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, but it is not a complete reference on the theory of ACT-R. More detailed information on the theory can be found in the paper "An integrated theory of the mind", which is available on the ACT-R website at: http://act-r.psy.cmu.edu/publications/pubinfo.php?id=526, and also in the book "How Can the Human Mind Occur in the Physical Universe?".

The goals of this unit are to introduce the knowledge representations which are used in ACT-R, present the formal notation for specifying that knowledge within an ACT-R model, and to describe how those types of knowledge interact in an ACT-R model.

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**. Thus 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 defined by its **chunk-type** and its **slots**. One can think of **chunk-types** as categories (e.g., birds) and slots as category attributes (e.g., color or size). 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 type of the first chunk is **chase** and its slots are **agent** and **object**. The **isa** slot is special and specifies the type of the chunk. The type of the second chunk is **addition-fact** and its slots are **addend1**, **addend2**, and **sum**.

```
Action023
    isa chase
    agent dog
    object cat

Fact3+4
    isa addition-fact
    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 set as a subgoal 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 operations 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 do, but do not necessarily detail everything that needs to happen within the production. You will learn the syntax for precise production specification in ACT-R later in this unit.

1.2 Creating Knowledge Elements

To create chunks, chunk types, and productions one must issue the necessary ACT-R commands. Because ACT-R commands are Lisp functions they must be enclosed in parentheses to execute them. The first term after the left parenthesis is the command name. That is followed by the details needed for the command and then a right parenthesis. In the following sections we will show how to use the commands to create the knowledge representations in ACT-R.

1.2.1 Creating New Chunk Types

To create a new type of chunk like “bird” or “addition fact”, you need to specify a frame for the chunk using the **chunk-type** command. This requires that you specify the name of the chunk type and the names of the slots that it will have. The general chunk type specification looks like this:

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

and here are some examples:

```
(chunk-type bird species color size)
(chunk-type column row1 row2 row3)
```

The first argument to **chunk-type** specifies the name of the new type. In the examples above the names are bird and column. Each type of chunk also has a number of slots each of which can hold one value. The remaining arguments in the **chunk-type** specification are the names of the slots for that type of chunk.

1.2.2 Creating Chunks

The command to create a set of chunks and add them to 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 we will discuss later:

```
(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 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 you want which is not already used as the name of a chunk as long as it is also a valid Lisp symbol. 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 to refer to the chunk - it is not considered to be a part of the chunk, and can in fact be omitted in which case a new and unique name will be generated for that chunk automatically. The rest of the list is pairs of slot names and initial values. The first pair must be the **isa** slot and the type of the chunk. The **isa** slot is special because every chunk has one. Its value is the type of the chunk, which must be either a type that was defined with **chunk-type** or one of the predefined types, and it cannot be changed once the chunk is created. The remainder of the slot-value pairs can be specified in any order, and it is not necessary to specify an initial value for every slot of the chunk. If an initial value for a slot is not given, that slot will be empty which is denoted with the Lisp symbol **nil**.

1.2.3 Productions

A production is a condition-action pair. The condition (also known as the left-hand side or LHS) specifies a pattern of chunks that must be present in the **buffers** for the production to apply. The action (right-hand side or RHS) specifies some actions to be taken when the production fires.

1.2.4 Buffers

Before continuing with productions we need to describe what these buffers are. The buffers are the interface between the procedural memory system in ACT-R and the other components (called modules) of the ACT-R architecture. For instance, the **goal** buffer is the interface to the goal module. Each buffer can hold one chunk at a time, and the actions of a production affect the contents of the buffers. Essentially, buffers operate like scratch-pads for creating, storing, and modifying chunks.

In this chapter we will only be concerned with two buffers -- one for holding the current goal and one for holding information retrieved from the model's declarative memory module. Later chapters will introduce other buffers and modules as well as further clarify the operations of the **goal** and **retrieval** buffers used here.

1.2.5 Productions Continued

The general form of 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. The buffer tests consist of a set of patterns to match against the current buffers' contents. If all of the patterns correctly match, then the production is said to match and it can be selected. It is possible for more than one production to successfully match the current buffer contents. However, among all the matching productions only one will be selected, and that production's actions will be performed. The process of choosing a production from those that match is called conflict resolution, and it will be discussed in detail in later units. For now, what is important is that only one production may fire at a time. After a production fires, matching and conflict resolution will again be performed and that will continue until the model has finished.

