease in the time the model takes because of a reduction of long production sequences one will likely want to start the model with productions which perform small steps so that there are opportunities for productions to be compiled together. One will also want to be careful about separating perceptual and motor tasks which will block the compilation of productions from those which are expected to be compiled together. If the proceduralization of declarative knowledge is desired, obviously one will first have to have a model which makes requests for declarative information. Then one will have to carefully consider the productions which request and harvest the **retrieval** buffer chunks. Those productions will need to be safe for compilation, and thus will need to avoid other actions like requesting and harvesting perceptual information or performing multiple motor actions since those cannot be combined through production compilation. In addition to that, one may want to consider the details of what information is used to make the requests and what is tested in the harvesting productions. Those details will shape how the compiled production works and are important when looking for particular results, like generalization.

If the model was not initially designed for production compilation, then one should look over the model with respect to the issues noted above to determine if compilation is going to be effective at performing the desired results. If a speed up from creating shorter production sequences is desired, then one will want to look at the productions and see if they seem amenable to compilation. Things to look for are whether the productions are already performing multiple actions which might prevent them being combined any further and whether or not the perceptual and motor actions are isolated or pervasive throughout the productions. If it does not look like there will be many opportunities for compilation to combine productions further then one may want to consider making some changes to provide those opportunities. That might involve breaking up existing productions to make the model slower initially so that production compilation can provide the speed up. It may also require creating productions specifically for the perceptual and motor actions so that they are separated from productions which can be compiled together. If the transition from declarative to procedural knowledge is desired, then, like above, one will want to look at the productions which request and harvest the declarative chunks to make sure that they can be composed.

Considering the starting model

With those concepts in mind, we will look at the task and starting model before enabling production compilation and running it again. Because the original task involved base-level learning the model already ran continuously over the trials. Also, the 200 trials provided enough repetition to show learning for the declarative information. So the task and model seem like they are functionally capable of working with production compilation.

Let us next consider what we expect production compilation to do for this model. Looking over the productions, this starting model has been written with productions which already combine multiple actions. In addition, there are only 10 productions fired to perform a trial of the task as it stands, and since many of those productions are involved with perceptual processes that will always be required there appears to be little opportunity for this model to improve performance from reducing long production sequences. If we were interested in fitting a particular gradual performance increase, then we may want to reconsider this as a starting model and perhaps simplify those productions or move to a model which uses a more general instruction following process to do the task, like the paired associate task from unit 7. For this example we will not make any changes to try to change that and just see if there are any gains in that respect as is. Transitioning the knowledge from declarative to procedural however does seem like something which would be desirable in this task. Instead of always having to retrieve a move from declarative memory we would like to see this model develop productions which are able to make a move directly. The productions which the model has for performing the critical retrievals are free of perceptual and motor actions (other than a final response). Therefore, it seems like it should be possible for this model to do that. We could look more closely at those productions now to make sure that they can safely be composed, but instead we will wait and let the production compilation mechanism itself indicate any problems it finds when we run it.

The last thing to consider is utility learning, and this starting model does not currently use it. Therefore we will need to add that to it before production compilation will be able to affect the operation of the model through a gradual introduction of newly learned productions. That will involve setting some general parameters as well as providing rewards to the model. Because the model's results did not depend on utility learning we will have to start by just setting some reasonable values, and then perhaps adjust them later once we enable production compilation and see how it performs. We will take a little time to walk through exactly how we will chose those initial values in the next few paragraphs.

Since the model already has three productions for processing the feedback, that seems like a good place to add rewards. To determine how much reward to provide, we will make some simple assumptions and go from there. If we assume that new productions will start at a utility of 0 (the default), we will want the initial productions to start somewhere above that. Another assumption that is usually a good one to make is that we do not want the initial productions to drop to a utility below where a newly created production starts since we do not want the newly learned productions to immediately be preferred. Since we are assuming that new productions start with a utility of 0, that means that the initial productions should always have positive utilities. To ensure that, we do not want productions to get negative effective rewards (the reward minus the time between the production selection and the reward being provided). Thus, the minimum reward we want to provide to the model will depend on the longest time we expect the model to take before getting a reward. That should happen on the first trial it does because that will result in a retrieval failure for a past game, which represents the maximum time a retrieval can take. To find that we will turn on the trace with the detail level set to low to see when the feedback production fires and run one trail (since the :seed parameter is not set in the starting model when you run it your trace will differ slightly from the one shown here):

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TASK0 NIL
     0.000   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED DETECT-TRIAL-START
     0.135   VISION                 SET-BUFFER-CHUNK VISUAL TEXT0
     0.185   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-1
     0.188   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   IMAGINAL               SET-BUFFER-CHUNK IMAGINAL CHUNK0
     0.300   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-1
     0.300   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
     0.350   PROCEDURAL             PRODUCTION-FIRED FIND-NUM-2
     0.435   VISION                 SET-BUFFER-CHUNK VISUAL TEXT1
     0.485   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-2
     0.489   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.539   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-2
     0.589   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-PAST-TRIAL
    10.632   DECLARATIVE            RETRIEVAL-FAILURE
    10.682   PROCEDURAL             PRODUCTION-FIRED NO-PAST-TRIAL
    10.684   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL RESPONSE-2
    10.734   PROCEDURAL             PRODUCTION-FIRED RESPOND
    10.734   MOTOR                  PRESS-KEY KEY d
    10.944   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2 NIL
    11.079   PROCEDURAL             PRODUCTION-FIRED DETECT-FEEDBACK
    11.164   VISION                 SET-BUFFER-CHUNK VISUAL TEXT2
    11.214   PROCEDURAL             PRODUCTION-FIRED ENCODE-FEEDBACK-LOSE
    11.214   MOTOR                  PRESS-KEY KEY SPACE
    11.214   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK1
    11.414   ------                 Stopped because no events left to process
```

It takes the model a little under 11 seconds to respond, and the feedback production fires at time 11.214 seconds. Therefore if we want all of the productions to receive positive rewards on all of the trials we will need to provide a reward greater than 11.214 in each case. Then, as long as the model responds, all of the productions will receive a positive reward and should not drop to a utility below 0.

We now need to decide exactly how much reward to provide for each result, and we will also need to consider the starting utility of the initial productions. What values to use can depend on many factors in a complex model, but in this case we will use the minimum reward value for a positive reward found above to provide some guidance. Thinking about the expected result, learning productions which respond without retrieving a past game, presumably we only really want to learn such productions for the responses which lead to wins, and not losses or draws. To achieve that we will want to have multiple reward values so that wins are favored over the others. Whether or not to favor a draw over a loss might matter for fitting real performance, but for this task we will assume that a draw is better than a loss. Thus, we will have three reward values provided to the model. Since we want all of the productions to receive positive rewards for completing the task, we will start by giving a loss a reward of 12. From there we will choose some larger values for a draw and a win. One could perform some analysis to determine values based on probability of being selected as a function of rewards, but since we do not exactly know how production compilation will affect this specific model we will just choose values of 15 and 18 for a draw and win respectively so that there is some distance between them and see how that works. Thus, here are the settings which we will add to the model:

```
  (spp encode-feedback-win :reward 18)
  (spp encode-feedback-lose :reward 12)
  (spp encode-feedback-draw :reward 15)
```

Now we need to choose the starting utility for the initial productions. Given the nature of the task and the rewards chosen already, starting with the initial productions having a utility equal to the reward given for a draw seems like a good place to start them. Then a win should result in increasing utilities while a loss will cause them to decrease.

The last thing we need to add is the noise. As with the rewards, we could try to determine a value analytically, but instead we will just pick a starting noise value of 1.0 and adjust it later if we notice any issues. We will leave the learning rate, alpha, at its default of .2. So, here are the settings which we need to add to the model now to enable utility learning and set those parameters:

```
(sgp :ul t :egs 1.0 :iu 15)
```

Those changes should not affect the operation of the current model since it does not have any productions which are currently competing for selection based on utility. If we run a few trials to check it still seems to be performing as before:

```
Average Score of 10 trials
1.70 6.20 7.80 7.90 7.50 8.50 8.30 9.10 9.50 8.60 9.40 9.10 8.70 8.80 9.30 9.30 8.90 9.40 8.90 9.40
Average Response times
8.78 5.02 3.38 2.40 2.06 1.88 1.51 1.50 1.30 1.33 1.28 1.25 1.27 1.24 1.21 1.23 1.18 1.20 1.18 1.15
```

You may want to inspect that in more detail using the Environment tools as described for the first model above to verify that it is always receiving a reward and to see how the utilities are changing, even though they are not affecting the operation of this model.

Now that we have inspected the model and made the changes that were necessary for production compilation to work well it is time to enable production compilation and start testing. To enable production compilation all we need to do is set the parameter :epl to t, but we are also going to turn on the additional trace output it provides so that we can see what it does as the model runs. So we will add this additional setting to the model:

```
(sgp :epl t :pct t)
```

Testing the Model

Testing a model which uses production compilation typically involves four phases. The first is making sure the model performs as expected without production compilation being turned on. After that, production compilation is turned on and one runs the model watching the productions which are generated by production compilation. The objective here is to verify that things are working well at the symbolic level. You want to make sure that production compilation is able to compose the starting productions into new productions, and that those new productions appear to be doing the things you expect. Once it looks like production compilation is producing reasonable new productions you want to make sure that those new productions are not going to cause problems for the model's operation. If the model is small and does not require a long time to run, then it may be sufficient to just run it for multiple trials and monitor its operation, but for a large or very long running model it may be easier to temporarily adjust some of the model parameters so that the new productions are used right away so that their effects are easier to see. Finally, once you are comfortable with the productions generated through compilation and how they affect the model's basic operation you can then start to run the model for comparison to data and determining whether or not you get the overall results you were looking for and attempt to adjust the parameters as needed to fit your data. As with all testing and debugging, that is not always going to be a simple sequential process since one may have to go back and perform earlier tests again because of changes or problems which are encountered in a later step.

Since we have already tested the model without production compilation we will now turn on compilation and look at the productions it generates and the places where it cannot generate productions. To see that we will need to turn the model trace on, the :v parameter, in addition to the production compilation trace value we just added. Since we may also want to be able to repeat the same trials again it is a good idea to have the model print out the current seed when it gets reset so we can set that value again later if we want to run a trial again to analyze. To do that one would add this setting to the top of the model definition:

```
(sgp :seed)
```

However, so that your model runs correspond to those shown in the text we will be setting explicit seed values in the model for testing purposes. The first seed we will use is this one:

```
(sgp :seed (1 3))
```

Now we will run the model one trial at a time to look at the results of production compilation. It will require multiple trials before we are going to see the primary result we are expecting because the model will have to first learn a chunk which represents a trial and then be able to successfully retrieve it. We could adjust the parameters and add additional chunks to the model to artificially create the situations we are interested in seeing production compilation applied to, but since this is a fairly simple model that is not really necessary because we can easily investigate the situations occurring under the model's normal operation.

Here is what we get with the production compilation trace enabled for the first trial with the :trace-detail set to low:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TASK0 NIL
     0.000   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED DETECT-TRIAL-START
Production Compilation process started for DETECT-TRIAL-START
  No previous production to compose with.
  Setting previous production to DETECT-TRIAL-START.
     0.135   VISION                 SET-BUFFER-CHUNK VISUAL TEXT0
     0.185   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-1
Production Compilation process started for ATTEND-NUM-1
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-TRIAL-START and ATTEND-NUM-1 cannot be composed.
  Setting previous production to ATTEND-NUM-1.
     0.187   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.250   IMAGINAL               SET-BUFFER-CHUNK IMAGINAL CHUNK0
     0.300   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-1
Production Compilation process started for ENCODE-NUM-1
  Production ATTEND-NUM-1 and ENCODE-NUM-1 are being composed.
  New production:

(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - THREE"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "3"
 ==>
   =IMAGINAL>
       NUM1 THREE-0
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
Parameters for production PRODUCTION0:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-1.
     0.300   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
     0.350   PROCEDURAL             PRODUCTION-FIRED FIND-NUM-2
Production Compilation process started for FIND-NUM-2
  Buffer VISUAL-LOCATION prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production ENCODE-NUM-1 and FIND-NUM-2 cannot be composed.
  Setting previous production to FIND-NUM-2.
     0.435   VISION                 SET-BUFFER-CHUNK VISUAL TEXT1
     0.485   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-2
Production Compilation process started for ATTEND-NUM-2
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production FIND-NUM-2 and ATTEND-NUM-2 cannot be composed.
  Setting previous production to ATTEND-NUM-2.
     0.487   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.537   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-2
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  New production:

(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO-0
   =GOAL>
```

```
       STATE RETRIEVE-PAST-TRIAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
     0.587   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-PAST-TRIAL
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  New production:

(P PRODUCTION2
  "ENCODE-NUM-2 & RETRIEVE-PAST-TRIAL"
   =GOAL>
       STATE ENCODE-NUM-2
   =IMAGINAL>
       NUM1 =N1
   =RETRIEVAL>
 ==>
   =IMAGINAL>
       NUM2 =RETRIEVAL
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   +RETRIEVAL>
       NUM1 =N1
       NUM2 =RETRIEVAL
       RESULT WIN
)
Parameters for production PRODUCTION2:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.
    10.630   DECLARATIVE            RETRIEVAL-FAILURE
    10.680   PROCEDURAL             PRODUCTION-FIRED NO-PAST-TRIAL
Production Compilation process started for NO-PAST-TRIAL
  Cannot compile RETRIEVE-PAST-TRIAL and NO-PAST-TRIAL because the time between them exceeds the
threshold time.
  Setting previous production to NO-PAST-TRIAL.
    10.682   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL RESPONSE-2
    10.732   PROCEDURAL             PRODUCTION-FIRED RESPOND
Production Compilation process started for RESPOND
  Production NO-PAST-TRIAL and RESPOND are being composed.
  New production:

(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
```

```
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
Parameters for production PRODUCTION3:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
    10.732   MOTOR                  PRESS-KEY KEY d
    10.942   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2 NIL
    11.077   PROCEDURAL             PRODUCTION-FIRED DETECT-FEEDBACK
Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  New production:

(P PRODUCTION4
  "RESPOND & DETECT-FEEDBACK"
   =GOAL>
       STATE RESPOND
   =IMAGINAL>
   =RETRIEVAL>
       IS-RESPONSE T
       KEY =KEY
   =VISUAL-LOCATION>
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE =RETRIEVAL
   =GOAL>
       STATE ENCODE-FEEDBACK
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
       CMD PRESS-KEY
       KEY =KEY
)
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
    11.162   VISION                 SET-BUFFER-CHUNK VISUAL TEXT2
    11.212   PROCEDURAL             PRODUCTION-FIRED ENCODE-FEEDBACK-LOSE
Production Compilation process started for ENCODE-FEEDBACK-LOSE
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-FEEDBACK and ENCODE-FEEDBACK-LOSE cannot be composed.
  Setting previous production to ENCODE-FEEDBACK-LOSE.
    11.212   MOTOR                  PRESS-KEY KEY SPACE
    11.212   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK1
    11.412   ------                 Stopped because no events left to process
```

After every production fires production compilation attempts to create a new production, and for each attempt the production compilation trace provides the details of what the resulting new production looks like or a description of an issue which prevented it from compiling the productions. We will look at each one that occurred in this trace to make sure things are working as expected.

Here is the first production compilation trace message:

```
Production Compilation process started for DETECT-TRIAL-START
  No previous production to compose with.
  Setting previous production to DETECT-TRIAL-START.
```

Since that is the first production there is nothing to compose it with and thus all it can do is record that that production is now the previous one for use when the next one fires. The next result is this:

```
Production Compilation process started for ATTEND-NUM-1
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-TRIAL-START and ATTEND-NUM-1 cannot be composed.
  Setting previous production to ATTEND-NUM-1.
```

It indicates that the productions cannot be composed because the visual buffer blocks it due to the request and harvesting of a chunk. Since perceptual information cannot be compiled into new productions that is what we would expect and there is not anything we need to do to try to fix that.

The next result is somewhat unexpected:

```
Production Compilation process started for ENCODE-NUM-1
  Production ATTEND-NUM-1 and ENCODE-NUM-1 are being composed.
  New production:

(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - THREE"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "3"
 ==>
   =IMAGINAL>
       NUM1 THREE-0
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
Parameters for production PRODUCTION0:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-1.
```

Since the encoding step which the model performs requires retrieving the number chunk from declarative memory, production compilation is able to compose those two into a new production which does not require the retrieval. We did not really consider that in what we expected from the model, but it appears to be another opportunity for the model to get faster over time which is in line with what we want so having such a production does not seem to be a problem.

The next two production compilation attempts are unsuccessful because the productions involved are performing perceptual actions:

```
Production Compilation process started for FIND-NUM-2
  Buffer VISUAL-LOCATION prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production ENCODE-NUM-1 and FIND-NUM-2 cannot be composed.
  Setting previous production to FIND-NUM-2.

Production Compilation process started for ATTEND-NUM-2
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production FIND-NUM-2 and ATTEND-NUM-2 cannot be composed.
  Setting previous production to ATTEND-NUM-2.
```

After that is a production very similar to production0 this time encoding the second number into the imaginal chunk without having to perform the retrieval:

```
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  New production:

(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO-0
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
```

Again, this was not expected, but seems to be in line with the general expectations.

The next composition results in a production which is just the composition of two productions without removing an intervening retrieval:

```
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  New production:

(P PRODUCTION2
  "ENCODE-NUM-2 & RETRIEVE-PAST-TRIAL"
   =GOAL>
       STATE ENCODE-NUM-2
   =IMAGINAL>
       NUM1 =N1
   =RETRIEVAL>
 ==>
   =IMAGINAL>
       NUM2 =RETRIEVAL
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   +RETRIEVAL>
       NUM1 =N1
       NUM2 =RETRIEVAL
       RESULT WIN
)
Parameters for production PRODUCTION2:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.
```

This is another opportunity for the model to speed up over time, and also in line with the general expectation for the model.

Next, we see a failure to compose productions because of the amount of time that passed:

```
Production Compilation process started for NO-PAST-TRIAL
   Cannot compile RETRIEVE-PAST-TRIAL and NO-PAST-TRIAL because the time between them exceeds the threshold time.
  Setting previous production to NO-PAST-TRIAL.
```

The threshold time is a settable parameter in the model which we might what to consider adjusting, but since there was also a failure to retrieve a chunk those productions would not have been composable anyway. So, we will hold off on adjusting the parameter until we see whether or not successful retrievals are taking too long.

The next opportunity for composition results in a production which eliminates another retrieval:

```
Production Compilation process started for RESPOND
  Production NO-PAST-TRIAL and RESPOND are being composed.
  New production:

(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
Parameters for production PRODUCTION3:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

This production effectively results in guessing “d” when it cannot remember a past move. While that does save time by eliminating a production and a retrieval, it probably will not be a very useful production overall and we may never see it actually being used.

Despite the number of different conditions involved across various cognitive, perceptual, and motor modules the respond and detect-feedback productions are able to be composed:

```
Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  New production:

(P PRODUCTION4
  "RESPOND & DETECT-FEEDBACK"
   =GOAL>
       STATE RESPOND
   =IMAGINAL>
   =RETRIEVAL>
       IS-RESPONSE T
       KEY =KEY
   =VISUAL-LOCATION>
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE =RETRIEVAL
   =GOAL>
       STATE ENCODE-FEEDBACK
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
       CMD PRESS-KEY
       KEY =KEY
)
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
```

This seems like it might be yet another helpful production to save time doing the task, but a careful look at the conditions and actions with respect to what happens in the task will expose an issue with this production. This production gets a visual-location buffer test from detect-feedback and the manual action from respond. However, a chunk only enters the **visual-location** buffer because of buffer stuffing after the model makes a response which causes the feedback to appear. Thus, while there is nothing syntactically wrong with production4 it will never be able to match during this task since it has conditions which only result from actions it performs. That happens because production compilation has no way to detect dependencies which occur outside of the productions, in this case that the screen changes as a result of the key press, and thus it creates a production which will never be able to fire. Typically, that will not be problematic since a production which does not match has no effect on the model’s performance, but in some rare situations it may be necessary to explicitly indicate dependencies of that nature somehow in the production conditions to avoid the composition of productions which violate implicit task dependencies.

Here is the final opportunity for composition in this trial:

```
Production Compilation process started for ENCODE-FEEDBACK-LOSE
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-FEEDBACK and ENCODE-FEEDBACK-LOSE cannot be composed.
  Setting previous production to ENCODE-FEEDBACK-LOSE.
```

which fails because of the perceptual action involved.

Looking at the first trial produced a couple of unexpected compositions, but nothing which seems to violate what we want the model to do overall. Now we will run a couple more trials looking for compositions we have not seen yet, and in particular we want to see what happens when there is a successful retrieval of a past trial. We need to make sure to run those additional trials without resetting the model, thus we will need to specify the optional reset value as nil/False so as to not reset the model:

```
? (pcomp-issues-trials 1 nil)

>>> pcomp_issues.trials(1,False)
```

The second trail still does not result in a successful retrieval, but there are a few new production compilation attempts worth looking at. The first occurs immediately when the feedback encoding production of the previous trial gets composed with the detect-trial-start production:

```
Production Compilation process started for DETECT-TRIAL-START
  Production ENCODE-FEEDBACK-LOSE and DETECT-TRIAL-START are being composed.
  New production:

(P PRODUCTION5
  "ENCODE-FEEDBACK-LOSE & DETECT-TRIAL-START"
   =GOAL>
       STATE ENCODE-FEEDBACK
   =IMAGINAL>
   =VISUAL-LOCATION>
   =VISUAL>
       VALUE "lose"
   ?IMAGINAL>
       STATE FREE
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESULT LOSE
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
```

```
       CMD PRESS-KEY
       KEY SPACE
   +IMAGINAL>
   +GOAL>
       STATE ATTEND-NUM-1
)
Parameters for production PRODUCTION5:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward 15.000
 :fixed-utility    NIL
  Setting previous production to DETECT-TRIAL-START.
```

Like production4 from the first trial this production has a visual-location condition which will not be satisfied while doing this task because it comes about from the action which this production would make. Thus, this is another production which will never match and fire.

A couple of other new productions are also composed and they appear similar to those created on the first trial to compose the declarative information in the encoding phase into productions, which again seems reasonable.

Later in the run we see two occasions where production compilation recreates the same productions which it did in the first trial:

```
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  Recreating production PRODUCTION2
Parameters for production PRODUCTION2:
 :utility -1.959
 :u   2.451
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.

Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  Recreating production PRODUCTION4
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   2.859
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
```

In both those cases we see that the true utility (:u value) of those productions has now increased from 0, their initial value when first composed, since they get rewards based on the parent productions' utilities with each recreation.

There is however one curious composition given what we saw with the first trial:

```
(P PRODUCTION9
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
```

```
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-1
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
Parameters for production PRODUCTION9:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

On the first trial we also saw the composition of a production which collapsed no-past-trial with respond removing the retrieval of the chunk response-2:

```
(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
```

So the question is why on this trial is production9 created instead of just strengthening production3? If we look closely at those productions we can see that they differ slightly in the modifications that they perform to the chunk in the **imaginal** buffer:

```
   =IMAGINAL>
       RESPONSE RESPONSE-2-1
```

and

```
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
```

So, now the question is why do they differ like that? If we look at declarative memory we do not find either of those chunks. That probably means that they have been merged with other chunks. We can find that out using the pprint-chunks/pprint_chunks command to display them:

```
RESPONSE-2-0 (RESPONSE-2)
   KEY  "d"
   IS-RESPONSE  T
```

```
RESPONSE-2-1 (RESPONSE-2)
   KEY  "d"
   IS-RESPONSE  T
```

Both have been merged with the original chunk response-2, which does not seem to help explain why those are different productions. To answer that, we will have to look at where that action comes from in the original productions.

The modification to the imaginal chunk is an action from the respond production:

```
  (p respond
     =goal>
       isa task
       state respond
     =retrieval>
       isa response
       key =key
     ?manual>
       state free
     =imaginal>
       isa trial
     ==>
     =imaginal>
       response =retrieval
     +manual>
       cmd press-key
       key =key
     =goal>
       state detect-feedback)
```

In that production the slot is set to the chunk which is currently in the **retrieval** buffer. Recall that buffers hold copies of chunks. Thus, when the respond production fires the chunk in the **retrieval** buffer is not the chunk response-2 but a copy of it and since the buffer has not yet been cleared (that happens after respond fires) that copy in the buffer has not yet been merged with response-3. Production compilation does not know anything about what will happen to chunks in the future when it uses them in composing a production. Therefore, every time production compilation combines those two productions the chunk in the **retrieval** buffer will always be a new chunk and since that chunk is used to set the response slot of the **imaginal** buffer it must create a new production each time.

