Unit 1: Introduction to ACT-R

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

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

1.1 Knowledge Representations

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

1.1.1 Chunks in ACT-R

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

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

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

1.1.2 Productions in ACT-R

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

IF the goal is to classify a person

  and he is unmarried

THEN classify him as a bachelor

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

  and d1 + d2 = d3

THEN create a goal to write d3 in the column

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

1.2 The ACT-R Architecture

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

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

1.2.1 Goal Module

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

1.2.2 Declarative Module

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

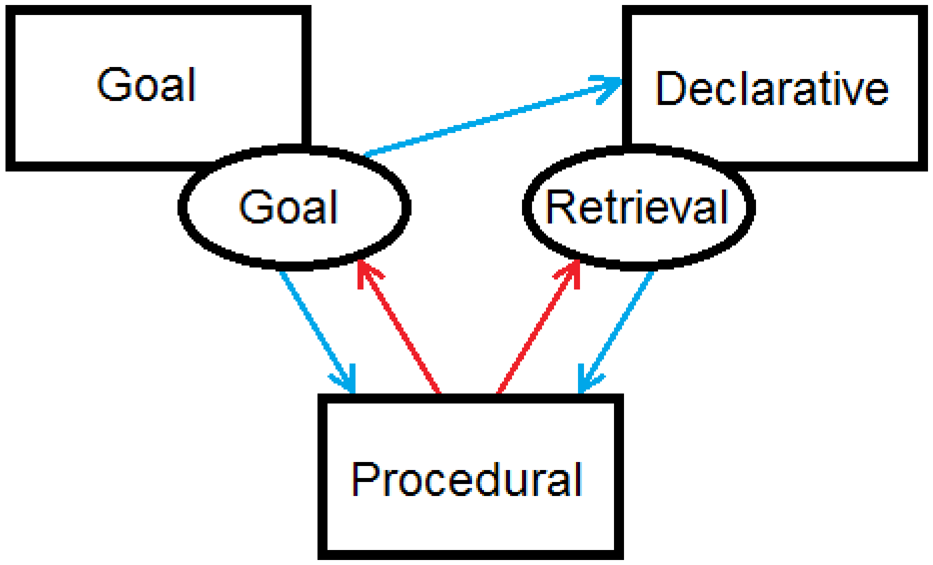
The declarative memory in the model consists of the chunks which are placed there initially by the modeler when defining the model and the knowledge which it learns as it runs. The learned knowledge is collected from the buffers of all of the modules. The declarative module monitors all of the buffers,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

1.2.3 Procedural Module

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

1.2.4 Overview

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



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 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 how one creates and runs a model using the ACT-R software. The ACT-R software is implemented in the ANSI Common Lisp programming language. It is distributed both as source code which can be loaded into any ANSI Common Lisp system and as applications for Windows and Mac OS X which can be run without having any Lisp software installed. It also contains an optional GUI for inspecting and debugging models called the ACT-R Environment. 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 runs in Lisp, but being able to program in Lisp will be useful for creating new tasks or experiments for an ACT-R model to perform. All of the example models in the tutorial units after this one will involve some Lisp code which implements the task the model performs, and the experiment text for each unit of the tutorial (the one with " _exp" on the end of the name) describes how that unit's experiments interact with the ACT-R software.

1.3.1 Interacting with Lisp

For those not familiar with Lisp, here is some basic information on how to use it for working with ACT-R. Lisp is an interactive language and provides a prompt at which the user can issue commands and evaluate code. 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 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. It would look like this if the prompt character is ">" (different Lisps may use different prompt characters):

```
> (+ 3 4)
7
>
```

It is also possible to create a file which contains commands which can be loaded into Lisp, and below we will describe how that is used to build a model in ACT-R.

1.3.2 Starting ACT-R

Here we will describe how to use the ACT-R source code distribution. [If you want to run the "standalone" ACT-R application then you will need to consult the documentation included with that software for how to install and run it.] The first thing you need to do is run your Lisp application. Once it is started, if you have a Lisp with a GUI then there should be an option, typically under the File menu, to load a file which you can use, but if you are using a command line Lisp you will need to execute the load command to load the file. The file you need to load is "load-act-r.lisp" found at the top level of the ACT-R source file distribution. At the prompt that would look like this:

```
(load "<full path>/load-act-r.lisp")
```

where <full path> is replaced with the appropriate location for the file.

Loading that file will compile and load all of the files necessary for using ACT-R and may result in a lot of messages being printed by Lisp as it does so. When it is finished loading it will print out several lines of text indicating the version information for all of the ACT-R components. Once that completes, ACT-R is ready to use. However, we do recommend that one also use the ACT-R Environment when working through the tutorial, and there are some additional steps necessary to run that.