1.3 Production Specification

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 the following production that counts from one number to the next:

```
(P counting-example                English Description
   =goal>                          If the goal chunk is
     isa       count                 of the type count
     state     incrementing          the state slot has the value incrementing
     number    =num1                 there is a number we will call =num1
   =retrieval>                     and the chunk in the retrieval buffer
     isa       count-order           is of type count-order
     first     =num1                 the first slot has the value  =num1
     second    =num2                 and the second slot has a value we will call =num2
==>                                Then
    =goal>                         change the goal
     number    =num2                 to continue counting from =num2
   +retrieval>                     and request a retrieval
     isa      count-order            of a count-order chunk to
     first    =num2                  find the number that follows =num2
)
```

1.3.1 Production Conditions

The condition of the preceding production specifies a pattern to match to the **goal** buffer and a pattern to match to the **retrieval** buffer:

```
   =goal>
     isa          count
     state        incrementing
     number       =num1
   =retrieval>
     isa          count-order
     first        =num1
     second       =num2
```

A pattern starts by naming which buffer is to be tested followed by the symbol ">". The names **goal** and **retrieval** specify the goal buffer and the retrieval buffer. It is also required to prefix the name of the buffer with "=" (more on that later). After naming a buffer, the first test must specify the chunk-type using the isa test and the name of a chunk-type. That may then be followed by any number of tests on the slots for that chunk-type. A slot test consists of an optional modifier (which is not used in any of the tests in this example production), the slot name, and a specification of the value it must have. The value may be either a specific constant value or a variable.

Thus, this part of the first pattern:

```
  =goal>
     isa         count
     state       incrementing
```

means that the chunk in the **goal** buffer must be of the chunk-type **count** and the value of its state slot must be the explicit value **incrementing** for this production to match.

The next slot test in the goal pattern involves a variable:

```
     number      =num1
```

The “=” prefix in a production is used to indicate a variable. Variables are used in productions to test for general conditions. They can be used for two basic purposes. In the condition they 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. They can also be used to copy values from one slot to another slot in the actions of the production. The name of the variable can be any symbol and should be chosen to help make the purpose of the production clear. A variable is only meaningful within a specific production. The same variable name used in different productions does not have any relation between the two uses. For a variable test to successfully match there must be some value in the slot. An empty slot, indicated with the Lisp symbol **nil**, will not match to a variable in a buffer specification.

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

```
   =retrieval>
      isa         count-order
      first       =num1
      second      =num2
```

First it tests that the chunk is of type **count-order**. Then it tests the **first** slot of the chunk with the variable **=num1**. Since that variable was also used in the goal test, this is testing that this slot of the chunk in the **retrieval** buffer has the same value as the **number** slot of the chunk in the **goal** buffer. Finally, it tests the **second** slot with a variable called **=num2**. This means that there must be a value in the **second** slot, it cannot be empty, but there are no constraints placed on what that value must be.

In summary, this production will match if the chunk in the **goal** buffer is of type count, the chunk in the **retrieval** buffer is of type count-order, the chunk in the **goal** buffer has the value incrementing in its state slot, the value in the number slot of the **goal** buffer's chunk and the first slot of the **retrieval** buffer's chunk match, and there is some value in the second slot of the chunk in the **retrieval** buffer.

One final thing to note is that **=goal** and **=retrieval**, as used to specify the buffers, are also variables. They will be bound to the chunk that is in the **goal** buffer and the chunk that is in the **retrieval** buffer respectively. These variables for the chunks in the buffers 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.

1.3.2 Production Actions

The right-hand side of a production consists of a set of actions that affect the buffers. Here are the actions from the example production again:

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

The actions are specified similarly to the conditions. They state the name of a buffer followed by ">" and then some slot and value specifications. There are three types of actions that can happen to a buffer and they are designated by the character that precedes the name of the buffer.

