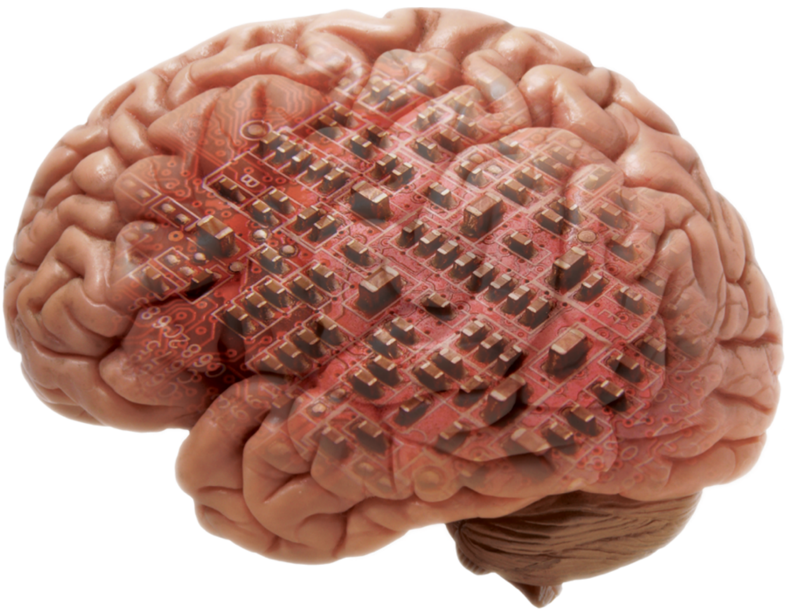
university of groningen / faculty of science and engineering / artificial intelligence

# Architectures of Intelligence

Academic year 2024-2025

WBAI009-05.2024-2025.1



Instructors:

Jelmer Borst (j.p.borst@rug.nl)

Catherine Sibert (c.l.sibert@rug.nl)

Ivo de Jong (ivo.de.jong@rug.nl)

## Welcome to Architectures of Intelligence!

*Architectures of Intelligence* are on the one hand general psychological theories of cognition, and on the other hand computational environments in which you can develop models of intelligent human behavior in particular tasks. In this course, you will gain insight into these architectures and their scientific status, experience in modeling within such an architecture, and practice comparing simulation results to empirical data. We will discuss two cognitive architectures in detail: the high-level architecture ACT-R and the low-level neural network architecture Nengo.

Throughout the course you will do seven assignments. In four of these assignments, you will learn to implement and run your own cognitive models in ACT-R by doing several units from the ACT-R tutorial. These units are described in this reader. More details about the course and the assignments can be found on Brightspace.

To get started with ACT-R, go to http://act-r.psy.cmu.edu/software/ and download the ACT-R standalone software. A detailed explanation on how to install and run ACT-R is provided below in *Unit 1*.

---

## Table of Contents

---

# 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" and 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.

## 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 followed by the 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.

## 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 all modules to see and modify, and the set of chunks in all of the buffers is the information that is immediately available to the model.

### 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.

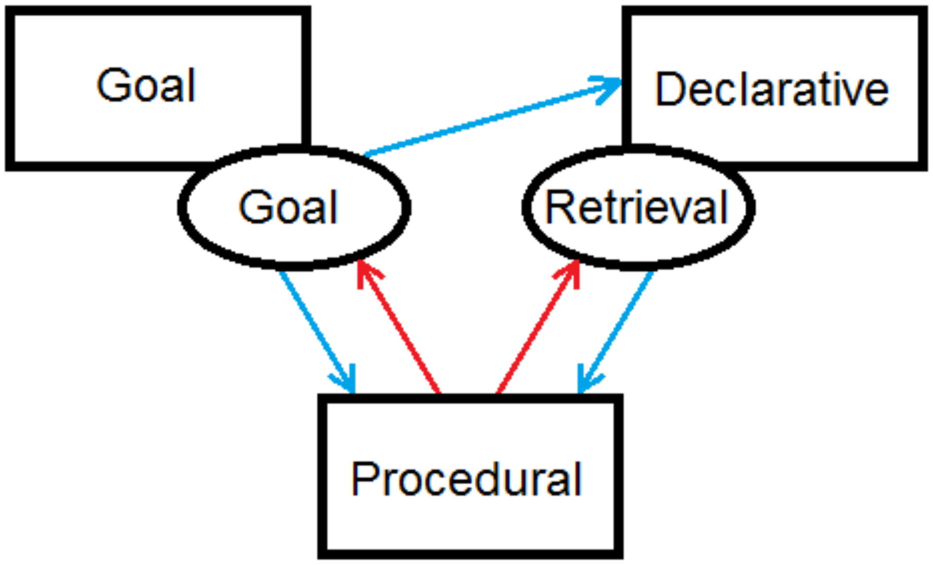
### 1.2.3 Procedural Module

The procedural module holds all of the productions which represent the model's procedural knowledge. It does not have a buffer of its own[1], 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 early 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:

[1]Technically, there is a buffer associated with the module in the software which is useful for tracking the activities of the module, but that is only there as a convenience for the user and not a component of the ACT-R architecture itself.



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 7.26 (or newer) of the ACT-R software. Current versions of the ACT-R software are distributed primarily as applications and the tutorial instructions will assume that the user is running one of those applications, but like the 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 which will be described in the tutorial. 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[2]. Using that Python module, all of the tasks for the models in the tutorial are implemented in Python as well as in Lisp,

[2] The Python module included with the tutorial is sufficient for running the tasks included with the tutorial. It does not contain functions for accessing all of the commands available through the ACT-R remote interface, and it has some assumptions about how it will be used based on the needs of the tutorial. For more complex tasks or where performance is of primary importance, one might be better served by creating a custom interface instead of using the one that was built for the tutorial.

and either version can be used with the models of the tutorial (the models do not depend upon which version of the task is being used). There are also examples of connecting other programming languages included with the ACT-R software, but only Lisp and Python will be used directly 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 called the ACT-R Environment[3]. 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 there are short videos on the ACT-R software site showing the typical OS protection warnings one may need to respond to the first time it is run. 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 the Environment tools are necessary to perform the tutored model running exercises described in this unit and it should not be closed now). Closing the ACT-R window will exit the software.

For this unit there are no tasks for the models to interact with, and all of 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 interactive tasks for the 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
```

[3]The ACT-R Environment is written in the Tcl/Tk language and connects through the same remote interface as the Python module.

?

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 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 which provides additional information about 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 called **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 still be a valid production for ACT-R. 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 white space characters between the items 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 example 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 of 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.

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. An action for a buffer is specified by a symbol which starts with a character that indicates the action to take, followed by the name of the 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 is for 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.

An important constraint on the use of the modification action is that a production can only use it for buffers that were matched to a pattern in the condition of the production – the production must test that there is a chunk in the buffer before it can modify it.

#### *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 a request 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. We will see another situation where implicit clearing occurs later in the unit.

## 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 in 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 mechanism that generated the event (typically the name of a module), 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[4], and this model is set to show the most information possible. That is more information than is typically needed, but some of the steps that will help to understand how the system works are only shown at this level of detail.

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.[5] The **count-from** chunk type specifies the slots that will be

[4] How to change the amount of detail shown in the trace is described in the unit 1 code description document (the "unit1_code" text).

[5] There are many ways which one could represent that information using either 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 which does not represent numbers as chunks and instead just uses digits), extending it when necessary to include more information.

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 slots indicating the number it represents and 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)
```

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).

### 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 that is bound to **=num1**. The other action is a request of the retrieval buffer. All requests to the declarative module (the retrieval buffer's module) perform a search of the chunks in declarative memory to find one that matches the information provided and then place that chunk into the **retrieval** buffer. This request is asking the declarative module to find a chunk that has the value which is bound to **=num1** in its number slot.

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 after another 50ms have passed and then the setting of the **retrieval** buffer with that chunk.

```
     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 **=num2** variable, 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 trace after the production fires:

```
     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 THREE
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
```

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 .3 seconds into the run:

```
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   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 Basic Model Operations Exercise

To help you understand how an ACT-R model runs, this section contains an exercise in which you will be asked to perform the operations of the procedural and declarative modules. You will be looking at the contents of the buffers to perform the conflict resolution process (determining which productions match the current state, if any), assigning the variables in the selected production to the values they have from the buffer chunks (called instantiating the production), and determining which chunks in the model's declarative memory, if any, match the **retrieval** buffer requests that the productions makes. This will require using two of the tools in the Control Panel while running the count model described in the previous section.