That may seem like a flaw with production compilation, but since it is not plausible for the mechanism to know the future that is all it can do. Therefore the flaw is really in the model design – specifically the representation of the knowledge it is using. It is the content of chunks which should be meaningful to the model, not their particular identity. While it is often convenient to refer to chunks by name like that in a model, there are situations where such shortcuts are inappropriate and should be avoided. There are a lot of ways that this model could be changed to not use the identity of the retrieved chunk directly, but since having those separate productions from this composition should not affect what we expect from the model we are not going to make any of those changes right now. However, if we encounter any other similar issues we will reconsider changing the model.

Since we still have not seen a successful retrieval we will run the model for a few more trials until we get one. On the fifth trial the model successfully retrieves a past trial chunk:

```
    48.372   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL CHUNK6-0
    48.422   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Production RETRIEVED-A-WIN is not valid for compilation
   because it has an indirect action with the RETRIEVAL buffer
```

Unfortunately, production compilation tells us that the retrieved-a-win production is invalid for compilation purposes because it makes an indirect retrieval action. Here is that production:

```
  (p retrieved-a-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
       result win
       response =response
     ==>
     +retrieval> =response
     =goal>
       state respond)
```

The value from the response slot of the trial chunk is being used to specify the retrieval, which is done as a consequence of a specific chunk reference being stored in that slot as was discussed for the respond production above. Since composing retrieve-past-trial and retrieved-a-win is something that we want the model to do we are going to have to change the representation stored in the response slot of the trial chunks and the productions which use them.

There are many ways which we could change the model, but because the model has such a simple representation for the response chunks we will start by making a small change and see how that affects things. The change that we will make is that instead of storing a response chunk itself in the response slot of the trial chunk we will store the value from the key slot of a response chunk in the response slot of the trial chunk. If the response chunks had contained more slots, then this simple change may not have been possible and a more thorough analysis of the model and its representations would have been required to determine how to request and harvest the chunks needed so that they would be compatible with production compilation.

Making that change requires changing three productions. The respond production needs to be changed to save the key slot's value instead of the response chunk itself:

```
  (p respond
     =goal>
       isa task
       state respond
     =retrieval>
       isa response
       key =key
     ?manual>
       state free
     =imaginal>
       isa trial
     ==>
     =imaginal>
       response =key
     +manual>
       cmd press-key
       key =key
     =goal>
       state detect-feedback)
```

Then the retrieved-a-win and retrieved-a-non-win productions need to be changed so that they retrieve a response chunk based on the key value instead of indirectly retrieving the chunk:

```
  (p retrieved-a-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
       result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state respond)

  (p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

After making that change, we should look at the model to make sure that there are not any other changes that should be made while we are adjusting it since we are going to have to retest it without production compilation before continuing to make sure it still works and making other changes now may save us from having to come back and test it without compilation yet again later.

One thing to notice is that since we now have the key to press in the trial chunks the model does not have to retrieve the response chunk in retrieved-a-win to be able to perform the key press. Similarly, retrieved-a-non-win does not need to retrieve the current response either since it could just retrieve a different response the way that guess-other does now and guess-other could be eliminated from the model. If we were not using production compilation those might be useful changes to make to the model, but production compilation should eliminate those retrievals from the model over time anyway so for now we will not make those changes to the model.

Looking at the encoding productions, encode-num-1 and encode-num-2, we see that the number chunks are also referenced by name for the trial encoding. If we go back and look at our first run with compilation turned on we can see that production0 and production1 which the model learned in fact also have references to specific chunks, three-0 and zero-0 respectively, and as we saw with production3 that means it is not going to be able to recreate and strengthen those productions. If we want to see the model response times decrease through eliminating the retrievals in that portion of the task we are also going to have to change how the model encodes the number chunks. In this case we need to have chunks in the num1 and num2 slots of the trial so that the similarities between those slot contents and the requested values will allow the model to retrieve a "close" trial chunk through partial matching when it does not have a perfectly matching trial chunk to retrieve. Thus, we cannot use the same change we did with the response chunks and just use the value of the visual-rep slot from the numbers in the trial chunks. Unlike the response chunks however the model will not need to retrieve the number chunks using the value from the slots of the trial chunk. Therefore we will not have the problem of a direct retrieval being necessary and all we need to do is provide a way for the model to reference the number chunks during the initial encoding without using the name of the chunk currently in the **retrieval** buffer.

That means that we will need to add an additional slot to the number chunk-type to hold the reference we want to use. We will call that slot representation and make this change to the chunk-type specification in the model:

```
(chunk-type number visual-rep representation)
```

That will then require making the following changes to the encoding productions to use that slot's value instead of the chunk in the **retrieval** buffer:

```
 (p encode-num-1
     =goal>
       isa task
       state encode-num-1
     =retrieval>
       isa number
       representation =number
     =imaginal>
       isa trial
     ==>
     =imaginal>
       num1 =number
     =goal>
       state find-num-2
     +visual-location>
       isa visual-location
       > screen-x current
       :attended nil)

 (p encode-num-2
     =goal>
       isa task
       state encode-num-2
     =retrieval>
       isa number
       representation =number
     =imaginal>
       isa trial
     ==>
     =imaginal>
       num2 =number
     =goal>
       state retrieve-past-trial)
```

Now we have to determine what value to store in that slot. It has to be a chunk so that similarities can be set, and there are basically two ways to handle that. One is to simply store the name of the number chunk itself in the representation slot when it is created. That would look like this in the current model:

```
(add-dm (zero isa number visual-rep "0" representation zero)
        (one isa number visual-rep "1" representation one)
        (two isa number visual-rep "2" representation two)
        (three isa number visual-rep "3" representation three))
```

The other option would be to create a more distributed representation which involves separate chunks for the visual mapping and the number itself. That might look something like this in the current model (though there are many ways to accomplish that):

```
(chunk-type number value)
(chunk-type number-visual visual-rep representation)

(add-dm (zero isa number value 0)
        (one isa number value 1)
        (two isa number value 2)
        (three isa number value 3)
        (isa number-visual visual-rep "0" representation zero)
        (isa number-visual visual-rep "1" representation one)
        (isa number-visual visual-rep "2" representation two)
        (isa number-visual visual-rep "3" representation three))
```

Note that for the number-visual chunks that perform the mapping from the visual representation to a number above there are no chunk names specified. The chunk name is optional when creating chunks and if one is not provided the system will generate a new name automatically. That reinforces the notion that the name of those chunks does not matter and only the content is important, but the downside to doing that is that it may make debugging the model more difficult since there will not be easily recognizable names in the trace or other inspection tools.

Which mechanism one chooses to use will depend on exactly what is required in the model and how one believes people encode that information. For this task we will go with the simpler single chunk representation, but you are welcome to try other alternatives and investigate the results as an additional exercise.

After making those changes, but before trying production compilation again, we should run the model without it to make sure that it still performs the task correctly. We need to remove the parameter setting which enables production compilation and also remove the seed value so that we can test it over multiple trials. Here are the results from the updated model:

```
Average Score of 10 trials
3.60 4.90 6.70 7.40 8.10 8.10 8.50 8.00 8.60 8.50 8.10 7.80 8.70 8.90 8.80 9.50 9.30 8.80 9.10 8.90
Average Response times
7.77 4.68 2.87 2.39 2.11 1.69 1.54 1.48 1.41 1.32 1.32 1.31 1.27 1.25 1.20 1.20 1.16 1.19 1.13 1.19
```

It still appears to be learning both with respect to increasing scores and decreasing response times. So we will re-enable production compilation, set the seed parameter again (so that we can recreate any issues which occur), and run it to see what happens with production compilation now. We will not include all of the trace here, but will include the details for important sections related both to the issues discussed above and any new issues which arise.

Looking at the productions learned during the initial encoding steps, like production0 and production1, we now see that they contain references to the number chunks themselves instead of the copy in the **retrieval** buffer when modifying the **imaginal** buffer:

```
(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
```

In addition to that, on a later trial we see a production combining attende-num-2 and encode-num-2 that has been recreated and strengthened:

```
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  Recreating production PRODUCTION7
Parameters for production PRODUCTION7:
 :utility  1.315
 :u   1.999
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
```

Thus, those changes to the model seem to have achieved their desired effects. Similarly, we now see production3 looking like this:

```
(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE "d"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
```

and on the second trial we see that it is now also recreated:

```
Production Compilation process started for RESPOND
```

```
  Production NO-PAST-TRIAL and RESPOND are being composed.
  Recreating production PRODUCTION3
Parameters for production PRODUCTION3:
 :utility  2.376
 :u   2.857
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

Running the model until we see it successfully retrieve a past trial shows the following in the trace:

```
    38.932   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL CHUNK4-0
    38.982   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Cannot compile RETRIEVE-PAST-TRIAL and RETRIEVED-A-WIN because the time between them
exceeds the threshold time.
  Setting previous production to RETRIEVED-A-WIN.
```

Previously we saw that retrieved-a-win was not valid for compilation, but now it is saying that the threshold time has been exceeded. That means the production is valid for composition, but too much time passed between the previous production's firing and the firing of this production. That happened because the retrieval took around 5 seconds to complete. Whether or not this is a problem, like many such issues, depends on one's hypothesis for what is happening when people learn in such tasks, and there are potentially multiple issues involved here. The first is whether or not one considers a 5 second retrieval to be reasonable for this task. If not, then one may want to adjust the declarative memory parameters to change that. Without data for comparison we are going to just assume that that retrieval is acceptable. Then, if one assumes that the retrieval time is acceptable, the next issue is whether one believes that the declarative knowledge must be strengthened prior to its being composed into procedural knowledge (have an activation value sufficient for it to be retrieved within the compilation threshold time) or whether production compilation should start compiling the knowledge immediately. The default setting for the production compilation threshold time is three seconds, but that value is just a conservative starting point for the system and not a strongly recommended value. For the purpose of this exercise we are going to adjust the threshold time parameter so that compilation can occur right away. To do that we must change the value of the :tt parameter to something larger than 5.224 (since that is how long the retrieval takes), and as a first pass we will choose 6 so that this pair of productions will fire. Thus, we will add this additional parameter setting to the model:

(sgp :epl t :pct t :tt 6)

After saving and reloading the model now when it gets to that point we see that it creates this production:

```
    38.982   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Production RETRIEVE-PAST-TRIAL and RETRIEVED-A-WIN are being composed.
  New production:

(P PRODUCTION21
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-WIN - CHUNK4-0"
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 ONE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
Parameters for production PRODUCTION21:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVED-A-WIN.
```

That production makes the request for a particular response, "s", based on testing the specific values encoded in the trial chunk without needing to retrieve a similar trial. That is what we want to see the model do. So, now all we need to do for verifying what happens symbolically in the model is see what happens when the model retrieves a non-winning past trial. However, after running many more trials that production still does not show up in the trace as being selected and fired.

One option would be to just ignore it since it did not fire and move on to testing the model over the whole task, but perhaps it did not fire because of the particular seed value we have set for the pseudo-random number generator. We want the model to work without requiring any particular seed value being set, and that production seems like it should fire sometimes. So, before moving on we will do some more tests to see if that production ever does fire, and if so what the results from production compilation are.

One way to test this would be to just remove the seed setting and then run the model repeatedly looking at the trace each time until we find one where it fires (we would probably also want to display the starting seed each time as was shown in the unit 3 modeling text so that we can recreate the trial once we find it). In some situations doing things that way might be acceptable, but it can be a very tedious process and might not be feasible in all situations. Something that can be useful to take advantage of is that we can use the !eval! operator in the productions to actually set a flag for us so that we can write some code that will run it until that flag is set.

There are many ways one could do that, and if one is running things from the ACT-R prompt it can be a little easier since you could put Lisp code directly into that !eval!, but for consistency with what we saw earlier in the tutorial and to keep the approaches similar between the Lisp and Python implementations we will add a new ACT-R command to set the flag.