1.3.2.1 Buffer Modifications

If the buffer name is prefixed with "=" then the action is for the production to immediately modify the chunk currently in that buffer. Thus this action on the **goal** buffer:

```
=goal>
   number       =num2
```

changes the value of the **number** slot of the chunk in the **goal** buffer to the current value of the **second** slot of the chunk in the **retrieval** buffer. This is an instance of a variable being used to copy a value from one slot to another.

1.3.2.2 Buffer Requests

If the buffer name is prefixed with a "+", then the action is a request to that buffer's module. Typically this results in the module replacing the chunk in the buffer with a different one, but it could also be a request for the module to make some change to the chunk that is already in the buffer. Each module has its own purpose and handles different types of requests. In later units of the tutorial we will describe modules that can handle visual attention, manual control requests, and some other types of actions.

In this unit, we are dealing only with the declarative memory and goal modules. Requests to the declarative memory module (the module for which the **retrieval** buffer is the interface) are always a request to retrieve a chunk from the model's declarative memory which matches the specification provided. If a matching chunk is found, it will be placed into the **retrieval** buffer.

Thus, this request:

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

is asking the declarative memory module to retrieve a chunk which is of type count-order and with a **first** slot that has the same value as **=num2**. If such a chunk exists in the declarative memory of the model, then it will be placed into the **retrieval** buffer.

1.3.2.3 Buffer Clearing

The third type of action that can be performed on a buffer is to explicitly clear the chunk from the buffer. This is done by placing "-" 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>
```

1.3.2.4 Implicit Clearing

In addition to the explicit clearing action one can make, there is also an implicit clearing that will occur for buffers. A request of a module with a "+" action will automatically cause that buffer to be cleared. So, this request from the example production:

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

results in the **retrieval** buffer being automatically cleared.

1.4 ACT-R Models

We will be going through a series of examples to illustrate how a production system like ACT-R's works and to introduce you to the ACT-R environment. All of your work in the ACT-R tutorial will probably involve using the ACT-R environment. The ACT-R environment is a GUI for running, inspecting, and debugging ACT-R models. It can be run with Lisp implementations from multiple vendors in a variety of operating systems or as a standalone application without the need for a Lisp application under Windows or Mac OS X.

Model files for ACT-R are text files that contain Lisp source code with the ACT-R commands to specify how the model works. They 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 models because it will provide lots of support for editing Lisp code, for example, things like matching parentheses and Lisp language reference tools. 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 environment you may use the text editing facilities it provides (however most text editing programs will provide more capabilities than the ACT-R environment's very basic text tools and they are not recommended unless you have no other options).

To use a model you must load it into a Lisp application that is running the ACT-R software. So, the first thing you need to do is start your Lisp application and load ACT-R. If you have a Lisp with a GUI then there should be an option, typically under the File menu, to load a file, 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 the "load-act-r-6.lisp" file found at the top level of the ACT-R source file distribution. Once that file has loaded you should start the ACT-R environment application. Once it is ready you will connect ACT-R to it by calling the **start-environment** command from Lisp (see the QuickStart.txt file in the docs directory of ACT-R 6 for more details on starting the system and a shortcut for starting the environment in some Lisp & OS combinations).

After ACT-R is loaded and the ACT-R environment is connected you are now ready to load and run ACT-R models. To load a model you can do that directly in your Lisp application just as you loaded ACT-R. Alternatively, you can use the "Load Model" button on the Control Panel of the ACT-R environment. For model files that contain Lisp functions to present an experiment to the model you will see better performance if you compile the model file before loading it. Most Lisp GUIs offer an option called "Compile and Load" which you can use and some Lisps compile all functions by default (consult your Lisp's documentation for specific details).

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)
```

Now, to put these ideas into action, we will start working with some example models.

1.5 The Count Model

The first model is a simple production system 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. It is contained in the file called “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 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 in the running of the model. The event shows the time in seconds at which it happened, the module that generated the event and the details of that event. Any output from the model is also shown in the trace. The default trace shows everything that happened in the model, but often that is more detail than is necessary and it is possible to change the level of detail that is displayed in the trace. How to do that is discussed in the unit 1 model code description document. A model code description document accompanies each of the tutorial units and provides more detailed information about the Lisp code and functions used in that unit's models. Those are the documents with "_exp" on the end of the name e.g. "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. We will be focusing on the specification of the chunks and productions. The other commands that are used in this unit's models are described in the unit 1 model description document.