### 1.6.1 Load or reset the model if necessary

If you have not yet loaded the count model then you need to press the "Load ACT-R code" button on the ACT-R Environment Control Panel and select the "count.lisp" file in the tutorial/unit1 directory. If you have already loaded the model and run it, then you need to reset it to its initial conditions so that you can run it again. To reset the model you should press the "Reset" button on the Control Panel.

### 1.6.2 The Stepper

Press the "Stepper" button on the Control Panel to open the Stepper tool. When the model is run, for each event that shows as a line in the trace, the stepper will pause 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 useful tool when developing and debugging models. For some events, the stepper will also show additional information after it happens. The production-selected and production-fired events 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 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).

For this exercise, you should click the "Tutor Mode" checkbox at the top of the Stepper window. That will enable the additional interactions which you will be asked to perform for the conflict-resolution, production-selected, and start-retrieval events.

### 1.6.3 The Buffers tool

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 by default. If you press the "Status" button the display will switch to show the information about queries for the buffer, and pressing the "Contents" button will switch it back to showing the chunk. At this point all of the buffers are empty, but that will change as the model runs.

### 1.6.4 Running with the Stepper

Now you should run the model by pressing the "Run" button in the Control Panel. When you do that you will notice that unlike before nothing shows up in the ACT-R window. Instead, the first action that will occur, the setting of the chunk in the **goal** buffer, now shows at the top of the Stepper window after "Next Step:". That is the next action which will occur, but it is delayed until you press the "Step" button in the Stepper window to allow that action to occur. You should press it now, and then you will see that line printed out in the model trace. The Stepper now shows that event as "Last Stepped:" and a new event, conflict-resolution, is shown as the next step. In the Buffers tool window you can now see that there is a chunk in the **goal** buffer.

### 1.6.5 The goal-chunk0 Chunk and Buffer Chunk Copying

The chunk in the **goal** buffer looks like this:

```
GOAL: GOAL-CHUNK0 [FIRST-GOAL]
GOAL-CHUNK0
   START  TWO
   END  FOUR
```

The goal-focus command in the model made the goal module set the **goal** buffer to the chunk named **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 **goal-chunk0** is currently in the **goal** buffer. Why is that? When a chunk is placed into a buffer the buffer always makes its own 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.

### 1.6.6 Conflict-resolution

You should press the Step button again to allow the conflict-resolution action to occur. When you do so, the tutoring mode will open another window. In that window the conditions of the productions in the model are shown, and it is your task to pick which, if any, match the current state. To determine that you will need to use the Buffers tool to look at the chunks in the buffers and compare them to the patterns that are specified in the productions. Once you have chosen the item or items that you think match the current state you should press the "Check" button to have your choices compared to what the procedural model did during conflict resolution. If all of your choices were correct the window will close and you can continue stepping through the run. If any of your choices were incorrect then the conflict resolution window will show those incorrect choices, and if one of the choices does not match there will be a short description of the reason that it does not match. You will then need to press the "Ok" button to close the window and continue.

### 1.6.7 Production Selection

The next step shown is now production-selected, and you should press the Step button to allow that to happen. When a production-selected event happens the Stepper shows additional information about the selected production, which in this case, is the **start** production. In tutor mode, the production 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.

You will again need the information from the Buffers tool to be able to instantiate the production. One thing to remember is that a variable which names the buffer, for example the variable **=goal** in the =goal> condition, will always bind to the name of the chunk in the corresponding buffer. The bindings for the other variables will correspond to the value of the indicated slot of the tested buffer. A variable will have the same value everywhere in a production that has been selected[6], 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.

[6] At least for the models used through most of the tutorial, but in unit 8 we will see situations where that may not always be true.

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.

Once the production is completely instantiated, you can continue stepping through the model run with the Step button. You should step the model to start-retrieval event which is the next part of the tutored exercise.

### 1.6.8 Declarative retrieval

The start-retrieval event is an indication from the declarative memory module that it has retrieved a request to find a chunk in memory. If you press the Step button now it will open another window which is similar to the conflict resolution selection window. In this window you are shown the retrieval request which the production made and all of the chunks in the model's declarative memory. Your task is to select the chunks which match the request that was made. Once you have made your choices you should press the "Check" button. If you correctly chose the chunk or chunks which matched the request the window will close and you can continue stepping through the model. If you made any incorrect selections then those items will be shown and you will need to press the "Ok" button to continue.

You should now continue to step through the model performing the conflict resolution, production instantiation, and declarative memory matching exercises until the model finishes the task.

## 1.7 The Semantic Model

The next 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 did for the previous model. 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, for example, is a canary an animal or is a shark a bird.

### 1.7.1 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** (set in the attribute slot) by setting the value slot to **true**. **p2** encodes that a shark can swim by setting the value slot to **swimming** with the attribute slot set as

**locomotion**. **p3** encodes that a shark is a fish by setting **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 additional aspects of the pattern matching in productions which could be used with such a representation to make the searching more general.

### 1.7.2 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 already 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: GOAL-CHUNK0
GOAL-CHUNK0
   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,** uses 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 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, 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.

Something to notice about this production is that after it fires we see this event in the trace from the procedural module:

```
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
```

This is another instance of implicit buffer clearing. If a production matches the chunk from a buffer on the LHS and does not modify it on the RHS, then for most buffers[7] that chunk will be automatically cleared from the buffer.

Some terminology which is often used when discussing productions which use a chunk that is in a buffer and then clear that chunk from the buffer is to say that it "harvests" the chunk – the **direct-verify** production harvests the chunk from the **retrieval** buffer. We would not say that it harvests the **goal** buffer's chunk because it is modified and remains in the buffer. The implicit clearing of a chunk which is matched and not modified is referred to as "strict harvesting" and it is there as a convenience so that one does not need to add lots of explicit buffer clearing actions to the productions they write.

If you would like some more experience with how the ACT-R modules work, you can reset the model, open the Stepper, enable tutor mode, and run it like you did with the count model.

### 1.7.3 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**:

[7] Of the buffers used in the tutorial models, only the goal buffer is not subject to that automatic clearing mechanism.

```
 (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.[8]

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 will attempt to retrieve the next 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>
```

[8]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.

```
      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.

Again, for more experience, you can reset this model and step through it in tutor model.

### 1.7.4 The Failure Case

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

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

When 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
```

There 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.

## 1.7.5 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.

### *1.7.5.1 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, as was the case for the chunk placed into the **goal** buffer as a result of the goal-focus command in the model[9]. Otherwise, it will be false:

[9] Later units will describe modules which can perform operations without being requested to do so that are not just the result of commands specified by the modeler.

```
  ?retrieval>
      buffer     unrequested
```

### *1.7.5.2 Querying a buffer's module*

Every module must respond to a query for its state which can be tested for being either free or busy. Most modules will respond to other queries that are specific to their operation, and those will be described in future units when needed. Below are some examples of querying the state 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
```

### *1.7.5.3 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
```

## 1.7.6 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.

Again, you can reset the model and work through its execution with the tutor mode of the Stepper tool. This time, you will also need to check the status of the **retrieval** buffer to determine when the fail production matches.

### 1.7.7 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.

## 1.8 The Addition Model

There is another example model included with the unit materials which is similar to the count model which 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. We will not cover that model in detail here. If you would like to examine and run that model, it is found in the "addition.lisp" file of unit 1, and you would use it the same way you loaded and ran the other models.

## 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 will hold the initial components of the task and 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 using the process described above and implemented 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 other models in this unit.

### *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 the value **busy**[10] 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.

[10]There is nothing special about the value busy in this model. It is simply being used to mark which step the model is currently performing. Any other symbol could be used in its place in these productions and the model would continue to work.

***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: GOAL-CHUNK0
GOAL-CHUNK0
   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.

## References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y . (2004). An integrated theory of the mind. *Psychological Review 111,* (4). 1036-1060.

Anderson, J. R. (2007). *How Can the Human Mind Occur in the Physical Universe?* New York: Oxford University Press.

# Unit 1: 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 about 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 usially cause things to stop until it is resolved by the user. 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 “>” or “]” character (depending on which version of the standalone you are running). The recommendation is to always resolve those errors before continuing, and usually it is best to just abort the error. If you have the “>” prompt for the error, then typing :pop will return the prompt back to the “?”. If you have the “]” prompt then you will want to type top instead to return to the main prompt.