1.3.2.1 Running the ACT-R Environment

The ACT-R Environment is a separate application which is written in the Tcl/Tk language which can be connected to ACT-R. It does not work with every Lisp that is capable of running ACT-R, but it will run with many of them and the details of which are compatible can be found in the file called QuickStart.txt in the docs directory of the ACT-R distribution. In some Lisps, an ACT-R command called **run-environment** is available which is all that is necessary to start the ACT-R Environment application and connect it to ACT-R. The command does not require any parameters so you can try that by entering this at the Lisp prompt:

```
(run-environment)
```

If that responds with a warning that the command is not available then you will have to perform the following steps to run and then connect to the ACT-R Environment. The first step is to run the Environment application. For Windows and Mac OS X the application can be found in the environment directory of the ACT-R distribution and is called either "Start Environment" or "Start Environment OSX" respectively. For Linux, there is a script in the environment/GUI directory named "starter.tcl" which should be run. Running the appropriate version should open a window titled "Control Panel" with the text "Waiting for ACT-R" at the top. Once that is waiting you must connect ACT-R to it by calling the **start-environment** command from Lisp:

```
(start-environment)
```

Regardless of which method was used to run and connect to the ACT-R Environment it should now contain several buttons which provide access to debugging and inspecting tools.

When you are done using ACT-R, before quitting the Lisp application you should disconnect it from the ACT-R Environment by calling the **stop-environment** command:

```
(stop-environment)
```

We will describe how to use some of the ACT-R Environment tools as we work through the tutorial, and you can find more information in the ACT-R Environment's manual found in the docs directory of the ACT-R distribution.

1.3.3 ACT-R models

An ACT-R model is a simulated cognitive agent created with the commands provided by the ACT-R software. A model is typically specified in a text file that contains the Lisp source code with the ACT-R commands that specify how the model works, and often also includes additional Lisp code for an experiment or task which the model will be performing. An ACT-R model file can be opened and edited in any application that can operate on text files. If you are running a Lisp with a GUI then that is probably the best tool for opening and editing your model files because it will provide lots of support for editing Lisp code, for example, things like matching parentheses and automatic indenting. If you are using a command line Lisp you will need to use some other text editor program, and if you are using the standalone version of the ACT-R software the ACT-R Environment included with that contains some additional tools for opening and editing text files (however most text editing programs will provide more capabilities than the ACT-R Environment's very basic text editing tools).

There are two ACT-R commands which will typically occur in every ACT-R model file: **clear-all** and **define-model**.

1.3.3.1 clear-all

The clear-all call will usually occur at the top of the model file. This command 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 if one is running a single model in a task then it is recommended to do so.

1.3.3.2 define-model

The define-model call is how one creates an ACT-R model. Within the call to define-model one includes all the calls to ACT-R commands which specify the initial conditions for that model. When the model file is loaded, define-model creates a model with the conditions specified, and whenever ACT-R is reset that model is returned to that same initial state.

1.3.4 Loading a model

To use an ACT-R model file it must be loaded, just as one loaded the ACT-R code itself. To do so, the Lisp load command can be called at the prompt with the file name or the Load menu option may be used to load the file. In addition to that, if you are using the ACT-R Environment, you can use the “Load Model” button on the Control Panel. If the model file contains Lisp functions to present an experiment to the model you may see better performance when running ACT-R if you compile the model file before loading it (where better performance means that the model and experiment will run faster but it will not change how the model itself performs the task). Most Lisp GUIs have a menu option called “Compile and Load” which you can use, but some Lisps compile all functions by default and thus make that unnecessary (consult your Lisp’s documentation for specific details). It is not required that one compile a model file however. Just loading it is sufficient, and most of the models in the tutorial run fast enough that compiling them will probably not make a significant difference.

1.3.5 Running a model

Once a model has been loaded, you can run it. Models that do not interact with an experiment can be run by calling the ACT-R **run** command. The **run** command requires one parameter, which is the length of simulated time to run the model measured in seconds. Thus, to run a model for 10 simulated seconds one would enter this at the Lisp prompt:

```
(run 10)
```

That will be sufficient for running the models in this unit. However, most of the future units will include functions which setup and run a task for the model to perform and the run command will be included in the operation of those functions. When there is a different function necessary to run the model and task the tutorial text will describe how to use it.

1.4 Creating knowledge in ACT-R

Now that we have described the basic components of ACT-R we will introduce the commands and syntax necessary to create the knowledge elements in an ACT-R model.