1.5.1 Chunk-types for the Count model

In the model file you will find the following two specifications for new chunk types used by this model:

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

The **count-order** chunk type is used for chunks that encode the ordering of numbers. The **count-from** chunk type will be used as the type for the goal chunk of the model and has slots to hold the starting number, the ending number, and the current count so far.

1.5.2 Declarative Memory for the Count model

In the model file you will find the initial chunks placed into the declarative memory of the model:

```
(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 are of the type **count-order** and each counting fact connects 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, **first-goal**, is of the type **count-from** and it 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 stated, it will be empty in the chunk **first-goal** which means that it will have the value **nil**.

1.5.3 Setting the Initial Goal

The chunk **first-goal** is declared to be the model's current goal (placed into the **goal** buffer) by the command **goal-focus** in the model:

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

The results of that command can be seen in the first line of the trace:

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

The trace shows that the chunk **first-goal** is set to be the chunk in the **goal** buffer by the goal module. It also indicates that this chunk was not requested by a production. For now, that is not an important detail, but we will come back to other instances of that in later units.

Now that we have seen the chunks the model has we will look at the productions that use those chunks to count.

1.5.4 The Start Production

The productions are specified with the command **p**, as described earlier. The count model has three productions: **start, increment,** and **stop**. The first production that gets selected and fired by the model is the production **start** which can be seen in this part of the trace:

```
     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 conflict resolution to determine which productions, if any, match the current contents of the buffers and then choosing one of them. The next line shows that among those that matched, the **start** production was selected.

The third line shows that the selected production tested the chunk in the goal buffer as a condition. The 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 every production will take 50ms between the time it is selected and when it fires.

Now we will look at the details of the **start** production. Here is its definition from the model:

```
(p start
   =goal>
      ISA         count-from
      start       =num1
      count       nil
 ==>
   =goal>
      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 checks that the count slot is currently empty i.e. that it has the value **nil**.

On the RHS it performs two actions. The first is to change the value of the count slot of the chunk in the **goal** buffer to be the value bound to **=num1**. The other action is to request that the declarative memory system retrieve a chunk of type **count-order** that has the value which is bound to **=num1** in its **first** slot.

These actions can be seen as the next two lines of the trace:

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

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 **retrieval** buffer that happens because of the +retrieval> request.

The next line 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
```

The next line indicates that the procedural system is 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 are no productions whose conditions are satisfied at this time.

The following two lines show the completion of the retrieval request by the declarative memory module and then the setting of the **retrieval** buffer to that chunk.

```
     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
```

1.5.5 The Increment Production

Here is the specification of the **increment** production:

```
(P increment
   =goal>
      ISA         count-from
      count       =num1
    - end         =num1
   =retrieval>
      ISA         count-order
      first       =num1
      second      =num2
==>
   =goal>
      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 this production will only match 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**.

The **retrieval** buffer test checks that it has retrieved a count-order chunk with a value of 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>
      count       =num2
   +retrieval>
      ISA         count-order
      first       =num2
   !output!       (=num1)
```

The first two actions are very similar to those in the **start** production. It updates the count slot of the goal chunk with the next number as found from the count-order chunk in the **retrieval** buffer and then requests that a count-order chunk be retrieved to get the next number. The third action is a special command that can be used in the actions of a production:

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

!output! (pronounced bang-output-bang) can be used on the RHS of a production to display information in the trace. It must be followed by a list of items and those items 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 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 occurs in the output list the specific value to which it was bound in the production will be substituted into the output which is displayed. Thus the reason 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. A retrieval request is made, the **goal** buffer is modified, 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
```

1.5.6 The Stop Production

The final production in the model is **stop**:

```
(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 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
```

1.5.7 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 action that shows as a line in the trace, the stepper will force the model to wait for your confirmation before taking that action. 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 it 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. It will also show the details of which chunk is retrieved when there is a retrieval request.

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 window.

3. Click on the “Buffer Viewer” button in the Control Panel to bring up a new Buffer viewer 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 the buffers are empty, but that will change as the model runs.

4. Now you should run the model for at least 1 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
```

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 Viewer or the **buffer-chunk** command.