## 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.

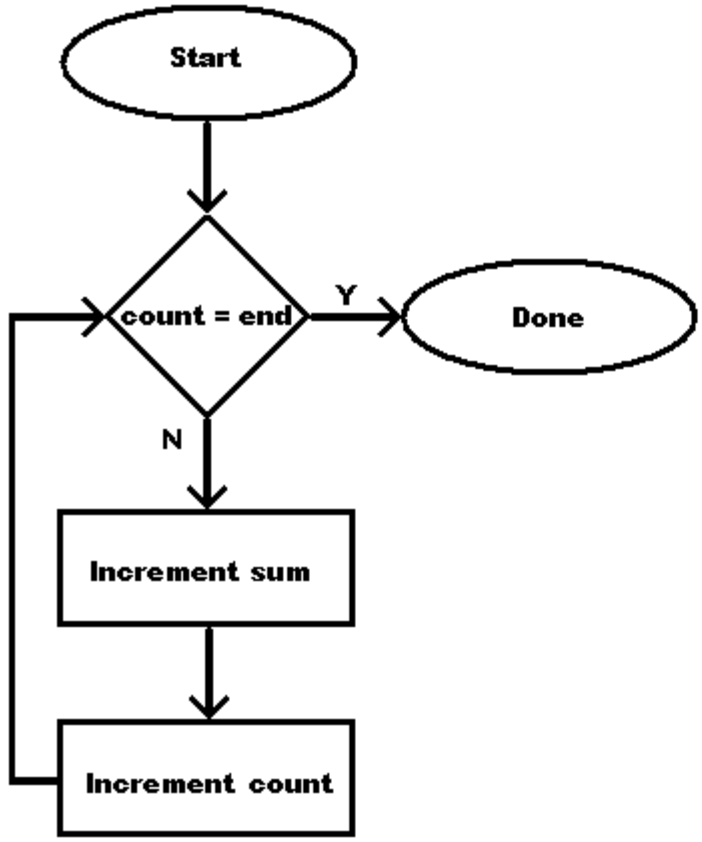
For this unit we will work through a broken version of the addition by counting model which is in the broken-addition.lisp file (a working version of the model is found in the addition.lisp file). Before starting the debugging process however, we will first look at the overall design of this model because 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 have some details that start like this (the output may vary in different versions of the standalone application):

```
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 you could also fix the problems one at a time, stopping here to fix the initialize-addition production and then reload and check the warnings at that point.

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 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 they may 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 the 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 of 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 before trying to use it to generate data from performing a task.

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:

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

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 of 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: GOAL-CHUNK0
GOAL-CHUNK0
   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: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  ONE
   ARG2  ZERO
   SUM  ONE
```

The result is 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 the run command we have been using, and it will allow you 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.] 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: GOAL-CHUNK0
GOAL-CHUNK0
   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 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-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
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.550   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.600   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.650   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.700   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.750   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.800   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.850   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     0.900   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.950   PROCEDURAL             PRODUCTION-FIRED INCREMENT-SUM
     1.000   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     1.000   ------                 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. Looking at the production it shows, 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. Here are the chunks in the goal and retrieval buffers:

```
? (buffer-chunk goal retrieval)
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  ONE
   ARG2  ONE
   SUM  TWO
   COUNT  ZERO

RETRIEVAL: RETRIEVAL-CHUNK0 [ZERO]
RETRIEVAL-CHUNK0
   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: GOAL-CHUNK0
GOAL-CHUNK0
   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: GOAL-CHUNK0
GOAL-CHUNK0
   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: GOAL-CHUNK0
GOAL-CHUNK0
   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:

```
 (six isa number number six next eight)
```

Looking at the other declarative memory chunks there is already a chunk for seven so all we need to do is change the value of next in chunk six to seven instead of eight:

```
 (six isa number number six next seven)
```

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: GOAL-CHUNK0
GOAL-CHUNK0
   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 now instead of just running the model itself one will have to run the corresponding task for the model to perform instead, and that task will handle the running of 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 all the tasks, and the experiment description document in each unit will provide additional details on how those tasks are implemented. From the model's perspective, it does not matter whether the Lisp or Python version of the task is used, and the single model file included with the tutorial will 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, 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 run that if you do not want to use the ACT-R Environment application.

### 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. There are two ways that can be done:

- You can use 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.
- You can call the actr-load[2] function directly from prompt specifying the location of that file. If the tutorial directory is still located with the rest of the ACT-R files from the standalone distribution then that would look like this:

```
? (actr-load "ACT-R:tutorial;lisp;demo2.lisp")
```

When you load that file it will also load the 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 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). For the examples in this tutorial we will assume that the directory containing the ACT-R Python modules (the tutorial/python directory) is either the current directory for the Python session or that it has been added to the Python search path, and we will also assume that the Python prompt is the three character sequence ">>>". Once your Python session is running there are two ways you can import the module:

- You can use import to directly import that module from the Python prompt:

[2]The Lisp load function could also be used, but that can vary based on the specific version of the software you are using and may require specifying the entire path instead of using the shortcut of "ACT-R:". The actr-load function works the same in all versions.

```
>>> import demo2
```

- If you first import the actr module instead, then you can use the "Import Python module" button in the Environment to pick a module to be imported into Python, and for this task the file is demo2.py in the python directory of the ACT-R tutorial.

Importing 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 the demo2 module, and it has one optional parameter which indicates whether a human 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:

```
>>> 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 that 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               SET-BUFFER-CHUNK-FROM-SPEC IMAGINAL
     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 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 step)
(chunk-type array letter)
```

The chunk-type read-letters specifies one slot which is called step 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 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 a chunk which contains 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 Step Slot

In this model, the step 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 step (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 step 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>
      ISA         read-letters
      step        attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      step        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 to the **imaginal** buffer will result in the imaginal module creating 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               SET-BUFFER-CHUNK-FROM-SPEC IMAGINAL
```

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 performs the action set-buffer-chunk-from-spec. This is similar to the set-buffer-chunk action we have seen previously. However, instead of setting the indicated buffer to hold a copy of a specific chunk as is done with the set-buffer-chunk action, that buffer is being set to hold a chunk which is based on a provided specification (shortened to spec in the action name), and in this case, that specification is the slots and values indicated in the request to the **imaginal** buffer.

Something to notice in the trace is that the buffer was not immediately set to have that chunk as a result of the request. It took .2 seconds before the chunk was made available in the buffer. 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, and that is done in the encode-letter production seen above:

```
   ?imaginal>
      state        free
```

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. Thus, for demonstration purposes, this model records that information in the **imaginal** buffer's chunk.

## 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. In the demo2 model, visual actions are performed in the 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 step slot (which is the value in the step slot of the chunk initially placed into the **goal** buffer):

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

It makes a request of the **visual-location** buffer and it changes the **goal** buffer chunk's step slot to the value find-location. It is important to note that those values for the setp slot of the **goal** buffer 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 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             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
```

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 of 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-LOCATION: VISUAL-LOCATION0-1 [VISUAL-LOCATION0]
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 any of those slots can be used to constrain the results. However, we will not need to do so for this unit and instead rely upon the ability of the vision module to remember visual features it has previously attended. Unit 3 will look in more detail at those slots and how to construct **visual-location** requests to search for items.

### *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 is similar to specifying a slot in the request, but it does not correspond to a slot in any chunk or chunk-type specification in the model. Request parameters always start with the ":" character which is what distinguishes it from a slot. They can be provided in a request to a buffer regardless of any chunk-type that is specified (or when no chunk-type is specified as is the case here), and they are used to supply additional information to the module about a request.

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 step 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
      step        find-location
   =visual-location>
   ?visual>
      state       free
==>
   +visual>
      cmd         move-attention
      screen-pos  =visual-location
   =goal>
      step        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 – the details of its slots and values do not matter. The other is a query of the **visual** buffer.

### 2.5.3 Testing 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** or **free**[3]. 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.

You can use the "Buffers" tool in the Control Panel to see the current status of a buffer's queries in addition to the chunk it contains. If you 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 information. That shows the standard queries for each buffer along with the current value (either **t** or **nil**) for such a query at this time as well as any additional queries which may be specific to that buffer (details on all of the queries for each buffer can be found in the reference manual).

[3]There is a third state query which modules will respond to called **error**, but that is rarely used since different modules use it to indicate different situations. The buffer failure query described in unit 1 is the more reliable and consistent test of a request to a buffer failing.

#### *2.5.3.1 Jamming a module*

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.

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 really necessary in this model 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 step slot. Thus there is no risk of jamming the imaginal module in this model, but omitting queries which appear to be unnecessary can be 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 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.

#### *2.5.3.2 Strict Safety*

In fact, like the strict harvesting mechanism described in unit 1 for automatically clearing buffers, there is also a "strict safety" mechanism which will automatically add a query for state free to a production that makes a request to a buffer which does not already have a query for that buffer (except for the **goal**, **retrieval**, and **visual-location** buffers which do not lead to jamming). That automatic query should prevent jamming in most cases, but explicitly adding queries to productions is still the recommended approach for readability of the model. The models in the tutorial will all include explicit queries and not rely upon the strict safety mechanism.

### 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 shown above for the chunk in the **visual-location** buffer.

### 2.5.5 Visual buffer

On the RHS of the **attend-letter** production 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 (remember that the condition specifying a buffer test also binds the variable naming the buffer to the chunk it contains). 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 chunk representing the visual object. Altogether, counting the two production firings (one to request the location and one to harvest the result and 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:

```
VISUAL: VISUAL-CHUNK0 [TEXT0]
VISUAL-CHUNK0
   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, like this one, the **value** slot will hold a string that contains the text encoded from the screen. We will describe some of the chunk-types that specify the slots for visual items below.

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

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

It 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 there is a chunk-type called visual-object which specifies the slots: screen-pos, value, status, color, height, and width. For the objects which the vision module encodes 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
      step        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, and later in the tutorial we will describe a way to avoid that redundancy and still maintain the safety of the chunk-type declaration.

## 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 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.

While this automatic re-encoding process of the vision module is a useful component of the module, it does require that you be careful when writing models that process changing displays. In particular, since the module is **busy** while the re-encoding is occurring, the vision module cannot handle a new attention shift. This is one reason why 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.

## 2.6 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
      step        respond
   =imaginal>
      isa         array
      letter      =letter
   ?manual>
      state       free
==>
   =goal>
      step        done
   +manual>
      cmd         press-key
      key         =letter
)
```

This production fires when the letter slot of the chunk in the **imaginal** buffer has a value, the **goal** buffer chunk's step 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 step 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[4], 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

[4]Even though the motor module does not put chunks into its buffer, the procedural module still performs the clear action since it treats all buffers the same because it does not know the details of how other modules operate.

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 someone 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

As mentioned in unit 1, the "strict harvesting" mechanism will implicitly clear a chunk from a buffer if the buffer is tested on the LHS of a production and that buffer is not modified on the RHS of the production. 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
```

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
      step        attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      step        respond
   +imaginal>
      isa         array
      letter      =letter
   =visual>
)
```

The strict harvesting mechanism also applies to the buffer failure query. A production which makes a query for buffer failure will also result in an automatic clearing of the buffer. That is done because clearing the buffer also clears the failure status, and having that automatically cleared makes it easier to write a model that handles failure conditions since the query will be false immediately after the production that tested it fires preventing it from firing again until there is another failure.

## 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 this model a good example for showing the visual and motor actions. The :trace-detail parameter was described in the unit 1 code document and setting it to high ensures that all of the details are shown in the trace. The others are described in detail in the code document for this unit, but we will describe the :v (verbose) parameter briefly here since it is an important one. The :v parameter controls whether the trace of the model is output. If :v is **t** then the trace is displayed and if :v is set to **nil** the trace is not output. The software can run significantly faster when the trace is turned off, and that will be important in later units when we are running the models through the experiments multiple times to collect data.

## 2.10 Unit 2 Assignment

Your assignment is to write a model which uses the visual and motor capabilities introduced in the demo2 model to perform a slightly more complex experiment. The new experiment presents three letters on the screen, and 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".

Alternatively, if you have imported the actr module into Python you can use the "Import Python module" button in the Environment to pick a module to be imported into Python, and for this task the file is unit2.py in the python directory of the ACT-R tutorial.

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 (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 step:

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

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

```
 (goal isa read-letters step 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, but for this assignment your model should follow the recommendation that any control information it uses is in the **goal** buffer and the problem representation is kept in the **imaginal** buffer. Otherwise, as long as the model is always correct at the task that is sufficient for this assignment.

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 in a way that is comparable to how 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.

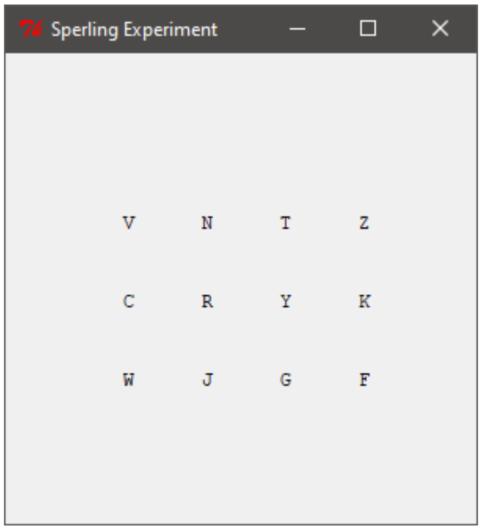
Kieras, D. & Meyer, D.E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction, 12,* 391-438.

# Unit 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

Here is the window showing the initial screen for the example task in this unit.



When a display like that is generated using the ACT-R GUI interface tools (AGI) and is presented to a model, a representation of all the visual information in the window is recorded in a visual icon maintained by the vision module of the model. 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):

```
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 for all of the features that are available for the model to find. That information is a combination of the internal attentional status and the information from chunks that represent the locations and the objects at those locations. The specific features of an item can vary, and they are based on the source of those features. The features shown above: text, 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

The features shown in the visicon are the items searched when a **visual-location** request is made. When requesting the visual location of an object any of the features available in the visicon may be used in the request. If a feature matches the request, then a chunk with that feature's location information is placed into the **visual-location** buffer. If there are no objects which match the request, then a **failure** will be signaled for the **visual-location** buffer. If there is more than one item in the visicon that matches the request, the one most recently added to the visicon (the newest one) will be chosen. If multiple items also match on their recency, then one will be picked randomly among those items.

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 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
     step     find
==>
   +visual-location>
     :attended nil
   =goal>
     step    attend)
```

In its action, this production requests the visual location of an object that has not yet been attended. Otherwise, it places no preference on the location to be found. 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 of a finst 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 status 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 number of finsts 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 (described in the code document for this unit). Thus, one solution is to just set those parameters to values which are large enough so that the model can just rely on finsts to do the task. However, changing architectural parameters like those is not recommended without a good reason since the default values typically represent a human performance constraint. Also, 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 (other devices may use different slots to represent locations):

```
  (chunk-type visual-location screen-x screen-y distance kind color value height
width size)
```

Those slots hold the location information for the features in the visicon shown above and that chunk-type could be used to declare the desired slots when making a 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, which is also measured in pixels, represents the distance to the screen. The default distance from the model to the screen is 15 inches (1080 pixels assuming a screen that has 72 pixels per inch), but there are parameters which can change the distance and the pixels per inch setting.

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 representing the color.

The kind slot specifies a general classification of the item, like text or line, which are also chunks created by the vision module.

The value slot of the chunk placed in the **visual-location** buffer will be a general representation of the item, just like the kind slot. The information shown for the value in the visicon is actually the information from the visual object at the location. That information can be used when making the request for a visual location, but the value itself will not be not available to the model until it attends to the item i.e. the model can request a visual location which has a value of "v" on the display, but the model must move its attention to that location using a **visual** request to actually encode the letter "v".[1]

### 3.1.4 Visual-location request specification

A **visual-location** request may contain any slots which have been created for visual features (both the location and object chunks). Any of the slots may be specified any number of times, and one can use any of the possible slot modifiers (-, <, >, <=, or >=) on the slots. If a location is found that matches all of the constraints provided then a chunk representing that location will 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. This example represents a request to find something that is black and not text at a specific position:

```
+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 Production variables*

Variables from the production can be used in 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:

[1]Finding locations by value is not a recommended practice because in most tasks that would represent a super human ability, but it is available to use. There are many places where the software allows one to use abilities that are not constrained to human performance in the models for convenience or flexibility purposes – not every modeling task requires matching all aspects of human performance.

```
  +visual-location>
    kind     =kind
    screen-x =x
    screen-y =y
```

### *3.1.4.3 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
```

### *3.1.4.4 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 a **visual-location** request for 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 a line of text 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 features 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 resulting from the most recent **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.