1.4.1 Chunk-Types

When creating both chunks and productions, there is an optional capability available in the ACT-R software called a **chunk-type** which can be helpful in specifying chunks and productions. 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 an ACT-R model.

Using chunk-type declarations does not affect the operation of the model. They exist only 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, and it will provide warnings when inconsistencies or problems are found relative to the chunk-types which are specified.

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.

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.

1.4.2 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. Here is an example from the **count** model which will be described in detail later in this unit:

```
(add-dm
   (b ISA count-order first 1 second 2)
   (c ISA count-order first 2 second 3)
   (d ISA count-order first 3 second 4)
   (e ISA count-order first 4 second 5)
   (f ISA count-order first 5 second 6)
   (first-goal ISA count-from start 2 end 4))
```

and here are the corresponding chunk-types which are defined in the **count** model:

```
(chunk-type count-order first second)
(chunk-type count-from start end count)
```

Each chunk 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 is a valid Lisp symbol and starts with an alphanumeric character. In the example above the names are **b**, **c**, **d**, **e**, **f**, 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 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 generally the symbols used in ACT-R commands are not case sensitive.

The rest of the chunk specification list 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.

1.4.3 Creating Productions

As indicated above, productions are maintained by the procedural module in ACT-R. It uses them to monitor the buffers of all modules and integrate their individual operations into coordinated behavior. The condition (also known as the left-hand side or LHS) of a production specifies tests for the content of buffers and the states of the buffers and modules. The action (also called the right-hand side or RHS) of a production specifies the actions to be taken by the production when it is fired, and will consist of changes to be made to the chunks in buffers along with new requests to be sent to the modules.

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

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

Each production must have a unique name and may also have an optional documentation string that describes what it does to help those reading the model code. 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 count-order first second)
(chunk-type count state number)
```

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

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

It is not necessary to space the production definition out over multiple lines or indent the components as shown above. It could be written on one line with only a single space between each symbol and result in the creation of the same production. However, since adding additional whitespace characters between the symbols does not affect the definition, spacing it out can make it easier to read.

1.4.3.1 Production Condition: Buffer Pattern Matching

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

After indicating which buffer to test, an optional declaration may be made using the symbol **isa** and the name of a chunk-type to provide a description of the set of slots which are being used in the test. In the condition above 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 there 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 to perform using 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
   number   =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 **number** and a variable value.

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

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

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

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

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

```
   =retrieval>
      first       =num1
      second      =num2
```

The first slot test it has tests the slot named **first** with the variable **=num1**. Since the variable **=num1** was also used in the **goal** buffer test, this is testing that the **first** slot of the chunk in the **retrieval** buffer has the same value as the **number** slot of the chunk in the **goal** buffer. The other slot test for the **retrieval** buffer is for the slot named **second** 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 first which has the same value as the slot named number of the chunk in the goal buffer, and it must have a slot named second whose value we will refer to using the variable =num2.

There is one other detail to note about the buffer pattern tests on the LHS of a production. Notice that the specification of a buffer pattern begins with the character “=” which is also used to indicate a variable in a production. The start of a buffer pattern is also specifying a variable using the name of the buffer. In this production that would be the variables =goal and =retrieval. Those variables will refer to the chunk that is in the **goal** buffer and the chunk that is in the

retrieval buffer respectively, and those variables can be used just like any other variable to test a value in a slot or to place that chunk into a slot as an action.

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

1.4.3.2 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
 number   =num2
+retrieval>
 ISA      count-order
 first    =num2
```

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

1.4.3.2.a Buffer Modifications, the = action

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

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

```
=goal>
 ISA      count
 number   =num2
```

It starts with the character "=" therefore this is a modification to the buffer. It will change the value of the **number** 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 **second** slot of the chunk in the **retrieval** buffer had in the condition). This is an instance of a variable being used to copy a value from one slot to another.

1.4.3.2.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 refer to such an action as a "*buffer* request" where *buffer* is the name of the buffer indicated e.g. a retrieval request or a goal request. Typically this results in the module replacing the chunk in the buffer with a different one, but not all modules handle requests the same way. As was noted above, the goal module handles requests by creating new chunks and the declarative module use 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 describe modules that perform perceptual and motor actions in response to requests.

Here is the retrieval request from the example production:

```
   +retrieval>
      ISA         count-order
      first       =num2
```

It is asking the declarative module to find a chunk that has a slot named **first** 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.

1.4.3.2.c Buffer Clearing, the - action

The 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.