To find a game in which that production fires we will add a !eval! action like this to the production to call a command called set-flag:

```
(p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     !eval! ("set-flag")
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

We also need to remove the :seed parameter setting from the model and just add a check of the parameter at the top of the model:

```
    (sgp :seed)
```

We also want to make sure that the :v parameter is set to nil to disable the trace. Then we need to add the set-flag command and write some code to actually run the model and test that result to find a game when it happens. Here is some Lisp code evaluated at the prompt to do so:

```
? (defvar *used* nil)
*USED*
? (defun set-flag () (setf *used* t))
SET-FLAG
?  (add-act-r-command  "set-flag"  'set-flag  "Command  for  testing  the  pcomp-issues
retrieved-a-non-win production.")
T
"set-flag"
? (while (null *used*)
    (pcomp-issues-game 1))
```

Here is some similar Python code to do the same thing:

```
>>> used = False
>>> def set_flag ():
...   global used
...   used = True
...
>>> actr.add_command("set-flag",set_flag,"Command for testing the pcomp-issues re
trieved-a-non-win production.")
True
>>> while used == False:
...   pcomp_issues.game(1)
...
```

When you use either of those you will see some output like this that prints the seed value and then the model results repeatedly until a game is found where it gets used:

```
:SEED (79246602991 0) (default NO-DEFAULT) : Current seed of the random number g
enerator
Average Score of 1 trials
3.00 9.00 8.00 6.00 8.00 6.00 7.00 7.00 7.00 9.00 9.00 9.00 8.00 9.00 7.00 6.00 10.00 6.00 10.00 10.00
Average Response times
9.72 3.92 1.83 2.03 1.54 1.73 1.44 1.47 1.43 1.17 1.34 1.24 1.19 1.11 1.28 1.33 1.01 1.29 1.01 1.03
:SEED (79246602991 9581) (default NO-DEFAULT) : Current seed of the random number generator
Average Score of 1 trials
5.00 5.00 8.00 4.00 8.00 8.00 8.00 4.00 10.0 10.0 10.0 10.0 10.0 10.0 9.00 10.0 10.0 10.0 8.00 10.0
Average Response times
8.09 2.67 1.78 1.44 1.80 1.66 1.57 1.31 1.35 1.16 1.17 1.09 1.02 1.27 1.10 1.09 1.11 1.08 1.00 0.95
...
:SEED (79246602991 77656) (default NO-DEFAULT) : Current seed of the random number generator
Average Score of 1 trials
```

```
-3.00 8.00 10.0 9.00 9.00 10.0 10.0 10.0 9.00 9.00 10.0 8.00 10.00 9.00 10.00 10.00 10.00 9.00 10.00 7.00
Average Response times
8.64 3.42 1.99 2.13 1.60 1.27 1.13 1.31 1.28 1.20 1.10 1.28 1.01 1.06 1.14 0.99 1.03 1.10 0.99 1.06
```

If you try that you will see different seed values displayed, but eventually it should stop and the last seed value shown will result in a game where that production fires. Before looking at the trace of that trail we will first remove that !eval! from the production because that will cause problems for production compilation with a warning like this:

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVED-A-NON-WIN is not valid for compilation
   because it contains one or more !eval! operators
```

Since the procedural system does not know what happens because of that external call through !eval! it considers it unsafe to compose that production.

We then need to set the seed to (79246602991 77656) since that is the value we found above and turn the trace back on. Then we will run it a trial at a time to find where that production fires.

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVED-A-NON-WIN is not valid for compilation
   because it has conditions with modifiers on slot tests
```

It indicates that the production is not valid for compilation because it has modifiers on the slot tests. Here is the production again for reference:

```
  (p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

The slot test highlighted above is the only one that has a modifier so that must be what is stopping compilation.

That is an important issue to keep in mind when working with production compilation -- it cannot compile productions which have tests for inequalities for reasons similar to not composing across a retrieval failure – one does not generally want to encode that something was not true and assume that will always be the case. However it is often convenient to have such tests in the productions which one wants to be compiled. There are a couple of ways to handle that. The first is to replace the production with one or more productions that perform the same calculation using a positive test. In this case that would mean adding retrieved-a-lose and retrieved-a-draw productions which test for those values explicitly as retrieved-a-win does. Since there are only three possible options that would not be a difficult change to make for the model, but in other situations that might not be feasible because there may be too many choices or not all the possibilities may be known in advance. An alternative, which we will use here, is to just bind the value from the slot to a variable in the buffer test and then perform the inequality test in code. Although we noted above that a !eval! blocks the composition, there is a special version which allows one to tell the procedural system that you are guaranteeing the external code to be “safe” with respect to production compilation. To do that you can use the !safe-eval! operator instead. That could go out through an external command, as we used above, but for simplicity we are just going to perform the check directly in Lisp code in the production:

```
  (p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
       result =result
       response =response
     !safe-eval! (not (equal =result 'win))
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

If we save that change and run the model to that point again we will now see the following production compilation trace result:

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVE-PAST-TRIAL and RETRIEVED-A-NON-WIN are being composed.
  New production:

(P PRODUCTION45
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-NON-WIN - CHUNK12-0"
   !SAFE-EVAL! (NOT (EQUAL (QUOTE LOSE) (QUOTE WIN)))
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 THREE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE GUESS-OTHER
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
Parameters for production PRODUCTION45:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVED-A-NON-WIN.
```

This time it created a production which will retrieve the response for "s" whenever it has encoded a trial of the numbers three and three. The !safe-eval! from the retrieved-a-non-win production has been included in the conditions of this production, but because the retrieval chunk's contents were compiled into the production the test is now explicitly testing that the symbol lose is not equal to the symbol win which will always be true. Unlike the compilation of retrieve-past-trial and retrieved-a-win however this production is not actually mapping a specific trial to a particular result because the production which fires after retrieved-a-non-win, guess-other, will retrieve a different response to make since the model does not want to make the response that did not lead to a win:

```
(p guess-other
     =goal>
       isa task
       state guess-other
     =retrieval>
       isa response
       key =key
     ==>
     +retrieval>
       isa response
      - key =key
     =goal>
       state respond)
```

Therefore when it retrieves a non-winning trial it is not going to immediately create a new production which performs a specific move. Since retrieved-a-non-win does not seem to fire very often (we had to search to find a game in which it did) that is not likely to be an issue in the model, but it is worth keeping in mind for any analysis we do later.

Before moving on to looking at the performance there is one last detail to mention. The guess-other production shown above includes a negative modifier in its request to the **retrieval** buffer so that it will retrieve a response which does not match the current one. Unlike inequality tests in the conditions however modifiers in a request are allowed for production compilation and we see this production as the result of production compilation for retrieved-a-non-win and guess-other in the trace and it keeps the modifier in the request:

```
Production Compilation process started for GUESS-OTHER
  Production RETRIEVED-A-NON-WIN and GUESS-OTHER are being composed.
  New production:

(P PRODUCTION46
  "RETRIEVED-A-NON-WIN & GUESS-OTHER - RESPONSE-1"
   !SAFE-EVAL! (NOT (EQUAL =RESULT (QUOTE WIN)))
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =RETRIEVAL>
       RESPONSE "s"
       RESULT =RESULT
 ==>
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
    -  KEY "s"
       IS-RESPONSE T
)
```

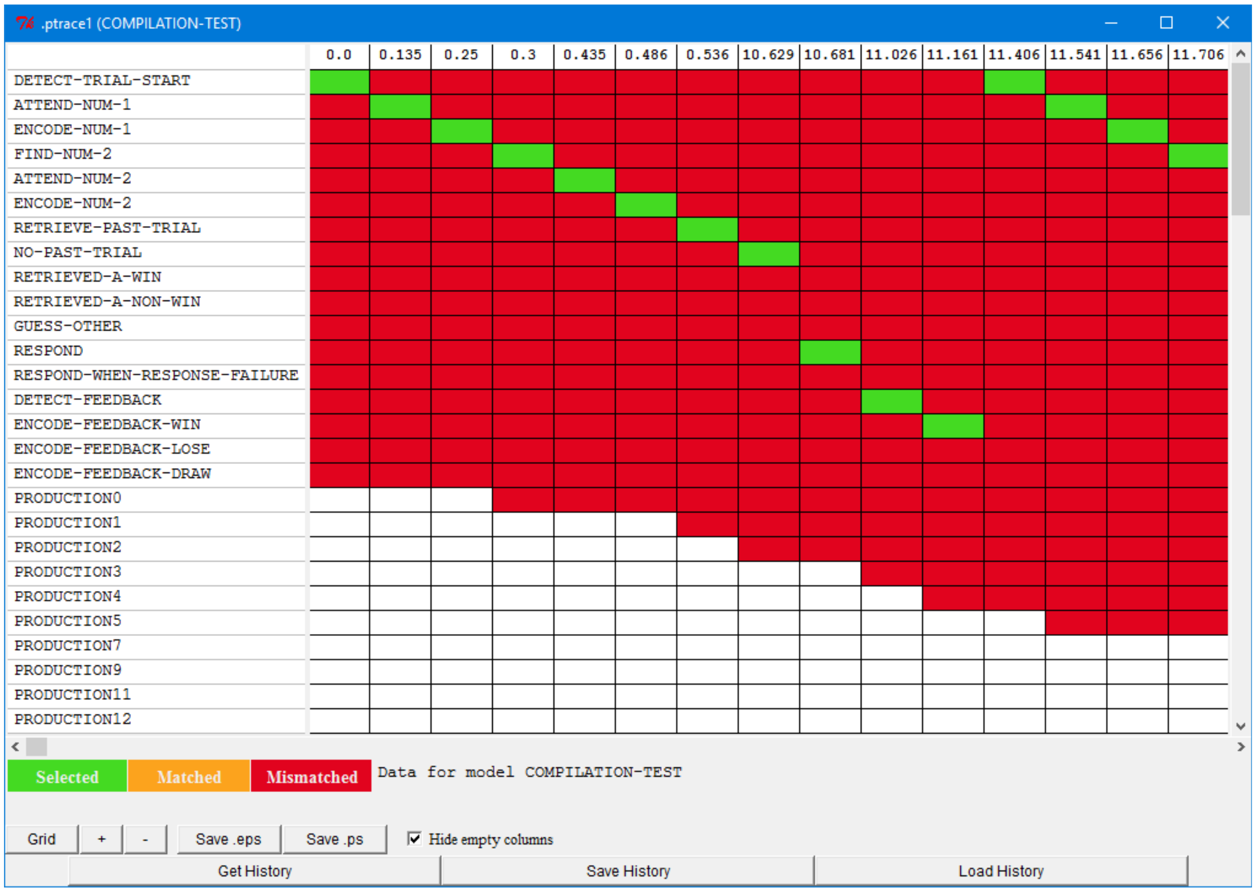
The reason it can keep a modifier in a request is because that request is not a constraint of the procedural system – all the information which is contained in a request is either constant values or was truthfully bound in the condition of the production. Therefore, it is not encoding any assumption that something is false or not available when composing that request.

Now we have verified that production compilation is able to compose the starting productions from the task into productions that seem reasonable. The next thing to investigate is whether or not the compiled productions are being used by the model and if so whether they are having an effect on how it performs the task.

There are many ways one can look for that, but here we will show how the “Production” grid tool in the Environment can be useful with production compilation. As we did above, we need to open the tool before running the model and then press the “Get History” button once the model is done. We will leave the seed value set to (79246602991 77656) so that we can recreate this run if we want to look at it again in detail, but turn the :v parameter off again since we don’t need to see the trace information. Here is the result from one game with that seed:

```
Average Score of 1 trials
-3.00 8.00 7.00 6.00 8.00 8.00 10.0 10.0 10.0 10.0 8.00 10.00 10.0 8.00 10.0 8.00 8.00 10.0 10.0 10.0
Average Response times
8.64 3.49 2.67 1.55 1.73 1.42 1.58 1.47 1.38 1.13 1.14 1.09 1.13 1.21 1.15 1.09 1.02 1.10 1.08 1.06
```

Looking at the results displayed the model still seems to be performing the task correctly and is still getting faster and more accurate as it performs the task. So there do not appear to be any problems introduced because of the productions that are being composed. In the production grid it is going to take a little while for the display to update after hitting “Get History” because of the amount of data to display, but once it does there should be a lot of new productions listed and many columns of data. It may help to check the "Hide empty columns" box at the bottom to remove the output for conflict resolution events that did not result in selecting a production. The results will look something like this:



	0.0	0.135	0.25	0.3	0.435	0.486	0.536	10.629	10.681	11.026	11.161	11.406	11.541	11.656	11.706
DETECT-TRIAL-START	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥
ATTEND-NUM-1	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ENCODE-NUM-1	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
FIND-NUM-2	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩
ATTEND-NUM-2	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-NUM-2	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVE-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
NO-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-NON-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
GUESS-OTHER	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND-WHEN-RESPONSE-FAILURE	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
DETECT-FEEDBACK	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-LOSE	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-DRAW	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION0				🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION1							🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION2								🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION3										🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION4											🟥	🟥	🟥	🟥	🟥
PRODUCTION5													🟥	🟥	🟥
PRODUCTION7															
PRODUCTION9															
PRODUCTION11															
PRODUCTION12															