### 3.1.5 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. 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 which do not depend upon finsts to be able to search all available features.

## 3.2 The Sperling Task

The example model for this unit can perform the partial report version of 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 briefly see displays of 12 letters arranged in a 4 x 3 grid like those shown in the image at the beginning of this unit. The subject is cued sometime after the display comes on as to which of the three rows of letters 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 cuing, 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 50ms 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 for that 1 second before responses are required. 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 for the model.

One thing you may notice when looking at this model is that it does not use the **imaginal** buffer, as described in the previous unit, to hold the problem representation separate from the control information. 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-FROM-SPEC GOAL
     1.110   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL VISUAL-CHUNK0-0
     1.160   PROCEDURAL             PRODUCTION-FIRED DO-REPORT
     1.160   MOTOR                  PRESS-KEY KEY C
     1.160   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL VISUAL-CHUNK0-4
     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 actions of 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 typing of the letters which the model remembers having attended from the target row.

## 3.3 Visual Attention

like 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
     step       attending
   =visual-location>
     isa        visual-location
   > screen-y   450
   < screen-y   470
   ?visual>
     state      free
==>
   =goal>
     location   medium
     step       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 that fired 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 a mechanism that allows the perceptual modules to provide an environment driven, or bottom-up, attention ability. 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 set-buffer-chunk 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 visual scene 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, but 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. The vision module also helps to avoid a chunk staying in the buffer because it will remove a chunk that it has stuffed into the **visual-location** buffer after .5 seconds if it is still there.

### 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 can be 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 dependent 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
     step      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 AGI defines this chunk-type which has the slots it uses to create the chunks for the **visual** buffer:

```
  (chunk-type visual-object screen-pos value status color height width)
```

but the AGI does not actually use the status slot when it creates the chunk and leaves that slot available for the modeler to use as needed. We do 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 step slot, and it 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 step 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 uses 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** buffer holds the location of an auditory percept 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 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 query to the **aural** buffer for state free is true then 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 for the built-in sounds, 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 with the event information 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 a section of the high detail trace for the same 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   PROCEDURAL             PRODUCTION-SELECTED ENCODE-ROW-AND-FIND
     0.150   AUDIO                  new-sound
     0.185   PROCEDURAL             PRODUCTION-FIRED ENCODE-ROW-AND-FIND
...
     0.185   PROCEDURAL             PRODUCTION-SELECTED ATTEND-HIGH
     0.200   AUDIO                    SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0
NIL
     0.235   PROCEDURAL               PRODUCTION-FIRED ATTEND-HIGH
...
     0.235   PROCEDURAL               PRODUCTION-SELECTED DETECTED-SOUND
...
```

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. That leads to the following actions:

```
...
     0.285   PROCEDURAL               PRODUCTION-FIRED DETECTED-SOUND
...
     0.285   AUDIO                    ATTEND-SOUND AUDIO-EVENT0-0
...
     0.320   PROCEDURAL               PRODUCTION-SELECTED ENCODE-ROW-AND-FIND
     0.370   PROCEDURAL               PRODUCTION-FIRED ENCODE-ROW-AND-FIND
...
     0.370   PROCEDURAL               PRODUCTION-SELECTED ATTEND-LOW
     0.420   PROCEDURAL               PRODUCTION-FIRED ATTEND-LOW
...
     0.505   PROCEDURAL               PRODUCTION-SELECTED ENCODE-ROW-AND-FIND
     0.555   PROCEDURAL               PRODUCTION-FIRED ENCODE-ROW-AND-FIND
...
     0.555   PROCEDURAL               PRODUCTION-SELECTED ATTEND-HIGH
     0.570   AUDIO                    AUDIO-ENCODING-COMPLETE AUDIO-EVENT0
     0.570   AUDIO                    SET-BUFFER-CHUNK AURAL TONE0
     0.605   PROCEDURAL               PRODUCTION-FIRED ATTEND-HIGH
...
     0.605   PROCEDURAL               PRODUCTION-SELECTED SOUND-RESPOND-MEDIUM
     0.655   PROCEDURAL               PRODUCTION-FIRED SOUND-RESPOND-MEDIUM
...
     0.690   PROCEDURAL               PRODUCTION-SELECTED ENCODE-ROW-AND-FIND
...
```

When the **detected-sound** production completes at .285 seconds the aural request is made to shift attention to the sound. The next audio module event in the trace occurs at time .57 seconds when the module completes encoding the sound and then puts the chunk in the **aural** buffer. So it took .285 seconds from when the request was made until the sound was attended and encoded. 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. 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 when it does, 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 of 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 an item has been retrieved and the motor module is free. As actions, it presses the key corresponding to the letter retrieved and makes a **retrieval** request for another letter. Notice that it does not modify the chunk in the **goal** buffer (which 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 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 VISUAL-CHUNK0-4
     1.760   PROCEDURAL             PRODUCTION-FIRED DO-REPORT
...
```

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 causing the run to terminate.

## 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:

```
(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)
```

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 default is four declarative finsts which last 3 seconds each, but those can be changed with parameters described in the code document for the unit. 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 since the focus is on visual and aural perception.

## 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 experiment 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 of the events as they are occurring, which is done by setting the :v parameter in the model to **nil** instead of **t**:

```
(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:  0.996
MEAN DEVIATION:  0.139

Condition    Current Participant   Original Experiment
 0.00 sec.           3.09                  3.03
 0.15 sec.           2.47                  2.40
 0.30 sec.           2.11                  2.03
 1.00 sec.           1.75                  1.50
```

```
>>> sperling.experiment(100)
CORRELATION:  0.993
MEAN DEVIATION:  0.208

Condition    Current Participant   Original Experiment
```

```
 0.00 sec.          3.25                 3.03
 0.15 sec.          2.57                 2.40
 0.30 sec.          2.14                 2.03
 1.00 sec.          1.79                 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 a difference in the code that implements the different versions of the task, but because each run of the model varies there is noise in the model's data which results in slightly different average performance each time the experiment is run.

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 the 0 key. 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 control 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         1.00  (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         2.41  (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 it shows 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 reflects the ability to perceive small numbers of objects as familiar patterns without needing to count them (which is called subitizing) 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 significantly more productions and likely require 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 step start)
        (start))
```

In addition to the number and next slots that were 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 defines a chunk-type which can be used for maintaining control information in the goal buffer:

```
(chunk-type count count step)
```

It has a slot to maintain the current count and a slot to hold an indication of the current step. An initial chunk named goal which has a step 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 to make it easier to focus on the visual portion of the modeling if you like. 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 4: Activation of Chunks and Base-Level Learning

## 4.1 Subsymbolic Components

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, the request would fail. That 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. Those components are an activation value associated with chunks and a utility value associated with productions. To use the subsymbolic components of ACT-R 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 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*.

$\varepsilon_i$: A 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:

$$B_i = \ln\left(\sum_{j=1}^{n} t_j^{-d}\right)$$

***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 described 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[1].

### 4.3.3 Optimized Learning

Because of the need to separately calculate the effect of each presentation, the base-level activation equation 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 of the base-level activation equation that can be used 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 instead:

$$B_i = \ln\left(\frac{n}{1-d}\right) - d * \ln(L)$$

***n*:** The number of presentations of chunk *i*.

***L*:** The lifetime of chunk *i* (the time since its creation).

***d*:** The decay parameter.

## 4.4 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, $\sigma^2$, is related to the *s* value by this equation:

[1]Additional details about this process will be described in the next unit.

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

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**).

## 4.5 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, $\tau$. 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 $\tau$ 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.

## 4.6 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 and 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:

$$Time = Fe^{-\tau}$$

$\boldsymbol{\tau}$: The retrieval threshold.

***F***: The latency factor.

## 4.7 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-FROM-SPEC IMAGINAL
     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-FROM-SPEC IMAGINAL
     10.385   PROCEDURAL             CONFLICT-RESOLUTION
     11.145   DECLARATIVE            RETRIEVED-CHUNK IMAGINAL-CHUNK0-0
     11.145   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL IMAGINAL-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 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:

```
IMAGINAL-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 new request to the **visual** buffer.

### 4.7.1 Stop Visually Attending

If you do not want the model to re-encode the information at the location it is currently attending you can request that it stop attending to the visual scene. That is done with a request to the **visual** buffer specifying a cmd value of clear:

```
+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.