1.4.3.2.d Implicit Clearing

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

1.5 The Count Model

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

If you run the model for 1 second using the run command you should see the following output in your Lisp application or the Listener window if you are using a standalone version of ACT-R:

```
> (run 1)
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED 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 C
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL C
     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
2
     0.150   PROCEDURAL             MOD-BUFFER-CHUNK GOAL
     0.150   PROCEDURAL             MODULE-REQUEST RETRIEVAL
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            START-RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK D
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL D
     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
3
     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
     0.250   PROCEDURAL             PRODUCTION-SELECTED STOP
     0.250   PROCEDURAL             BUFFER-READ-ACTION GOAL
     0.300   DECLARATIVE            RETRIEVED-CHUNK E
     0.300   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL E
     0.300   PROCEDURAL             PRODUCTION-FIRED STOP
4
     0.300   PROCEDURAL             CLEAR-BUFFER GOAL
     0.300   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   ------                 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 module that generated the event and some details describing the event. Any output from the model is also shown in the trace. By default the trace only shows the important events which occurred. However it is possible to change that to see more or less information in the trace as needed, and this model is set to show the most information possible. How to change the amount of detail shown in the trace is discussed in the unit 1 experiment description document, named “unit1_exp”.

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. 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 experiment 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 count-order first second)
(chunk-type count-from start end count)
```

The **count-order** chunk-type specifies the slots that will be used to encode the ordering of numbers. The **count-from** chunk type specifies the slots that will be used for the goal chunk of the model and 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
 (b ISA count-order first 1 second 2)
 (c ISA count-order first 2 second 3)
 (d ISA count-order first 3 second 4)
 (e ISA count-order first 4 second 5)
 (f ISA count-order first 5 second 6)
 (first-goal ISA count-from start 2 end 4))
```

Each of the lists in the add-dm command specifies one chunk. The first five define the counting facts named **b, c, d, e,** and **f**. They connect the number lower in the counting order (in slot **first**) to the number next in the counting order (in slot **second**). This is the knowledge that enables the model to count.

The last chunk created, **first-goal**, encodes the goal of counting from 2 (In slot **start**) to 4 (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 count slot.

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. This call indicates that the chunk named **first-goal** should be placed into the **goal** buffer. That setting of the buffer will happen when the model runs, and the results of that command can be seen in the first line of the trace shown above and copied here:

```
 0.000   GOAL      SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED 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 “requested nil” at the end of the event details). For now, that last part is not an important detail, but we will come back to other instances of that in later units.

1.5.4 The Productions

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

1.5.4.1 The Start Production

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

On its LHS it tests the **goal** buffer. It tests that there is a value in the start slot which it now references with the variable **=num1**. This is often referred to as binding the variable, as in,

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

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

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

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

The first line shows that the procedural module is performing an action called conflict-resolution. That action is the process which the procedural module uses to select which of the productions that matches the current state (if any) to fire next. The next line shows that among those that matched, the **start** production was selected. The important thing to remember is that the production was selected because its condition was satisfied. It did not fire first 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 system, 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 an action which modifies a chunk in a buffer, in this case the **goal** buffer, and module-request indicates that a request is being sent to a module through the indicated buffer.

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

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

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

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

```
     0.050   DECLARATIVE            START-RETRIEVAL
```

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

```
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
```

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

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

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

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         count-order
      first       =num1
      second      =num2
 ==>
   =goal>
      ISA         count-from
      count       =num2
   +retrieval>
      ISA         count-order
      first       =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 will also be true if the chunk in the **goal** buffer does not have a slot named **end** because a test for a slot value of a slot which does not exist is false, and thus the negation of that will be true.

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

```
   =retrieval>
      ISA         count-order
      first       =num1
      second      =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         count-order
      first       =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 **second** slot of the chunk in the **retrieval** buffer, and it makes a retrieval request to get the chunk which indicates what number is next. The third action is a special command that can be used in the actions of a production:

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

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

```
     0.150   PROCEDURAL             PRODUCTION-FIRED INCREMENT
2
```

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 specific value to which it was bound in the production will be used in the output which is displayed. That is why the trace shows 2 instead of =num1.

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

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

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

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

It will continue to be selected and fired until the value of the **count** and **end** slots of the goal chunk are the same, at which time its test of the **goal** buffer will fail. We see that it fires multiple times until this point in the trace:

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

1.5.4.3 The Stop Production

```
(P stop
   =goal>
      ISA         count-from
      count       =num
      end         =num
==>
   -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. The actions it takes are to again print out the current number and now to also clear the chunk from the **goal** buffer:

```
     0.300   PROCEDURAL             PRODUCTION-FIRED STOP
4
     0.300   PROCEDURAL             CLEAR-BUFFER GOAL
```

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

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

1.6 Pattern Matching Exercise

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

1. Either load the count model if it is not currently loaded or reset it to its initial conditions if it has been loaded. That is done either by pressing the “Reset” button in the Control Panel or by calling the ACT-R command **reset** at the Lisp prompt.