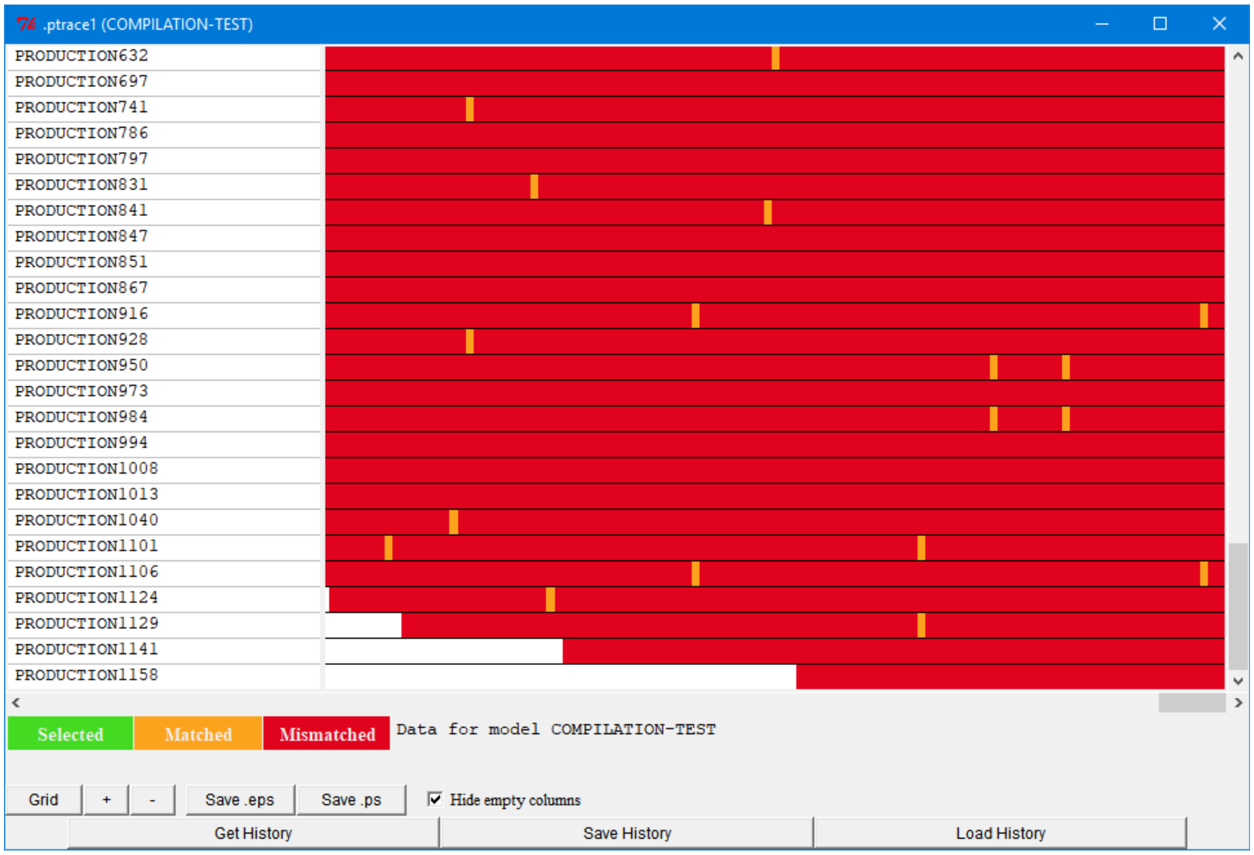
🟩 Selected 🟧 Matched 🟥 Mismatched

```
Data for model COMPILATION-TEST
```

[✓ Hide empty columns]

We discussed how to read the results of this display previously, but there is something new about this display because of production compilation. The newly compiled productions have white boxes in some columns which do not report any details when the mouse is placed over them. Those boxes indicate that that production did not exist at that time. Thus the first non-white box in a row indicates approximately when the production was created because that was the first time it was attempted to be selected.

The composed productions are also displayed in the order in which they were created. This provides us with a fairly easy way to determine if the model is continuing to learn new productions throughout the task, or if there appears to be a point at which it has learned all the new productions that it can. If we zoom out on the display by hitting the "-" button, turn off the vertical lines by hitting the "Grid" button, scroll down to the last new production, and then scroll right to see the end of the task we will see something like this:



That shows that even near the end of the task this model was still composing new productions. That may or may not be a good thing depending on what one was expecting for the task. Given the overall length of our task, approximately 10 minutes, it does not seem unreasonable that there are still opportunities for further learning at the end, but in other models one might expect compilation to slow down or stop before the end of the task.

Now we will start looking at the productions which the model has generated in more detail. If there were not as many then it might be worthwhile to use the "Procedural" viewer to look at all of them to see what they look like and what their utilities are at the end. However, since there are more than 100 composed productions and there did not appear to be any problems as it performed the task we are going to just look for productions that have an interesting history to investigate. In particular, the things that will be considered interesting are productions which never match because those might indicate a problem which we did not notice previously and new productions which are actually used by the model because those should be the ones that we are expecting it to learn and use.

There are a few ways to find productions which are never matched based on the details recorded automatically by ACT-R. One way is by using the Grid tool in the Environment and looking for rows with no orange or green boxes in them. If we zoom out they should be fairly easy to locate, and some of the first few productions learned, production0, production4, and production5 all seem to have that property, as do several others. Another way to find them would be to use the "Procedural" viewer to look for productions which have a :utility parameter value of nil. That parameter records the utility the production had the last time it matched, and if it is nil it means that it has never matched. We can also test that parameter value in code because we can get the production parameters using the spp command. That allows us to do something like this in Lisp to create a list of all the productions which have a nil :utility parameter setting:

```
? (mapcar 'car (remove-if (lambda (x) x) (no-output (spp :name :utility)) :key 'second))
(RESPOND-WHEN-RESPONSE-FAILURE  PRODUCTION0  PRODUCTION4  PRODUCTION5  PRODUCTION6  PRODUCTION11
PRODUCTION12  PRODUCTION22  PRODUCTION29  PRODUCTION46  PRODUCTION47  PRODUCTION189  PRODUCTION242
PRODUCTION847   PRODUCTION867   PRODUCTION973   PRODUCTION994   PRODUCTION1008   PRODUCTION1141
PRODUCTION1158)
```

or something very crudely like this in Python (I’m sure there are much nicer ways to do so):

```
>>> actr.hide_output()
>>> all = actr.spp(':name',':utility')
>>> actr.unhide_output()
>>> for i in all:
...   if i[1] == None:
...     print(i[0])
...
RESPOND-WHEN-RESPONSE-FAILURE
PRODUCTION0
PRODUCTION4
PRODUCTION5
PRODUCTION6
PRODUCTION11
PRODUCTION12
PRODUCTION22
PRODUCTION29
PRODUCTION46
PRODUCTION47
PRODUCTION189
PRODUCTION242
PRODUCTION847
PRODUCTION867
PRODUCTION973
PRODUCTION994
PRODUCTION1008
PRODUCTION1141
PRODUCTION1158
>>>
```

However we go about finding them, there are 20 such productions in this model. We will not look at each individually here, but what you will find if you do is that they basically fall into four general categories which we will discuss. Before continuing, you might want to look them over and see if you can find the similarities among them yourself.

The first category are those that we already knew would not be used -- productions which are composed from a production which makes a response and one which detects the result of that response. Those involve either detect-feedback or detect-trial-start as the second production in the pair. Since we expected these to occur we do not need to investigate them further.

The next category are productions for rare situations, particularly those dealing with the retrieved-a-non-win production. We know that is not a common occurrence in the model since we had to search to find a game in which it occurred. Because of that the productions composed from it are also not likely to have an opportunity to match either. That does not seem to be a problem we need to investigate any further. One of the productions which never matches is actually starting production in the model: respond-when-response-failure. The respond-when-response-failure production is only needed if the model ever fails to retrieve a response, and since that should not happen we would not expect to see that production selected and fired. It could probably be removed from the starting model without affecting things, but it is often safest to include productions like that in a model so that it can deal with unexpected situation. It is possible, no matter how unlikely, for the noise in the activations to push all chunks below the retrieval threshold and if the model does not have any productions to deal with failures to retrieve it will be stuck and unable to perform the task.

Another category of productions which does not match are those created late in the run which have very specific constraints. Presumably those productions are not matching because that specific pair of numbers is not presented again before the end of the experiment. Here are some examples of that:

```
(P PRODUCTION973
  "PRODUCTION19 & RETRIEVED-A-WIN - CHUNK74-0"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
       NUM1 ZERO
   =VISUAL>
       VALUE "1"
 ==>
   =IMAGINAL>
       NUM2 ONE
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
(P PRODUCTION1158
  "RETRIEVE-PAST-TRIAL & PRODUCTION26 - CHUNK92-0"
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 ZERO
       NUM2 TWO
   ?MANUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE "s"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "s"
)
```

Production1158 is the type of production that we were expecting the model to learn. It maps a specific **imaginal** chunk representation to a specific action. Production973 seems to be a further step in that process where the encoding of the number from the **visual** buffer into the **imaginal** is also composed with the response. Seeing these productions is a good sign because they are what we expected and are composed from previously composed productions so that means the model is actually using some of the composed productions which we will look into further shortly.

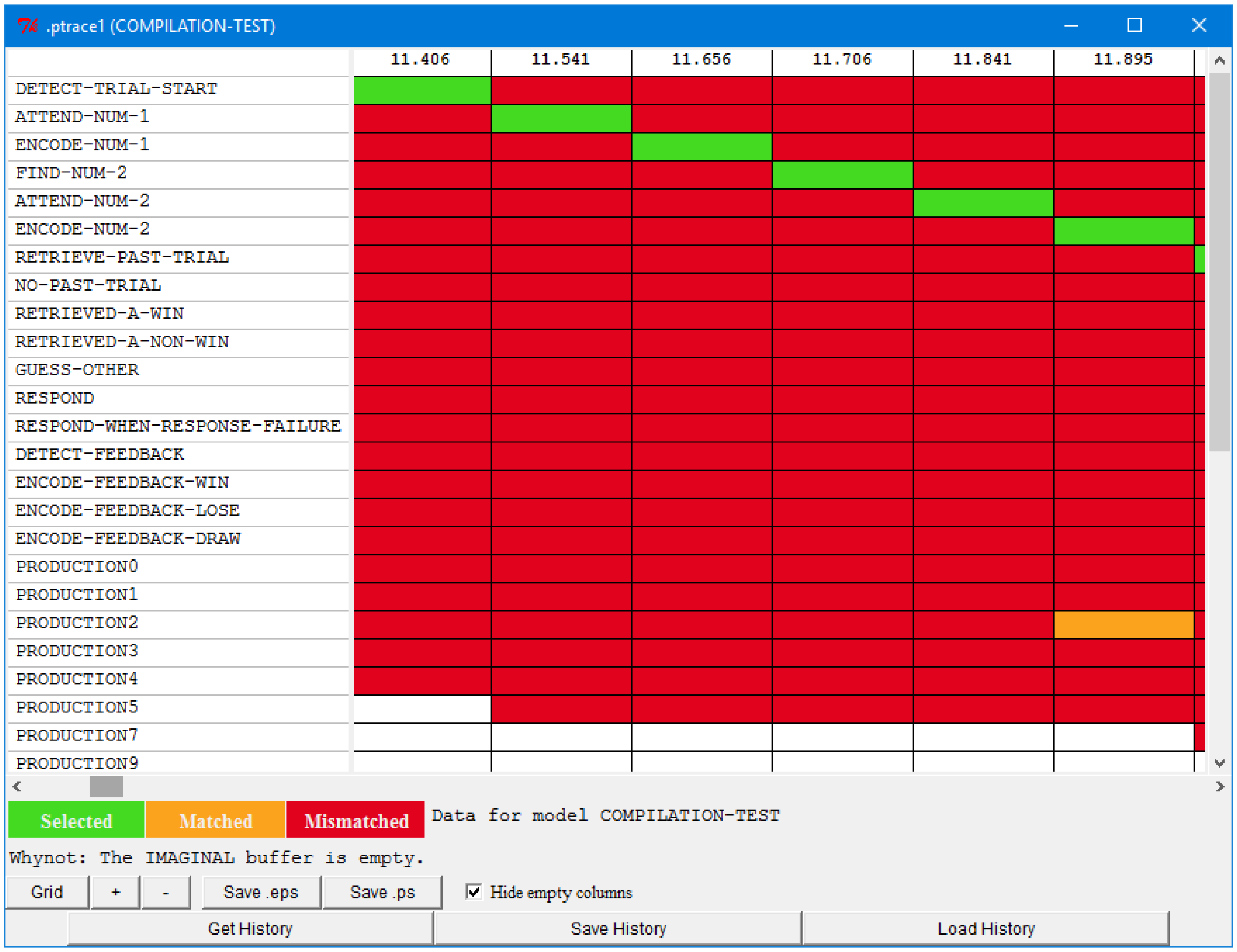
The final category of productions which are not being matched are productions composed from attend-num-1 and encode-num-1. There are four such productions, one for each of the numbers retrieved (zero, one, two, and three). They all have the same structure and here is one of them for reference:

```
(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM1 ZERO
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
```