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

If you inspect the **goal** buffer you will see that it contains a chunk named **first-goal-0** and not the chunk **first-goal**. When a chunk is placed into a buffer it is always a new copy of the chunk that goes into the buffer. The name of the chunk that was copied is also shown in the Buffer Viewer (and by the **buffer-chunk** command) in square brackets after the name of the chunk actually in the buffer. The buffers operate as scratch pads for modifying chunks. As long as a chunk is in a buffer it can be manipulated by productions or any of the other modules. However, once it leaves the buffer it can no longer be modified by the model because any attempt to put it back into a buffer where it can be changed will only result in a new copy.

1.5.9 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 will open in which you can enter its value. You must enter the value for every instance of a variable in the production (including multiple instances of the same variable) before it will allow you to progress to the next production.

Here are the rules for doing this:

- **=goal** will always bind to the name of the current contents of the goal buffer. This can be found with the Buffer 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 viewer or the **buffer-chunk** command.
- A variable will have the same value everywhere in a production that matches. The bound values are displayed in the Stepper window as you enter them.
- 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 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 **Increment**. Then, when the **start** and **end** slots of the goal are equal, the **Stop** 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 the next model.

1.6 The Addition Model

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

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 those that encode all the numbers up to 10. The chunk type for the goal now has slots to hold the starting number (**arg1**) and the number to be added (**arg2**):

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

There are two other slots in the goal called **count** and **sum** which will be used to hold the results of the counting and the total so far as the model progresses. Here is the initial goal chunk created for the model:

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

Since the **count** and **sum** slots are not specified, they are empty, which is indicated with the default value of **nil**.

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

The first thing you will 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 a reduced trace to make it easier to read the sequence of productions that fire. The model code description document for this unit describes how one can change the amount of detail shown in the trace. Sometimes having all the details can be useful for determining what a model is doing and other times a more concise trace is desirable.

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 requests a retrieval of an increment to the sum. **Increment-sum** processes that retrieval and requests a retrieval of an increment to the count. 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 modifies the **goal** to make the model stop.

1.6.1 The initialize-addition and terminate-addition Productions

The production **initialize-addition** initializes an addition process whereby the system 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		**English Description**	
`=goal>`		If	the goal is
`ISA`	`add`		to add the arguments
`arg1`	`=num1`		=num1 and
`arg2`	`=num2`		=num2
`sum`	`nil`		but the sum has not been set
==>		Then	
`=goal>`			change the goal
`sum`	`=num1`		by setting the sum to =num1
`count`	`0`		and setting the count to 0

```
   +retrieval>                         and request a retrieval
      isa         count-order             of a chunk of type count-order
      first       =num1                   for the number that follows =num1
)
```

This production initializes the **sum** slot to be the first digit and the **count** slot to be zero. It requests a retrieval of 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** applies:

```
(P terminate-addition              English Description
   =goal>                              If the goal is
      ISA          add                     to add
      count        =num                    and the count has the same value
      arg2         =num                    as arg2
      sum          =answer                 and there is a sum
==>                                    Then
   =goal>                                  change the goal
      count        nil                     to stop counting
)
```

This production clears the **count** slot of the goal by setting it to **nil** (remember **nil** is the value of an empty slot). This causes the model to stop because other than **initialize-addition**, (which requires that the sum slot be empty) all the other productions require there to be a chunk in the **count** slot. So, after this production fires none of the productions will match the chunk in the **goal** buffer.

1.6.2 The increment-sum and increment-count Productions

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

```
(P increment-sum                       English Description
   =goal>                              If  the goal is
      ISA          add                     to add
      sum          =sum                    and the sum is =sum
      count        =count                  and the count is =count
    - arg2         =count                 and the count has not reached the end
=retrieval>                           and a chunk has been retrieved
      ISA          count-order            of type count-order
      first        =sum                   where the first number is =sum
      second       =newsum                and it is followed by =newsum
==>                                   Then
   =goal>                                 change the goal
      sum          =newsum                so that the sum is =newsum
   +retrieval>                        and request a retrieval
      isa         count-order             of a chunk of type count-order
      first       =count                  for the number that follows =count
)
```