2. Click on the “Stepper” button in the Control Panel to open the Stepper window. In general, this tool will stop the model before every operation it performs. For each event that shows as a line in the trace, the stepper will force the model to wait for your confirmation before handling that event. That provides you with the opportunity to inspect all of the components of the model as it progresses and can be a very valuable tool for debugging models. Each time a production is selected the Stepper will show the text of the production, the bindings for the variables as they matched that production, and a set of parameters for that production (which we will not discuss until later in the tutorial). When the production is fired it will show the same information except that the production text will have the variables replaced with their bound values. When 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).

Right now you are going to use a feature of this window that allows you to manually instantiate the productions. That is, you will assign all of the variables the proper values when the production is selected before continuing to the firing of that production. To enable this functionality of the Stepper, click the "Tutor Mode**"** checkbox at the top of the Stepper window.

3. Click on the "Buffer Contents" 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. You can also use the command **buffer-chunk** to find the names of the chunks in the buffers. Calling it without any parameters will show all of the buffers and the chunks they contain. If you call it with the name of a buffer, for instance, (buffer-chunk goal), then it will print out the chunk that is in the named buffer. At this point all of the buffers are empty, but that will change as the model runs.

4. Now you should run the model for at least one second using the run command as shown above. The first action that occurs is the setting of the chunk in the **goal** buffer as was seen in the first line of the trace above:

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

That line should be displayed at the top of the Stepper window where it says "Next Event:". Hitting the "Step" button in the Stepper window will allow that action to occur. You can now inspect the chunk in the **goal** buffer using either the Buffer Contents window in the Environment or the **buffer-chunk** command.

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

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

```
GOAL: FIRST-GOAL-0 [FIRST-GOAL]
FIRST-GOAL-0
   START  2
   END  4
```

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

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

1.6.2 Pattern Matching Exercise Continued

5. If you Step past the conflict-resolution event you will come to the first production to match, **start**. Its structure will be displayed in the Stepper window and all of the variables will be highlighted. Your task is to go through the production rule replacing all of the variables with the values to which they are bound. When you click on a variable a dialog window will open in which you can enter its value. You must enter the value for every variable in the production (including multiple instances of the same variable) before it will allow you to progress to the next production.

Here is how you can determine the variable bindings:

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

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

6. Once the production is completely instantiated, you can fire it by hitting the "Step" button. The Stepper will then advance through the other events of the model as you continue to hit the "Step" button. You should step the model to the next production that matches and watch how the contents of the **goal** and **retrieval** buffer change based on the actions taken by the **start** production.

7. Then, at the risk of too much repetition, you will need to instantiate two instances of the production i**ncrement**. When the **start** and **end** slots of the goal are equal, the s**top** production will match and that will be the last one which you need to instantiate.

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

1.7 The Addition Model

The second example model uses a slightly larger set of count facts to do a somewhat more complicated task. It will perform addition by counting up. Thus, given the goal to add 2 to 5 it will count 5, 6, 7, and report the answer 7. You should load the **addition** model now.

The initial count facts are the same as those used for the **count** model with the inclusion of a fact that encodes 1 follows 0 and facts that encode counting from 7 up to 10. The chunk-type created for the goal information now has slots to hold the starting number (**arg1**), the number to be added (**arg2**), a slot for holding the current value counted to (**sum**), and a slot for holding the count that has been added so far (**count**):

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

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

```
(second-goal ISA add arg1 5 arg2 2)
```