We discussed this production before and expected it to help the model speed up over time, so the question is why isn't it being selected? If we look for the similar productions which compose attend-num-2 and encode-num-2, like production1:

```
(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ONE"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "1"
 ==>
   =IMAGINAL>
       NUM2 ONE
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
```

we find that it is matched multiple times over the course of the experiment so it seems odd that production0 is not also matched. To figure out why production0 is not matched we can use the Grid tool to look at the why-not information for production0 when we would expect it to be matched, which is when attend-num-1 matches since that is the parent production with which it should be competing. To find that it is probably easiest to zoom in again and restore the grid lines. The first column that we find which has attend-num-1 selected while production0 exists is at time 11.541 and this is what we find when we place the cursor over the red box in the production0 row:



	11.406	11.541	11.656	11.706	11.841	11.895
DETECT-TRIAL-START	🟩	🟥	🟥	🟥	🟥	🟥
ATTEND-NUM-1	🟥	🟩	🟥	🟥	🟥	🟥
ENCODE-NUM-1	🟥	🟥	🟩	🟥	🟥	🟥
FIND-NUM-2	🟥	🟥	🟥	🟩	🟥	🟥
ATTEND-NUM-2	🟥	🟥	🟥	🟥	🟩	🟥
ENCODE-NUM-2	🟥	🟥	🟥	🟥	🟥	🟩
RETRIEVE-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥
NO-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-WIN	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-NON-WIN	🟥	🟥	🟥	🟥	🟥	🟥
GUESS-OTHER	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND-WHEN-RESPONSE-FAILURE	🟥	🟥	🟥	🟥	🟥	🟥
DETECT-FEEDBACK	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-WIN	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-LOSE	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-DRAW	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION0	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION1	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION2	🟥	🟥	🟥	🟥	🟥	🟧
PRODUCTION3	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION4	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION5		🟥	🟥	🟥	🟥	🟥
PRODUCTION7						
PRODUCTION9						

🟩 Selected 🟧 Matched 🟥 Mismatched Data for model COMPILATION-TEST

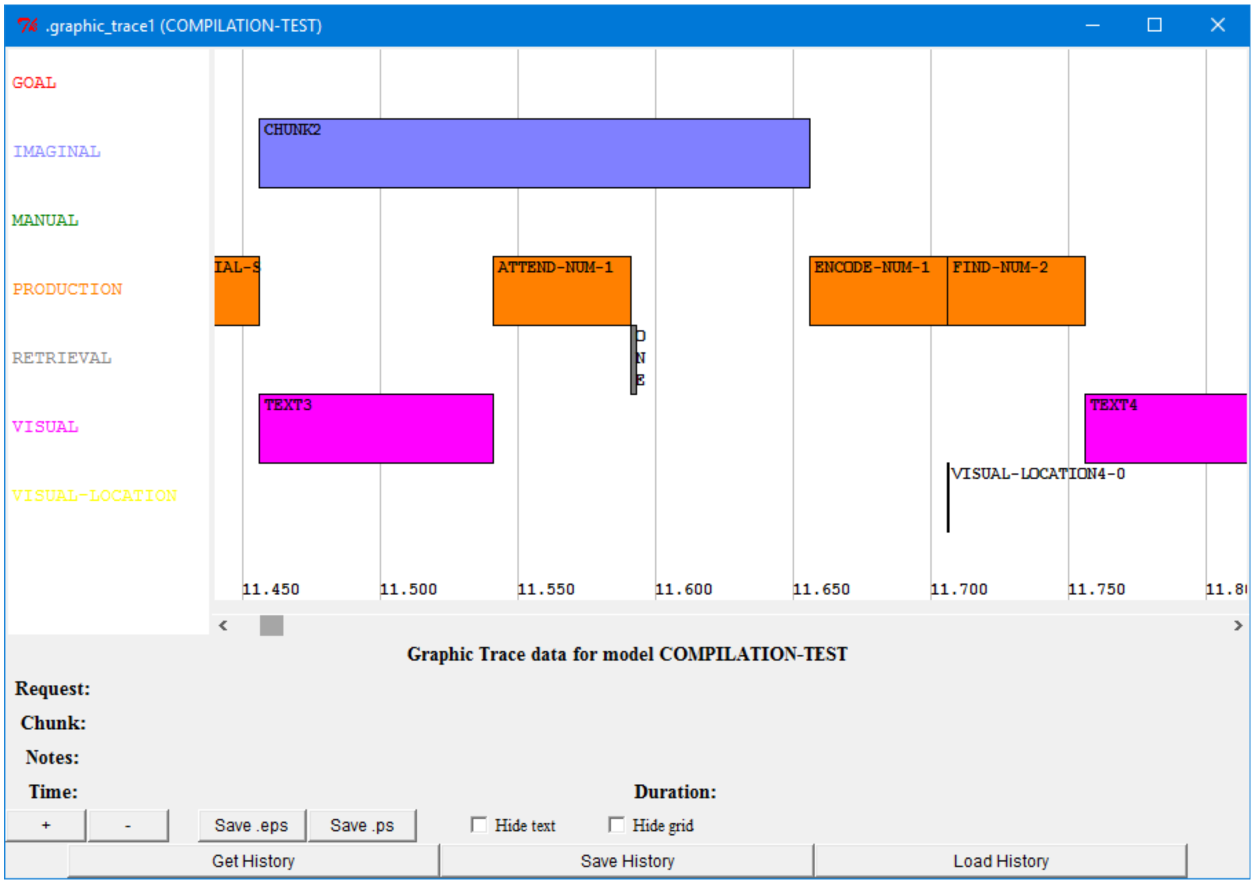
Whynot: The IMAGINAL buffer is empty.

☑ Hide empty columns

It is not matching because the **imaginal** buffer is empty. The question then becomes why is the **imaginal** buffer empty at that time? If you look at the model's productions you may be able to ascertain why that is happening, but if not there are multiple ways to look into that further. One would of course be to run it again with the trace on and look at the trace to see if you can determine why. Another option would be to step through the operation with the Stepper tool so that you can inspect things more closely as they occur. Something else which can be done, and which we will use here, is to use a “Graphic trace” tool for recorded data from the Environment instead of the text trace to try to determine what is happening.

To do that we need enable the saving of the “Graphic trace” information by opening the tool before we run the model (either the horizontal or vertical version can be used and we’ll show the horizontal one here). Since we know this happens on the second trail we can just run the model for two trials instead of waiting for it to run the entire experiment.

Once that’s done we can press the “Get History” button and then go to the appropriate time of that happening:



There we see that the **imaginal** module is busy creating a chunk at time 11.541 as requested by the detect-trial-start production, and attend-num-1 is selected while that request is ongoing. Production0, like encode-num-1 which it is composed from, requires that there be a chunk in the **imaginal** buffer to match. Since attend-num-1 does not have that requirement it can be selected while the imaginal module is still busy. That is another important thing to remember about production compilation - a composed production will have to meet the constraints imposed by both parents. If, as is the case here, the constraints for the second production take time to occur then that composed production may not compete with its first parent and may never match. While it seems like this is a lost opportunity for speedup in the model, looking at the other information in the graphic trace actually shows that it does not really matter. That is because the retrieval of the number chunk also completes before the **imaginal** chunk is created. Thus, the time spent creating that **imaginal** chunk determines when encode-num-1 (or our composed production0) will be able to be selected and fired. Eliminating attend-num-1 and the number retrieval through composition would not change that timing. If we wanted to see a speedup from composing these productions we would have to adjust when the model makes the request to create the **imaginal** chunk so that it does not dominate this timing or change the time it takes for **imaginal** actions to occur. That does not seem like something worth changing in the model since we are primarily expecting the speedup to occur because of composing the specific response information in this model, but you are welcome to try those options as an additional exercise if you like.

Now that we have looked at the composed productions which are not matching we will look at those which are being selected and fired to make sure that the model is learning to use the new productions that we expected. Like finding those that were not matched there are multiple options available for finding those which do match. However, there is not a simple parameter or other automatically recorded information which we can test to do so. Thus, getting this information will require either using the production grid tool or setting additional parameters in the model before running it. Probably the easiest way is to again use the “Production” grid, and this time instead of looking for empty rows we are looking for rows with lots of green and orange in them. If we want to see which productions are selected we can get that from the model trace if we enable it, but to see those that match but which are not selected we will also have to enable either the :cst or :crt parameter to include the additional conflict resolution information. If we want to collect that information in a list or process it in code then we would have to explicitly collect the information while the model runs using the :conflict-set-hook parameter or ask the module to record the history internally and then get the data when it is done. How to use the conflict-set-hook and the history recording tools are beyond the scope of this document, but details can be found in the reference manual.

Looking at the history there are lots of new productions which are matching frequently, but there are only a few which are getting selected and fired frequently. Those productions are production2, production7, production26, production57, production86, production226, and production377. Those productions seem to fall into two categories: productions which are collapsing the steps needed for encoding the second item and productions which are making a response based on a retrieved response chunk. We expected items of the first type to be created and used, but the second type, like production26, are not quite what we were looking for:

```
 (P PRODUCTION26
  "RETRIEVED-A-WIN & RESPOND - RESPONSE-1"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   =RETRIEVAL>
       RESPONSE "s"
       RESULT WIN
   ?MANUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE "s"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "s"
)
```

That production has removed a retrieval which should reduce the time it takes to respond, but it is not the main type of production we were looking to create. That production does not map the trial information to a particular response. It just eliminates the retrieval of the response chunk that occurred before it made the response. The productions we really want the model to start using will be a combination of retrieve-past-trial and retrieved-a-win or another production which has been composed from retrieved-a-win. So, now we will look for some of those and see why they are not being selected.

Looking through the generated productions we do find instances of the productions we want, like production45:

```
(P PRODUCTION45
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-NON-WIN - CHUNK12-0"
   !SAFE-EVAL! (NOT (EQUAL (QUOTE LOSE) (QUOTE WIN)))
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 THREE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE GUESS-OTHER
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
```

Looking at the history shows productions like that do match a few times, but they are not being recreated enough to raise their utilities to a point where they are able to be selected over the original productions. Since it is creating them and they do match, that is all we are concerned with for now.

Now that we have looked at the productions the model learns and seen that they do not cause any problems for the model's ability to do the task we can start looking at the effect they have on the model's performance on the task. To do that we will want to remove the seed setting from the model and run it over multiple trials to see the average results. When doing that it will also be a good idea to look at the individual game outcomes as well to make sure there are not any problems along the way, and printing the seed for each will allow us to recreate a bad run if we see one.