### 4.7.2 The Retrieval Request

Now, consider the **retrieval** request in the read-probe production again:

```
    +retrieval>
      isa      pair
      probe    =val
```

In response to that request 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 and 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 when the buffer is cleared 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:  0.996
MEAN DEVIATION:  0.099
Trial    1       2       3       4       5       6       7       8
        0.000   2.173   1.856   1.682   1.531   1.435   1.362   1.289

Accuracy:
```

```
CORRELATION:  0.994
MEAN DEVIATION:  0.042
Trial    1       2       3       4       5       6       7       8
       0.000   0.555   0.764   0.858   0.905   0.935   0.952   0.964
```

## 4.8 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 (or setting any other declarative parameters) produces results like this:

```
Latency:
CORRELATION:  0.922
MEAN DEVIATION:  0.283
Trial    1       2       3       4       5       6       7       8
       0.000   1.548   1.548   1.547   1.550   1.554   1.548   1.550

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 0.5 produces 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:  0.961
MEAN DEVIATION:  1.124
Trial    1       2       3       4       5       6       7       8
```

```
        0.000   3.691   3.561   3.492   2.727   2.305   2.047   1.880

Accuracy:
CORRELATION:  0.880
MEAN DEVIATION:  0.220
Trial    1       2       3       4       5       6       7       8
        0.000   0.100   0.260   0.925   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, which as described earlier determines how active a chunk has to be to be retrieved 50% of the time, is set at -2. The instantaneous activation noise s value is set at 0.5. This determines how quickly probability of retrieval changes as we move past the threshold. The latency factor, which determines the magnitude of the activation effects on latency, is set at 0.4. 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 and running the model to see the effect that they have on the model and data. There are other ways of determining parameters that can be used, but we will not be covering any such mechanisms in the tutorial.

## 4.9 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 IMAGINAL-CHUNK0-0 matches
Computing activation for chunk IMAGINAL-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 -0.7636829
Adding permanent noise 0.0
Chunk IMAGINAL-CHUNK0-0 has an activation of: -0.8752547
Chunk IMAGINAL-CHUNK0-0 has the current best activation -0.8752547
Chunk IMAGINAL-CHUNK0-0 with activation -0.8752547 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.

## 4.10 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.

## 4.11 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 the 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:  0.292
MEAN DEVIATION:  1.539

              2 (64)      3 (64)      4 (64)
Block  1  2.408 (64)  3.045 (64)  3.651 (64)
Block  2  2.403 (64)  3.048 (64)  3.639 (64)
Block  3  2.398 (64)  3.045 (64)  3.643 (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 starting model already creates chunks with the correct answer for a problem it solves by counting. A completed problem for a trial where the stimulus was "A+2 = C" would look like this:

```
IMAGINAL-CHUNK0-0
   RESULT  C
   ARG1  A
   ARG2  TWO
```

The result slot contains the result of counting to solve the addition problem that was presented – not the letter that was presented as a possible answer.

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. It counts as many letters as indicated to determine the correct result, and stores that in the result slot of the **imaginal** buffer. Then it compares the counted result to the presented answer to decide how to respond. Thus the same chunk will result from a trial where the stimulus presented is "A+2 = D" because it 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. 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:  0.949
MEAN DEVIATION:  0.642

             2 (64)      3 (64)      4 (64)
Block  1  1.334 (64)  1.312 (64)  1.528 (64)
Block  2  1.030 (64)  1.019 (64)  1.031 (64)
Block  3  0.997 (64)  0.994 (64)  1.005 (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 and gain some experience with how it affects the model's performance. Here is our fit to the data adjusting only the retrieval threshold:

```
CORRELATION:  0.989
MEAN DEVIATION:  0.117

             2 (64)      3 (64)      4 (64)
Block  1  1.972 (64)  2.383 (64)  2.756 (64)
Block  2  1.359 (64)  1.566 (64)  1.601 (64)
Block  3  1.106 (64)  1.221 (64)  1.334 (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 to gain some experience with the effects they have. 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:  0.993
MEAN DEVIATION:  0.071

              2 (64)      3 (64)      4 (64)
Block  1  1.977 (64)  2.374 (64)  2.805 (64)
Block  2  1.262 (64)  1.482 (64)  1.505 (64)
Block  3  1.095 (64)  1.154 (64)  1.219 (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 in 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 will 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=e, 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  2.395 ( 8)  3.045 ( 8)  3.657 ( 8)

>>> zbrodoff.block()

              2 (64)      3 (64)      4 (64)
Block  1  2.392 (64)  3.045 (64)  3.650 (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)

>>> zbrodoff.experiment()


              2 (64)      3 (64)      4 (64)
Block  1  2.406 (64)  3.045 (64)  3.643 (64)
Block  2  2.395 (64)  3.043 (64)  3.658 (64)
Block  3  2.392 (64)  3.047 (64)  3.643 (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 runs to average:

```
? (zbrodoff-compare 10)

>>> zbrodoff.compare(10)

CORRELATION:  0.288
MEAN DEVIATION:  1.540

              2 (64)      3 (64)      4 (64)
Block  1  2.406 (64)  3.045 (64)  3.643 (64)
Block  2  2.395 (64)  3.043 (64)  3.658 (64)
Block  3  2.392 (64)  3.047 (64)  3.643 (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 of 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 any warnings that may be reported when the model file is loaded and what is happening in your model when it runs. 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. 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 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*.

$\varepsilon$: 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 $W_k / n_k$ where $n_k$ is the number of slots in the chunk in buffer *k* which contain values that are chunks.

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)

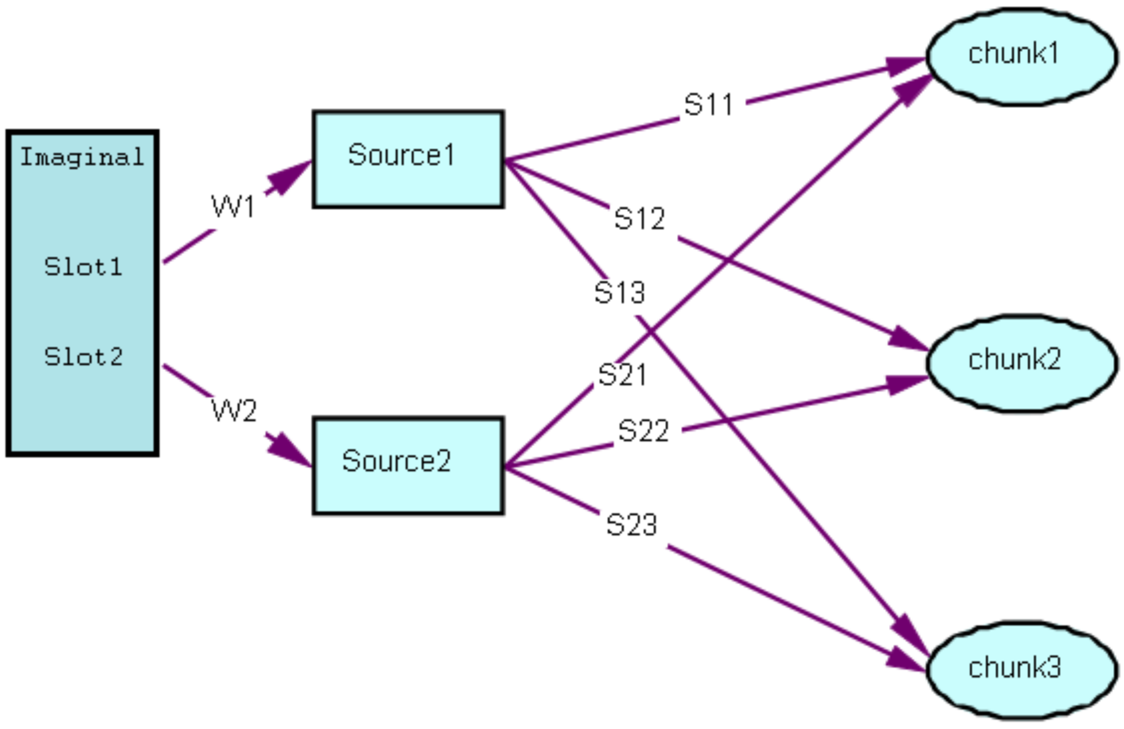
***fan$_j$*:** is the number of chunks in declarative memory in which *j* is the value of a slot plus one for chunk *j* being associated with itself.[1]

By default, only the **imaginal** buffer serves as a source of activation. The $W_{imaginal}$ value defaults to 1 and all other buffers have a default weight of 0.[2] Therefore, in the default case, the activation equation can be simplified to:

$$A_i = B_i + \sum_j \frac{1}{n} S_{ji} + \varepsilon$$

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:

[1]This is 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.