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

```
     0.000   GOAL                    SET-BUFFER-CHUNK GOAL SECOND-GOAL REQUESTED NIL
     0.000   PROCEDURAL              CONFLICT-RESOLUTION
     0.050   PROCEDURAL              PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE             START-RETRIEVAL
     0.050   PROCEDURAL              CONFLICT-RESOLUTION
     0.100   DECLARATIVE             RETRIEVED-CHUNK F
     0.100   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL              CONFLICT-RESOLUTION
     0.150   PROCEDURAL              PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE             START-RETRIEVAL
     0.150   PROCEDURAL              CONFLICT-RESOLUTION
     0.200   DECLARATIVE             RETRIEVED-CHUNK A
     0.200   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL A
     0.200   PROCEDURAL              CONFLICT-RESOLUTION
     0.250   PROCEDURAL              PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE             START-RETRIEVAL
     0.250   PROCEDURAL              CONFLICT-RESOLUTION
     0.300   DECLARATIVE             RETRIEVED-CHUNK G
     0.300   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL G
     0.300   PROCEDURAL              CONFLICT-RESOLUTION
     0.350   PROCEDURAL              PRODUCTION-FIRED INCREMENT-SUM
     0.350   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE            START-RETRIEVAL
     0.350   PROCEDURAL             CONFLICT-RESOLUTION
     0.400   DECLARATIVE            RETRIEVED-CHUNK B
     0.400   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL B
     0.400   PROCEDURAL             CONFLICT-RESOLUTION
     0.450   PROCEDURAL             PRODUCTION-FIRED INCREMENT-COUNT
     0.450   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.450   DECLARATIVE            START-RETRIEVAL
     0.450   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   DECLARATIVE            RETRIEVED-CHUNK H
     0.500   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL H
     0.500   PROCEDURAL             PRODUCTION-FIRED TERMINATE-ADDITION
7
     0.500   PROCEDURAL             CONFLICT-RESOLUTION
     0.500   ------                 Stopped because no events left to process
```

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

In this sequence we see that the model alternates between incrementing the count from 0 to 2 and incrementing the sum from 5 to 7. The production **initialize–addition** starts things going and makes a retrieval request for the chunk which indicates the number following the arg1 number. **Increment-sum** then uses that retrieved chunk to update the current sum and requests a retrieval of a chunk to determine the next count value. That production fires alternately with **increment-count**, which processes the retrieval of the counter increment and requests a retrieval of an increment to the sum. **Terminate-addition** recognizes when the counter equals the second argument of the addition and it modifies the **goal** to make the model stop and outputs the answer.

1.7.1 The initialize-addition and terminate-addition productions

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

```
(P initialize-addition
   =goal>
      ISA         add
      arg1        =num1
      arg2        =num2
      sum         nil
  ==>
   =goal>
      ISA         add
      sum         =num1
      count       0
   +retrieval>
      ISA        count-order
      first      =num1
)
```

When the chunk in the goal buffer has values for the **arg1** and **arg2** slots but does not have a **sum** slot this production matches. Its actions modify the goal to add a **sum** slot set to be the first digit and a **count** slot to be zero, and make a retrieval request for a chunk to find the number that follows **=num1**.

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

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

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

1.7.2 The increment-sum and increment-count productions

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

```
(P increment-sum
   =goal>
      ISA         add
      sum         =sum
      count       =count
    - arg2        =count
   =retrieval>
      ISA         count-order
      first       =sum
      second      =newsum
```

```
  ==>
   =goal>
      ISA         add
      sum         =newsum
   +retrieval>
      ISA        count-order
      first      =count
)

(P increment-count
   =goal>
      ISA         add
      sum         =sum
      count       =count
   =retrieval>
      ISA         count-order
      first       =count
      second      =newcount
  ==>
   =goal>
      ISA         add
      count       =newcount
   +retrieval>
      ISA        count-order
      first      =sum
)
```

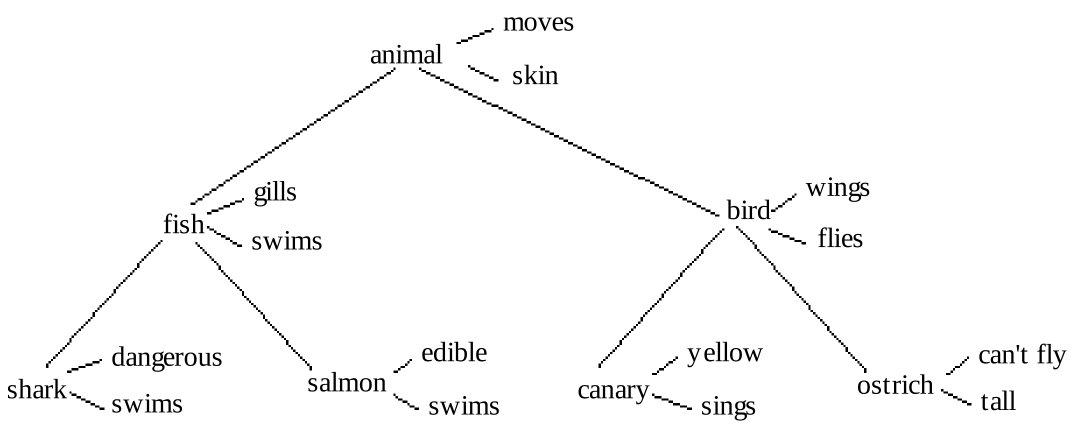
Some terminology which is often used when discussing productions which use a chunk which was placed into a buffer and then immediately clear that chunk from the buffer is to say that it "harvests" the chunk. Both of these productions harvest the chunk which resulted from the request of the other because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

1.7.3 The Addition Exercise

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

1.8 The Semantic Model

The last example model for this unit is the **semantic** model. It contains chunks which encode the following network of categories and properties. Its productions are capable of searching this network to make decisions about whether one category is a member of another category.