To help with that we can use the optional parameter of the pcomp-issues-game function in Lisp and the game function in the pcomp_issues module for Python to have it output the results for each game run before displaying the average at the end. Here are some results from running 10 games with the individual game results and each game's starting seed shown:

```
:SEED (5775195006 238051) (default NO-DEFAULT) : Current seed of the random number generator
Score
   4  10   9  10   7   9  10  10   9  10  10  10  10  10  10  10  10  10  10  10
Average response times
6.82 2.44 1.93 2.45 1.45 1.38 1.31 1.29 1.41 1.22 1.10 1.16 1.19 1.05 1.13 1.10 1.04 1.10 0.95 1.09
:SEED (5775195006 248483) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3  10   7   7   9  10  10  10   9  10  10  10  10  10  10  10   9   7  10  10
Average response times
8.76 3.40 1.96 1.87 1.33 1.45 1.29 1.34 1.20 1.37 1.11 1.14 1.07 1.17 1.14 1.03 1.03 1.13 1.01 1.04
:SEED (5775195006 258658) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -1  -2  -2   8  10   4   8   8  10   8  10  10   8  10  10  10  10  10  10   8
Average response times
6.56 5.86 5.59 2.82 1.84 1.79 1.56 1.55 1.31 1.28 1.20 1.11 1.17 1.35 1.19 1.05 1.15 0.97 1.05 0.99
:SEED (5775195006 269418) (default NO-DEFAULT) : Current seed of the random number generator
Score
   1   3   6   6   8   7   8   6   4  10   6   8   8  10   8   8   8   6  10  10
Average response times
8.35 6.48 3.22 2.27 1.52 2.22 2.02 1.43 1.57 1.42 1.31 1.37 1.12 1.16 1.36 1.18 1.11 1.08 1.01 1.05
:SEED (5775195006 279707) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3   3   9   6   8   8  10   8   9  10  10   9  10  10  10  10  10  10  10  10
Average response times
```

```
8.29 4.19 3.51 1.83 2.27 1.19 1.16 1.37 1.25 1.33 1.16 1.16 1.11 1.02 1.08 1.07 1.04 1.00 1.02 1.01
:SEED (5775195006 290250) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -2   6   7   8  10  10  10   8  10  10  10  10  10  10  10  10  10  10  10  10
Average response times
10.02 6.12 3.14 1.97 1.57 1.55 1.34 1.64 1.15 1.25 1.14 1.05 1.22 1.12 1.16 1.05 1.09 1.01 1.08 1.00
:SEED (5775195006 301195) (default NO-DEFAULT) : Current seed of the random number generator
Score
   6  10  10   8   9   7   6   8  10   7   9   6   8   8   8   8  10   9   9  10
Average response times
5.82 3.63 1.84 1.51 1.93 1.59 1.62 1.42 1.51 1.31 1.23 1.16 1.24 1.18 1.07 1.08 1.01 1.10 1.10 1.00
:SEED (5775195006 310963) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3   7   2   7   6  10   8   8   4   8   8  10   9   6   5   9  10   8  10  10
Average response times
7.45 3.85 2.91 2.87 2.64 1.93 1.38 1.42 1.39 1.48 1.51 1.16 1.12 1.45 1.17 1.21 1.32 1.06 1.12 1.03
:SEED (5775195006 321560) (default NO-DEFAULT) : Current seed of the random number generator
Score
   7   9   9   7   8  10   8   8   8   9  10   8   9  10   9  10   9  10  10   9
Average response times
8.16 2.85 1.74 1.87 1.63 1.49 1.30 1.39 1.23 1.41 1.25 1.23 1.22 1.13 1.07 1.05 1.13 1.04 1.04 1.07
:SEED (5775195006 331681) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -1   9   9   8   8   9   8   8   8   7  10   8  10   7  10   9  10   9   9   8
Average response times
9.74 4.49 3.06 1.80 1.49 1.37 1.45 1.32 1.34 1.23 1.28 1.12 1.13 1.03 1.17 1.04 1.05 1.10 1.02 1.04

Average Score of 10 trials
2.30 6.50 6.60 7.50 8.30 8.40 8.60 8.20 8.10 8.90 9.30 8.90 9.20 9.10 9.00 9.40 9.60 8.90 9.80 9.50
Average Response times
8.00 4.33 2.89 2.13 1.77 1.60 1.44 1.42 1.34 1.33 1.23 1.17 1.16 1.17 1.16 1.09 1.10 1.06 1.04 1.03
```

The average results still show the same learning patterns we expect, the scores go up and response time goes down, and the individual games do not seem to show any particularly unusual situations occurring. We could run some more tests, but since we have inspected the productions the model learns fairly thoroughly and this small test looks good we are going to assume that it is working well and move on to looking at the average data.

Here are the results of the model without production compilation averaged over 50 runs:

```
Average Score of 50 trials
2.06 5.22 6.12 7.56 7.86 8.00 8.22 8.36 8.44 7.94 8.86 8.86 8.52 8.62 8.74 9.24 8.82 8.70 9.10 9.00
Average Response times
7.97 4.68 3.21 2.36 1.94 1.68 1.56 1.47 1.39 1.38 1.31 1.27 1.26 1.25 1.20 1.19 1.21 1.18 1.16 1.15
```

and here are the results of the model with production compilation averaged over 50 runs:

```
Average Score of 50 trials
2.88 5.78 6.76 7.52 7.90 7.76 8.10 8.16 8.88 8.00 8.40 8.78 8.42 8.72 8.98 9.02 8.82 8.44 8.46 8.88
Average Response times
7.73 4.63 2.96 2.31 1.78 1.60 1.51 1.39 1.35 1.27 1.20 1.20 1.17 1.14 1.11 1.10 1.09 1.08 1.08 1.06
```

The average scores look fairly similar between the two as do the response times, and about the only difference seems to be that the model is slightly faster with production compilation. So, unfortunately, setting up production compilation to work with that starting model has had little effect on the results. The most likely reason for the response times not being much different is because the initial model was already fairly compact in terms of the number of productions which it needed to perform the task and its use of base-level learning quickly sped up the retrievals necessary. Thus there was not a lot that compilation could remove to improve the speed. As for the scores, the effect we wanted (compiling specific response productions for the winning move on each potential trial) does not happen because as we saw above there are not enough trials for those productions to learn a utility strong enough to dominate the initial productions. Without any actual data to fit the model to there are no specific adjustments that we need to make now to adjust the model's performance, but we will describe some adjustments that could be made and you are welcome to investigate those changes or others to see what effects they have on the model's results.

If we wanted the model to show a more gradual speedup in response time through production compilation then we would have to make significant changes to the starting model so that it required more productions and more retrievals to perform the task initially. One way to do that would be to convert the model so that it has to retrieve task instructions like the unit 7 paired associate task instead of starting out with an already optimized set of task specific productions. Alternatively, we could change the declarative memory parameters that it uses so that it is not as fast to begin with, but that could also be done without the need for production compilation. Just changing the parameters for production compilation, like slowing the learning rate or adjusting the initial utilities, would not allow us to make the model perform any slower than the starting model because utility learning will favor the faster productions as long as they lead to the same rewards which they will in this task as long as the model is responding correctly.

If we want the model to speed up even more through production compilation then we could increase the utility learning rate so that the new productions get higher utilities sooner. We could also increase the noise parameter or the starting utilities of composed productions so that they are more likely to be selected and gain their own rewards sooner. That might help the model to use the productions we wanted it to learn sooner. However a change like that might also make the scores go down because it could allow composed productions which make bad responses to get selected more often as well as the good ones. As an example, here are the results from running the model with an :alpha value of .9 (a very fast learning rate):

```
Average Score of 30 trials
2.13 3.90 4.90 6.00 6.93 7.20 7.70 7.63 7.37 7.37 7.77 7.73 8.10 7.63 8.23 8.17 8.17 8.47 8.10 8.40
Average Response times
8.38 5.73 3.65 2.90 2.62 2.06 1.79 1.63 1.63 1.38 1.24 1.18 1.03 1.20 0.98 0.99 1.01 0.89 0.92 0.89
```

The response times have gotten smaller, but the scores have dropped by about a point as well. To see why that is happening you would have to look at the history of production usage and utilities that are learned, which we will not do here.

That brings up the final issue that we will discuss. Adjusting the parameters for a model which uses production compilation can be a more difficult process than for other models. That is because of the potential for indirect effects to occur because of the automatic composition of new productions. Thus, unlike other models where the parameters often map fairly directly onto behavior, now one also has to consider what new productions can be learned and how the parameters affect those as well. Those effects may not always be in the same direction as one would expect (for example a faster production compilation learning rate leading to fewer correct responses). So, just like the extra work that was required to test the model to make sure it operated correctly, adjusting the parameters can also require looking at the new productions which are created and how their utilities are changing as a result of parameter adjustments when trying to achieve a particular fit to data or other explicit result from the model.

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 two or more 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 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 partial matched in productions. If one wants all chunks to potentially match to any other 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 described before. The difference when procedural partial matching is enabled is that productions which are not a perfect match receive a reduction 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 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 there 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:

```
TEST0-0
   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 which is set to **t** in the model is the conflict set trace parameter. 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 this case that is only 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 a utility and U(t) of 0:

```
(P SMALL
   =GOAL>
       VALUE SMALL
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production SMALL:
 :UTILITY  0.000
 :U  0.000
```

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 comparison for the value slot, instead of just seeing medium as was specified in the production, we see a set of items in brackets (the red text above). 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 small (the blue text above). 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 and here is a run with medium being selected:

```
    0.000   PROCEDURAL             CONFLICT-RESOLUTION
(P SMALL
   =GOAL>
       VALUE SMALL
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production SMALL:
 :UTILITY -0.012
 :U  0.000
(P MEDIUM
   =GOAL>
       VALUE [MEDIUM, SMALL, -0.5]
       VALUE SMALL
 ==>
   !OUTPUT! SMALL
   -GOAL>
)
Parameters for production MEDIUM:
 :UTILITY  1.418
 :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
    =goal>
      isa   try-strategy
      state choose-strategy
     - strategy   over
    =imaginal>
      isa         encoding
      goal-length =d
      b-len       =d
   ==>
    =imaginal>
    =goal>
      state   prepare-mouse
      strategy over
    +visual-location>
      isa   visual-location
      kind     oval
      value    "b")
```

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

```
CHUNK0-0
   A-LOC  LINE-LOCATION0-0
   B-LOC  LINE-LOCATION1-0
   C-LOC  LINE-LOCATION2-0
   GOAL-LOC  LINE-LOCATION3-0
   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:

```
(p decide-over
...
    =imaginal>
      isa         encoding
      goal-length =d
      b-len       =d
   ==> ...)

(p decide-under
...
    =imaginal>
      isa         encoding
      goal-length =d
```

```
        c-len       =d
==> ...)
```

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:

```
(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"
)
```

```
(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"
)
```

The effective utility of decide-over is -6.666666 (10 + 40 * -0.41666666) and the effective utility of decide-under is 0.666667 (10 + 40 * -0.23333333). Knowing that 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:

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

$$\begin{aligned}\Pr obability(over) &= \frac{e^{-6.666/4.24}}{e^{-6.666/4.24} + e^{.666/4.24}} \\ &\approx .15 \\ \Pr obability(under) &= \frac{e^{0.666/4.24}}{e^{-6.666/4.24} + e^{.666/4.24}} \\ &\approx .85\end{aligned}$$

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:  0.803
MEAN DEVIATION: 17.129

Trial 1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
     23.0 60.0 59.0 70.0 91.0 42.0 80.0 86.0 59.0 34.0 33.0 22.0 54.0 72.0 56.0

DECIDE-OVER : 13.1506
DECIDE-UNDER: 11.1510
FORCE-OVER  : 12.1525
FORCE-UNDER : 6.5943
```

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

```
CORRELATION:  0.851
MEAN DEVIATION: 14.839

Trial 1    2    3    4    5    6    7    8    9    10   11   12   13   14   15
     16.0 69.0 49.0 74.0 89.0 31.0 67.0 93.0 66.0 33.0 31.0 23.0 57.0 64.0 43.0

DECIDE-OVER : 11.7534
DECIDE-UNDER: 6.9756
```

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 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
    ==>
     ...)


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

The slot tests in red above are instances of dynamic pattern matching. 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:

```
FACT0-0
   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 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:

```
(P START
   =GOAL>
       CONTEXT DATA
       DATA 10
   ?IMAGINAL>
       STATE FREE
       BUFFER EMPTY
 ==>
   =GOAL>
       CONTEXT DESTINATION
   +IMAGINAL>
       DATA1 10
   +RETRIEVAL>
       STEP FIRST
)
```

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:

```
  (p retrieve-first-step
     =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, the **goal** buffer's chunk looks like this:

```
FACT0-0
   CONTEXT  DESTINATION
   DATA  10
```

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.

The chunk created by the start production in the **imaginal** buffer looks like this:

```
CHUNK0-0
   DATA1  10
```

That matches the pattern for the **imaginal** buffer in retrieve-first-step since =x is bound to 10.

The chunk which is in the **retrieval** buffer at that time looks like this:

```
A-0
   STEP  FIRST
   DESTINATION  DATA2
```

The =slot variable is bound to destination from the **goal** buffer, and thus the variable =target will be bound to the value of the destination slot from the chunk in the **retrieval** buffer, which is data2. If there was not a slot named destination in the chunk in the **retrieval** buffer then this pattern would not have matched and the production would not be selected. 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:

```
  (p retrieve-first-step
    ...
    ==>
     =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.

```
CHUNK0-0
   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: FACT0-0
FACT0-0
   CONTEXT  DESTINATION
   DATA  11
RETRIEVAL: B-0 [B]
B-0
   STEP  SECOND
   DESTINATION  DATA3
```

The variable bindings from the conditions of the retrieve-second-step production are: =slot is bound to destination, =x is bound to 11, and =target is 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:

```
CHUNK0-0
   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 slots (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 the 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 a lot 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 rather 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
  ==>
   =goal>
     state       attr-check
   +retrieval>
     ISA         property
     object      =obj
     - =attr     nil)

(P direct-verify
   =goal>
     ISA         test-attribute
     original    nil
     object      =obj
     attribute   =attr
     value       =val
     state       attr-check
   =retrieval>
     ISA         property
     object      =obj
     =attr       =val
  ==>
   =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. You should be able to understand how those work now and we will not cover all of them in detail.

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:

```
     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 IMAGINAL CHUNK0
     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
...
     0.750   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 CHUNK0-0
     0.850   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL 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:

```
CHUNK0-0
   WORD  "tent"
   NUMBER  "5"
```

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 retireve-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.