[2]The reference manual indicates the parameter for each buffer that will change its spreading weight value.

$$A_1 = B_1 + W_1S_{11} + W_2S_{21}$$
$$A_2 = B_2 + W_1S_{12} + W_2S_{22}$$
$$A_3 = B_3 + W_1S_{13} + W_2S_{23}$$

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 $S_{ji}$ 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 referred to as their fan in the experiment. The following tables show the recognition latencies from the experiment in seconds for targets and foils as a function of person and location fans:

```
                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 to account for the initial study). 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. 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. Those functions take 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 can be run to produce the trace below for the sentence “The lawyer is in the store” which is true (it is in the study set):

```
? (fan-sentence "lawyer" "store" t 'person)
>>> fan.sentence('lawyer','store',True,'person')
```

```
     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-FROM-SPEC IMAGINAL
     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-PERSON
     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
```

The model can also 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 much 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 left at 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 have 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 productions (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 currently free.

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: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   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, and 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 on only one of the items from the probe sentence. By doing so it ensures that one of the study sentences will be always be retrieved instead of resulting in a retrieval failure for the foil trials. 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 of a retrieval failure 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: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   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 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 (see the code document for a note about this):

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   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 IMAGINAL-CHUNK0
    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
     0.839   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P13
     0.839   PROCEDURAL             CONFLICT-RESOLUTION
```

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 0.907 which 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. The activation noise for chunks is turned off in this model to make it easier to see the spreading activation effects. Thus the activation of chunk p13 is:

$$A_i = B_i + \sum_j \frac{1}{n} S_{ji} + \varepsilon$$

$$A_{p13} = 0 + [(.5 * .907) + (.5 * .907)] + 0 \ = .907$$

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 0.26 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.

**5.4.2 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 is the activation trace for that situation (like the :trace-detail parameter, the :act parameter can be set to different values to change the amount of information provided and here we have set it to medium to reduce the output slightly):

```
     0.585   DECLARATIVE            start-retrieval
Chunk P3 matches
Chunk P2 matches
Chunk P1 matches
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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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
     0.585   PROCEDURAL             CONFLICT-RESOLUTION
     1.026   DECLARATIVE            RETRIEVED-CHUNK P3
     1.026   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P3
     1.026   PROCEDURAL             CONFLICT-RESOLUTION
```

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:

```
     0.585   DECLARATIVE            start-retrieval
Chunk P6 matches
Chunk P3 matches
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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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
     0.585   PROCEDURAL             CONFLICT-RESOLUTION
     1.026   DECLARATIVE            RETRIEVED-CHUNK P3
     1.026   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P3
     1.026   PROCEDURAL             CONFLICT-RESOLUTION
```

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 due to the higher fan of the source elements resulting in lower $S_{ji}$ values. 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.

### 5.4.3 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 is the medium activation trace when retrieve-from-person is chosen:

```
     0.585   DECLARATIVE            start-retrieval
Chunk P10 matches
Chunk P9 matches
Chunk P8 matches
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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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
     0.585   PROCEDURAL             CONFLICT-RESOLUTION
     1.151   DECLARATIVE            RETRIEVED-CHUNK P10
     1.151   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P10
     1.151   PROCEDURAL             CONFLICT-RESOLUTION
```

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:

```
     0.585   DECLARATIVE            start-retrieval
Chunk P6 matches
Chunk P3 matches
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 IMAGINAL-CHUNK0
    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 IMAGINAL-CHUNK0
    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
     0.585   PROCEDURAL             CONFLICT-RESOLUTION
     1.075   DECLARATIVE            RETRIEVED-CHUNK P3
     1.075   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL P3
     1.075   PROCEDURAL             CONFLICT-RESOLUTION
```

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 fan 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 values in the slots of the retrieval request and the values 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:

$$A_i = B_i + \sum_k \sum_j W_{kj} S_{ji} + \sum_l PM_{li} + \varepsilon$$

$B_i$, $W_{kj}$, $S_{ji}$, and $\varepsilon$ have been discussed previously. The new term is the partial matching component.

***Specification elements l*:** The 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 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 the activation is decreased. To demonstrate partial matching we will look at two example models.

## 5.6 Grouped Recall

The first of these models is called grouped and found in the grouped-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 only 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 transposed 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 the one 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

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
```

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
```

That means 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 about -.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 of 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
```

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 matching 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 showing the proportion of possible answers given to each problem:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Other |
|---|---|---|---|---|---|---|---|---|---|---|
| 1+1 | 0.00 | 0.05 | 0.86 | 0.00 | 0.02 | 0.00 | 0.02 | 0.00 | 0.00 | 0.06 |
| 1+2 | 0.00 | 0.04 | 0.07 | 0.75 | 0.04 | 0.00 | 0.02 | 0.00 | 0.00 | 0.09 |
| 1+3 | 0.00 | 0.02 | 0.00 | 0.10 | 0.75 | 0.05 | 0.01 | 0.03 | 0.00 | 0.06 |
| 2+2 | 0.02 | 0.00 | 0.04 | 0.05 | 0.80 | 0.04 | 0.00 | 0.05 | 0.00 | 0.00 |

```
2+3  0.00  0.00  0.07  0.09  0.25  0.45  0.08  0.01  0.01  0.06
3+3  0.04  0.00  0.00  0.05  0.21  0.09  0.48  0.00  0.02  0.11
```

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:  0.947
MEAN DEVIATION:  0.066
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.09  0.67  0.20  0.02  0.00  0.00  0.00  0.00  0.02
1+2  0.00  0.00  0.18  0.64  0.09  0.01  0.00  0.00  0.00  0.08
1+3  0.00  0.00  0.00  0.14  0.78  0.05  0.00  0.00  0.00  0.03
2+2  0.00  0.00  0.01  0.17  0.78  0.03  0.00  0.00  0.00  0.01
2+3  0.00  0.00  0.00  0.00  0.25  0.50  0.05  0.00  0.00  0.20
3+3  0.00  0.00  0.00  0.00  0.01  0.02  0.62  0.14  0.01  0.20
```

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 below. There is also a description of how the parameters were adjusted to fit the data in the code document for the unit which may be helpful to go through.

### 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:

```
   *goal>
      ISA         count-from
      count       =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 which module performed an action 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 (assuming the initial chunk was set appropriately and ignoring the retrieval request since all we care about here is the **imaginal** buffer action):

```
(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 but for this example we are relying on the strict safety mechanism to add it automatically):

```
(p start
   =imaginal>
      ISA      count-from
      start    =num1
      count    nil
 ==>
   *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 uses the value from the sum slot of that chunk in a modification request to 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 is necessary because 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 when that is done 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 (thus the reason for calling it indirect). 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 receive no penalty to its activation while all of 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.

## 5.8 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 and 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 the 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 each 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)
    - **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)
    - none of the other slots will be set in the chunk
- After the 10 seconds for deciding have expired and player responses processed
    - **state** slot will be set to the value **results**
    - **mc1**, **mc2**, **mstart**, **oc1**, and **ostart** slots will be the same values as at the start of the hand
    - **mc3** slot
        - if the model hits
            - the value of the model's third card (a number from 1-10)
        - if the model stays
            - the slot will not be set
    - **mtot** slot will hold the 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** slot
        - if the opponent hits
            - the value of the opponent's third card (a number from 1-10)
        - if the opponent stays
            - the slot will not be set
    - **otot** slot will be the 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 known 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.

The productions in the starting model are fairly straight forward and it should be clear what they are doing. However, there is one action in the remember-game production which has not been used previously in the tutorial:

```
(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>)
```

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 when it was the only choice.

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 so that it will be able 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 of 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 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 at the end of the hand 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 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)
(2 3 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:  6 10   -> 16 (lose)   Opponent: 10  1 -> 21 (win )
Model:  2  6  3  -> 11 (lose)   Opponent:  6  5  9 -> 20 (win )
Model:  5  9  7  -> 21 (win )   Opponent:  3  6 10 -> 19 (lose)
Model:  6  8  6  -> 20 (win )   Opponent:  2 10  3 -> 15 (lose)
[2, 2, 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:

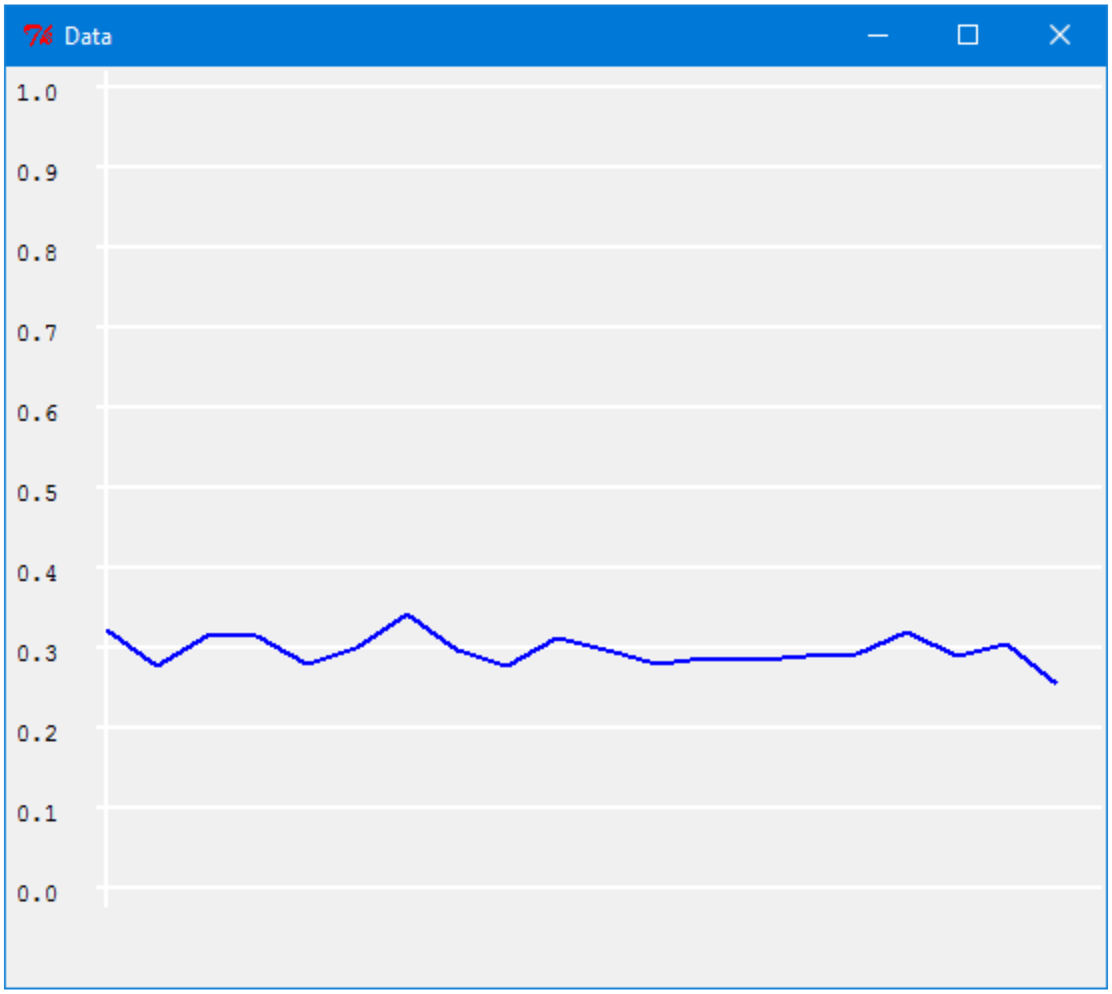
```
? (onehit-blocks 2 5)
((2 3 0 0) (3 2 0 0))

>>> onehit.blocks(2,5)
[[1, 4, 0, 0], [2, 3, 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 50)
((0.30159998 0.30800003 0.3 0.3152) (0.332 0.264 0.308 0.336 0.268 0.324 0.328 0.28
0.34 0.268 0.308 0.3 0.31199998 0.292 0.28800002 0.296 0.344 0.3 0.332 0.304))
>>> onehit.learning(50)
[[0.2976, 0.30639999999999995, 0.2992, 0.3152], [0.33599999999999997, 0.268, 0.304,
0.32, 0.26, 0.32799999999999996, 0.32, 0.284, 0.33999999999999997, 0.26, 0.304, 0.3,
0.32, 0.288, 0.284, 0.292, 0.344, 0.3, 0.32799999999999996, 0.312]]
```



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)
```

```
>>> onehit.learning(100,False)
```

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.

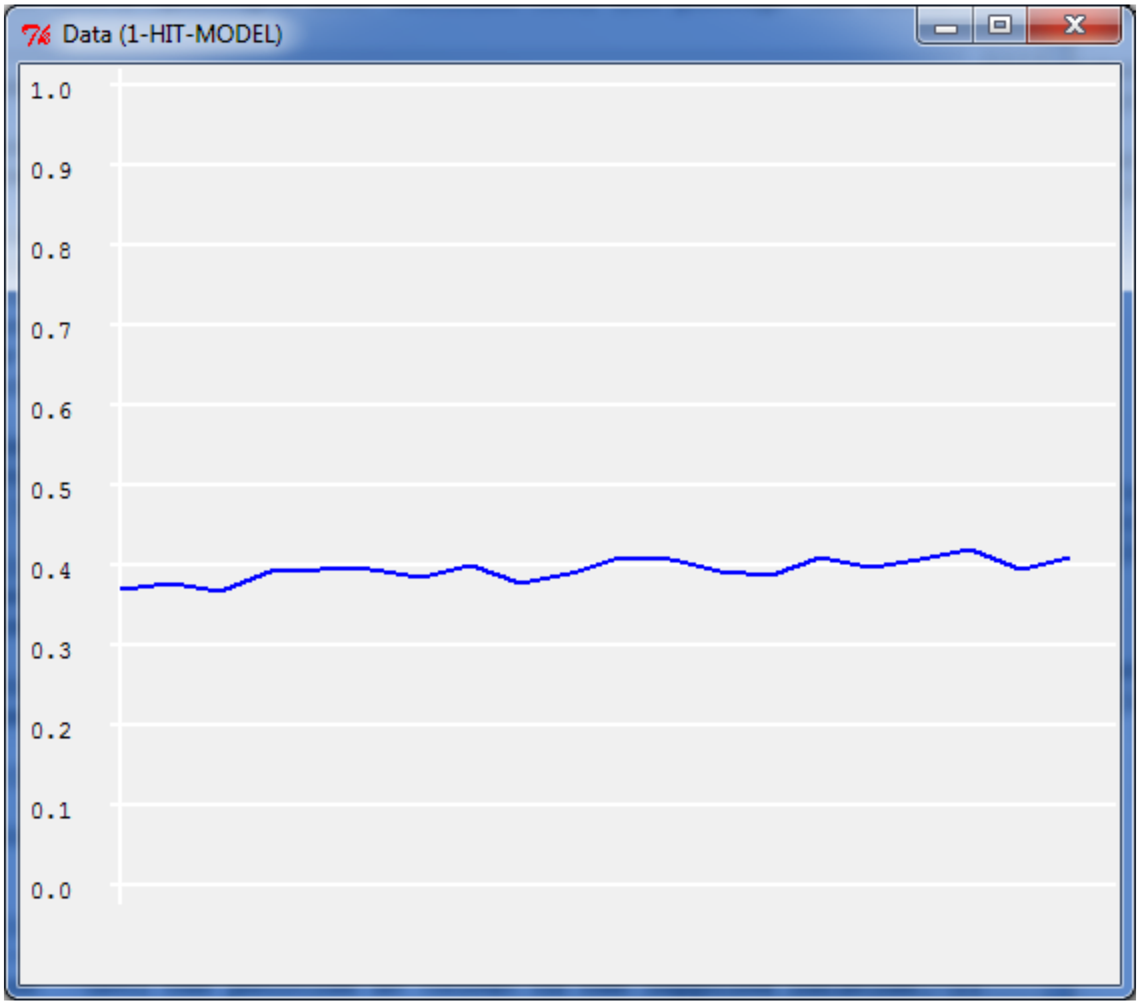
There is one other function which can be used to run the model called play-against-model in Lisp and play_against_model in the onehit module of Python. This function is similar to the one for playing hands except that instead of running an opponent from code it opens a window like the one shown below 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. 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.

### 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.



That model 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.