1.8.1 Encoding of the Semantic Network

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

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

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

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

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

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

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

How one chooses to organize the knowledge in a model can have an effect on the results, and thus that may 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, it would be difficult to use that representation with what has been discussed so far in the tutorial for performing a more general search for information in the network. When we get to unit 8 of the tutorial we will see some more advanced aspects of the pattern matching allowed in productions which could be used with such a representation to make the searching more general.

1.8.2 Testing for Category Membership

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

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

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

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

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

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

This is among the simplest cases possible and involves only the retrieval of this property to determine the answer:

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

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

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

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

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

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

```
   =goal>
      judgment    yes
)
```

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

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

1.8.3 Chaining Through Category Links

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

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

You can change the goal to **g2** using the goal-focus command. This can be done by editing the model and reloading it or by entering **(goal-focus g2)** at the Lisp prompt. Running the model with chunk **g2** as the goal will result in the following trace:

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

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

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

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

You should now go through the pattern matching exercise using the Stepper with **g2** set as the goal.

1.8.4 The Failure Case

Now change the goal to the chunk **g3**.

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

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

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

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

1.8.5 Query Conditions

In addition to testing the chunks in the buffers as has been done in all of the productions 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 to the buffer itself. A module must respond to some specific queries, but may have any number of additional queries to which it will respond. The result of a query will be either true or false. If any query tested in a production has a result which is false, then the production does not match.

1.8.5.1 Querying the buffer itself

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

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

```
  ?retrieval>
     buffer      full
```

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

```
  ?retrieval>
      buffer      failure
```

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

```
   ?retrieval>
       buffer     empty
```

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

```
   ?retrieval>
       buffer     requested
```

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

```
   ?retrieval>
       buffer     unrequested
```

1.8.5.2 Querying a buffer's module

The queries that are available for every module allow one to test for one of three possible states of the module: free, busy, or error. Below are examples of those queries 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 (the declarative module) is currently performing an action and will be false if it is not currently performing an action:

```
   ?retrieval>
      state       busy
```

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

```
?retrieval>
   state       error
```

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

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

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

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

1.8.5.3 Using queries in productions

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

```
?retrieval>
   buffer      empty
   state       free
```

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

```
?retrieval>
   state       free
```

or

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

1.8.6 The fail production

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

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

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

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

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

1.8.7 Model Warnings

Now that you have worked through the examples we will examine another detail which you might have noticed while working on this model. When you first loaded the **semantic** model there should have been warnings like this displayed:

```
#|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 |#
```

Lines that begin with ”#|Warning:” are warnings from ACT-R. These indicate that there is something in the model that may need to be addressed. This differs from warnings or errors which may be reported by your Lisp application which indicate a problem in the overall syntax or structure of the Lisp code in the file. If you see ACT-R warnings when loading a model you should always check 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.

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

That would provide you with an opportunity to fix the problem before running the model and trying to determine why the production did not fire when 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 running the model.

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** file included with the unit 1 files 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 addition of two two-digit numbers. Once all of the pieces have been created as described, 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 REQUESTED NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            START-RETRIEVAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            START-RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.250   PROCEDURAL             PRODUCTION-FIRED PROCESS-CARRY
     0.250   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE            START-RETRIEVAL
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
     0.300   DECLARATIVE            RETRIEVED-CHUNK FACT34
     0.300   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL FACT34
     0.300   PROCEDURAL              CONFLICT-RESOLUTION
     0.350   PROCEDURAL              PRODUCTION-FIRED ADD-TENS-CARRY
     0.350   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE             START-RETRIEVAL
     0.350   PROCEDURAL              CONFLICT-RESOLUTION
     0.400   DECLARATIVE             RETRIEVED-CHUNK FACT17
     0.400   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL FACT17
     0.400   PROCEDURAL              CONFLICT-RESOLUTION
     0.450   PROCEDURAL              PRODUCTION-FIRED ADD-TENS-DONE
     0.450   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.450   PROCEDURAL              CONFLICT-RESOLUTION
     0.450   ------                  Stopped because no events left to process
```

There is a working solution model included with the unit 1 files, and it is also described in the experiment description text for this unit. So, 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** 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).

1.9.1 Chunk-types

The first thing we should do is define the chunk types that we will use in writing the model. There are two chunk-types which we will define for doing this task. One to represent addition facts and one to represent the goal chunk which holds the components of the task for the model. These chunk types will be created with the **chunk-type** command as described in section 1.4.1.

1.9.1.1 Addition Facts

The first chunk type you will need is one to represent the addition facts. It should be named **addition-fact** and have slots named **addend1**, **addend2,** and **sum**.

1.9.1.2 The Goal Chunk Type

The other chunk type you will need is one to represent the goal of adding two two-digit numbers. It should be named **add-pair.** It must 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 is described in section 1.4.2.

1.9.2.1 The Addition Facts

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