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:

```
  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
)
```

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:

```
  Production RETRIEVE-OPERATOR and RETIREVE-ASSOCIATE are being composed.
  New production:

(P PRODUCTION2
  "RETRIEVE-OPERATOR & RETIREVE-ASSOCIATE - OP2"
   =GOAL>
       STATE STIMULUS-READ
       STEP READY
   =IMAGINAL>
       WORD =STIMULUS
 ==>
   =GOAL>
       CONTEXT NUMBER
       STATE RECALLED
       STEP RETRIEVING-RESULT
   +RETRIEVAL>
       WORD =STIMULUS
)
```

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 does, 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:  0.989
MEAN DEVIATION:  0.105
Trial    1       2       3       4       5       6       7       8
        0.000   2.059   1.929   1.806   1.730   1.663   1.628   1.589


Accuracy:
CORRELATION:  0.996
MEAN DEVIATION:  0.034
Trial    1       2       3       4       5       6       7       8
        0.000   0.432   0.657   0.788   0.873   0.927   0.943   0.957
```

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:

```
CHUNK1-0
   NAME  EH
   VALUE  0.704
   STATE  ADD-ATTRIBUTE
```

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 more 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:

```
CHUNK0-0
   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:

```
CHUNK0-0
   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 is currently encoded in the **imaginal** buffer. That goal chunk will have the state slot set to the value categorize:

```
CHUNK5-0
   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 using the default attribute names:

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

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

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
 …
 ==>
  +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 examples of calling that:

```
? (categorize-attribute size -.9)

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

Which would create this chunk in the **goal** buffer:

```
CHUNK1-0
   NAME  SIZE
   STATE  ADD-ATTRIBUTE
   VALUE  -0.9
```

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

```
CHUNK0-0
   SIZE  SMALL
   CATEGORY  UNKNOWN
```

That’s because a value of -.9 is most similar to the small label (although sometimes noise in the utility calculations should cause one of the other labels to also be applied). You should work with this command only 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 did this:

```
? (categorize-attribute s2 1.0)

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

The chunk in the **imaginal** buffer should look similar to this:

```
CHUNK0-0
   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. 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.

Once 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 encoded in the **imaginal** buffer:

```
CHUNK5-0
   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 and return the value 2 since that is the description of an item from category 2 in the model's declarative memory (again because of noise it may not always produce that same description and categorization):

```
CHUNK0-0
   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:  0.880
MEAN DEVIATION:  0.209
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 0.700 0.800 0.800 0.900 0.700 0.100 0.200 0.400 0.100 0.100 1.000 0.900 0.900 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:

```
CORRELATION:  0.948
MEAN DEVIATION:  0.174
```

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. Through adjusting those parameters the reference model was able to achieve this fit which improves the deviation slightly compared to using the starting values:

```
CORRELATION:  0.948
MEAN DEVIATION:  0.142
P(C=1)
      (500) (500) (500) (500) (500) (500) (500) (500) (500) (500) (500) (500) (500) (500)
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 0.992 0.780 0.978 0.988 0.918 0.118 0.052 0.130 0.050 0.030 0.966 0.798 0.986 0.534
```

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.

Unit 8 Code Description

The example models for this unit either have no experiment code, or are driven by code that has been described in previous units. The assignment task's experiment code uses two new ACT-R commands (one in the experiment file and one in the model definition), but otherwise is similar to code from other unit's tasks. Therefore only those new commands will be described here without describing all of the experiment code. In addition to that, this text is going to explain how the new Building Sticks model avoids using a !bind! to do a calculation and instead uses a request to the imaginal module to do so, which relies on code in the updated bst experiment files for this unit.

Extending chunks from code

The one new command used in the assignment task code is found in the create-example-memories function in Lisp and create_example_memories in Python:

```
(defun create-example-memories ()
  (dolist (x *slots*)
    (extend-possible-slots x nil)
    (define-chunks-fct `((,x isa chunk))))
  (dolist (x *cat1*)
    (add-dm-fct `((isa example category 1 ,@(mapcan 'list *slots* x)))))
  (dolist (x *cat2*)
    (add-dm-fct `((isa example category 2 ,@(mapcan 'list *slots* x))))))
```

```
def create_example_memories():

    for s in slots:
        actr.extend_possible_slots(s,False)
        actr.define_chunks([s,"isa","chunk"])

    for c in cat1:
        chunk = ["isa","example","category",1]
        for slot,value in list(zip(slots,c)):
            chunk.append(slot)
            chunk.append(value)
        actr.add_dm(chunk)

    for c in cat2:
        chunk = ["isa","example","category",2]
        for slot,value in list(zip(slots,c)):
            chunk.append(slot)
            chunk.append(value)
        actr.add_dm(chunk)
```

Those functions are responsible for adding the chunks which represent the studied items to the model's declarative memory using slot names for the features from a globally defined list of slots. Since those slot names can be specified arbitrarily by the modeler, the code needs to make sure that the model will accept them as valid slot names before creating the chunks since they may not have been specified in any of the model's chunk-types. That is what the extend-possible-slots and extend_possible_slots functions do. They have one required parameter and one optional parameter. The required parameter is the name of a slot to add to those which can be used in chunks (using a symbol in Lisp and a string in Python). The optional parameter indicates whether or not to print a warning if the slot which is provided has already been used to name a slot. If the optional parameter is specified as non-true (nil for Lisp or False/None for Python) then no warning is provided when a previously named slot is specified, otherwise it will print such warnings.

You may also notice looking at the code that those functions are added as ACT-R commands:

```
(add-act-r-command "create-example-memories" 'create-example-memories
  "Categorize task function to add the initial example chunks to simulate the training process.")

actr.add_command("create-example-memories",create_example_memories,
  "Categorize task function to add the initial example chunks to simulate the training process.")
```

That is done so that they can be called in the model's definition to generate those initial chunks for the model every time that it is reset.

Calling commands from the model definition

In the starting model file for the assignment you will find this line:

```
  (call-act-r-command "create-example-memories")
```

The call-act-r-command command can be used in a model definition to call any command which has been added to ACT-R. It requires one parameter which is the string that names the command and any number of additional parameters can be provided which will be passed to that command when it is called. In this case the command requires no parameters and thus none are given.

Instead of using call-act-r-command it is also possible when initially adding a command to specify a name which can be used directly in the model definition like the built-in ACT-R commands, but there are some additional complications with doing that and how to do so will not be described in the tutorial.

The Imaginal-action buffer

The imaginal module has a second buffer called **imaginal-action** which can be used by the modeler to make requests that perform custom actions. Those actions are typically used to modify the chunk in the **imaginal** buffer, replace the chunk in the **imaginal** buffer with a new one, or clear the **imaginal** buffer and report an error, but may perform any other arbitrary calculation desired. Those requests can also take time during which the imaginal module will be marked as busy. Note however, the **imaginal-action** buffer is not intended to be used for holding a chunk. The **imaginal** buffer is the cognitive interface for the imaginal module and the **imaginal-action** buffer exists for the purpose of allowing modelers to create new operations which can manipulate the **imaginal** buffer.

There are two types of requests which can be made to the **imaginal-action** buffer which are referred to as a generic action and a simple action. The model for the Building Sticks task in this unit uses a simple action with no extra information to create a new chunk for the **imaginal** buffer. The generic action is more powerful in terms of what it can do and for either the generic or simple action it is possible to provide additional details in the request. Those capabilities however require more care and programming from the modeler in handling the action and are beyond the scope of the tutorial. Users interested in using those capabilities should consult the reference manual for details.

Here is the production from the model which uses a simple action request to the **imaginal-action** buffer:

```
(p encode-line-current
    =goal>
      isa      try-strategy
      state    attending
    =imaginal>
      isa      encoding
      goal-loc =goal-loc
    =visual>
      isa      line
      width    =current-len
    ?visual>
      state    free
    ?imaginal-action>
      state    free
  ==>
    =imaginal>
      length     =current-len
    +imaginal-action>
      action     "bst-compute-difference"
      simple     t
    =goal>
      state      consider-next
    +visual>
      cmd        move-attention
      screen-pos =goal-loc)
```

A simple action request to the **imaginal-action** buffer requires specifying a slot named action which must contain a string that names a valid command or a symbol naming a Lisp function, and a slot named simple with any true value. At the bottom of the Building Stick experiment code files for this unit we find the function which implements the “bst-compute-difference” command and it being added:

```
(defun compute-difference ()
  (let* ((chunk (buffer-read 'imaginal))
         (new-chunk (copy-chunk-fct chunk)))
    (mod-chunk-fct new-chunk (list 'difference
                                   (abs (- (chunk-slot-value-fct chunk 'length)
                                           (chunk-slot-value-fct chunk 'goal-length)))))))

(add-act-r-command "bst-compute-difference" 'compute-difference
                   "Imaginal action function to compute the difference between sticks.")


def compute_difference():
    c = actr.buffer_read('imaginal')
    n = actr.copy_chunk(c)
    actr.mod_chunk(n,'difference',
                   abs(actr.chunk_slot_value(c,'length') - actr.chunk_slot_value(c,'goal-length')))
    return n

actr.add_command("bst-compute-difference",compute_difference,
                 "Imaginal action function to compute the difference between sticks.")
```

Those functions create a copy of the chunk in the **imaginal** buffer and then modify that copy to contain a slot named difference which holds the difference between the values in the length and goal-length slots of that chunk.

When a simple action request is made to the **imaginal-action** buffer the imaginal module performs the following sequence of actions:

- the imaginal module is marked as busy
- if the imaginal module is currently signaling an error that is cleared
- the named command is called with no parameters
- the **imaginal** buffer is cleared

Then after the current imaginal action time has passed (default of 200ms and set with the :imaginal-delay parameter) the following actions will happen:

- the imaginal module will be marked as free
- if the call to the action command returned the name of a chunk then that chunk will be placed into the **imaginal** buffer
- If the call to the action command returned any other value the **imaginal** buffer will remain empty, the imaginal module's error state will become true, and the **imaginal** buffer's failure query will be true.

Here is the segment from a trace showing the actions related to the simple-action request when the encode-line-current production fires:

```
     2.191   PROCEDURAL             PRODUCTION-FIRED ENCODE-LINE-CURRENT
...
     2.191   PROCEDURAL             MODULE-REQUEST IMAGINAL-ACTION
...
     2.191   PROCEDURAL             CLEAR-BUFFER IMAGINAL-ACTION
...
     2.191   IMAGINAL               CLEAR-BUFFER IMAGINAL
...
     2.391   IMAGINAL               SET-BUFFER-CHUNK IMAGINAL CHUNK0-0-0
```

Except for the additional clearing of the **imaginal-action** buffer, which should not hold a chunk anyway, it performs the same actions as an **imaginal** buffer request to create a new chunk would.

In the conditions of the encode-line-current production a query is made to test that the **imaginal-action** buffer has state free. That query will return the same state as the **imaginal** buffer does. Both buffers pass their requests to the same module which can only perform one action at a time regardless of which of its buffers was used to make the request. Thus it does not matter which buffer is used to test the state for the performance of the model, but to avoid a style warning testing the buffer for which a request is being made is preferable.

One important thing to note about a simple action request is that it will always clear the **imaginal** buffer. That means that the chunk currently in the buffer will become an element of the model's declarative memory at that time. In this model that does not matter because it is not retrieving those chunks later. However, in models where later retrieval is important, having intermediate chunks added to memory like that could cause problems. In those cases, one would probably want to use the generic action request to extend the imaginal capabilities because it does not clear the buffer automatically.