```
3+4=7
6+7=13
10+3=13
1+7=8
```

They will use the slots of the type addition-fact and should be named based on their addends. For example, the fact that 3+4=7 should be named **fact34**. The addends and sums for these facts will be the corresponding numbers. Thus, **fact34** will have slots with the values 3, 4, and 7.

1.9.2.3 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.

Once you have completed adding the chunk-types and chunks to the model you should be able to load it and inspect the components you have created. To see the chunks you have created you can press the "Declarative Viewer" button on the Control Panel. That will open a declarative memory viewer (every time you press that button a new declarative viewer window will be opened). You can view a particular chunk by clicking on it in the list of chunks on the left of the declarative memory viewer. If you define a lot of chunks it may be difficult to find a particular one in the list. To help in that situation there is a filter at the top of the declarative memory viewer (the recessed button that defaults to saying **none**) that will allow you to select a set of slot names. Only chunks in declarative memory which have values for all of the slots in the set chosen will be displayed. Try selecting **(addend1 addend2 sum)** as the filter. You should now only see the chunks for the addition facts that you created. If you select **none** for the filter, then all of the chunks in declarative memory are displayed.

You can also inspect declarative memory from the Lisp prompt. The command **dm** will print out all of the chunks in declarative memory. You can also specify the name of chunks as parameters to the **dm** command and only those chunks will be printed. The command **sdm** can be used to display a subset of declarative memory. Its parameters are a chunk specification (the same as one would specify in a retrieval request in a production but without the +retrieval> indicator) and it prints out only those chunks from the model's declarative memory which match that specification. For example, **(sdm - addend1 nil)** will print out all of the chunks which have a value in the addend1 slot (which could also be specified by **(sdm addend1 =x)** since a variable binding is only possible for a slot which has a value) whereas **(sdm addend1 3)** will print only the addition facts that have the value **3** in the **addend1** slot.

1.9.3 Productions

So far, we have been looking mainly at the individual productions in the models. However, production systems get their power through the interaction of the productions. Essentially, one production will set the condition for another production to fire, and it is the sequence of productions firing that lead to performing the task.

Your task is to write the ACT-R productions which perform as described below in English 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 1.4.3.

First we will describe the overall process which this model will use to add the numbers (this is of course not the only way to perform this task):

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.

Now here is a detailed description of six productions which will implement that process using the chunks created previously. This is not the only way to write productions to perform that process, but they are sufficient to perform the task for any two digit additions.

START-PAIR

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

ADD-ONES

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

PROCESS-CARRY

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

NO-CARRY

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

ADD-TENS-DONE

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

ADD-TENS-CARRY

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

1.9.4 Running the model

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

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

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

1.9.5 Incremental Creation of Productions

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

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED START-PAIR
     0.050   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE            START-RETRIEVAL
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     0.100   DECLARATIVE            RETRIEVED-CHUNK FACT67
     0.100   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT67
     0.100   PROCEDURAL             CONFLICT-RESOLUTION
     0.150   PROCEDURAL             PRODUCTION-FIRED ADD-ONES
     0.150   PROCEDURAL             CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE            START-RETRIEVAL
     0.150   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   DECLARATIVE            RETRIEVED-CHUNK FACT103
     0.200   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL FACT103
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
     0.200   ------                 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 more productions the system 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 a command called **whynot**. The **whynot** command can be used to tell you why the condition of a production does not match the current state. It can be called from the Lisp prompt with the name of a production and it is also available through the ACT-R Environment. To use the ACT-R Environment's version you need to first press the "Procedural" button in the Control Panel to open a production viewer window. Then you can select a production from the list of defined productions in the production viewer window and press the "Why not?" button. It will open a window with the details of why the production does not match or print the current instantiation of that production if in fact it does match.

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 debug those issues.