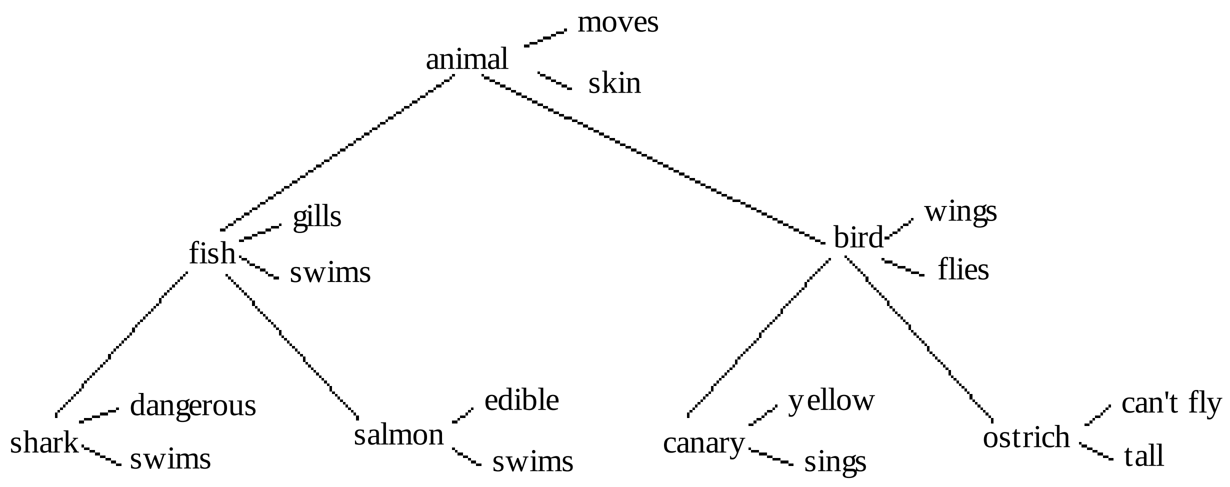
```
(P increment-count                    English Description
   =goal>                             If the goal is
      ISA          add                    to add
      sum          =sum                   and the sum is =sum
      count        =count                 and the count is =count
   =retrieval>                        and a chunk has been retrieved
      ISA          count-order            of type count-order
      first        =count                 where the first number is =count
      second       =newcount              and it is followed by =newcount
==>                                   Then
   =goal>                                 change the goal
      count        =newcount              so that the count is =newcount
   +retrieval>                        and request a retrieval
      isa         count-order             of a chunk of type count-order
      first       =sum                    for the number that follows =sum
)
```

1.6.3 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.

1.7 The Semantic Model

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



1.7.1 Encoding of the Semantic Network

All of the links in this network are encoded by chunks of type **property** 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 a true value on the **dangerous** attribute. **p2** encodes that a shark can swim by encoding the value **swimming** on the **locomotion** attribute. **p3** encodes that a shark is a fish by encoding fish as the value on the **category** attribute.

You can inspect the chunks in the **add-dm** command of the model to see how the rest of the semantic network is encoded.

1.7.2 Queries about Category Membership

Queries about category membership are encoded by goals of the **is-member** type. There are 3 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 judgment nil)
```

which represents the query to decide if a canary is a bird. The judgment slot is **nil** reflecting the fact that the decision has yet to be made about whether it is true. 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
```

This is among the simplest cases possible and involves only the retrieval of this property

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

for verification of the query. 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                         English Description
   =goal>                                   If    the goal is
      ISA         is-member                       to judge membership
      object      =obj                            of =obj
      category    =cat                            in the category =cat
      judgment    nil                             and the judgment has not begun
==>                                         Then
  =goal>                                          change the goal
      judgment    pending                         so that the judgment is pending
   +retrieval>                        and request a retrieval
      ISA          property               of a chunk of type property
      object       =obj                   for the object =obj
      attribute    category               involving the attribute category
)
```

```
(P direct-verify                      English Description
   =goal>                             If     the goal is
      ISA          is-member                 to judge the membership
      object       =obj                      of =obj
      category     =cat                      in the category =cat
      judgment    pending                    and the judgment is pending
   =retrieval>                         and a chunk has been retrieved
      ISA          property                  of type property
      object       =obj                      for the object =obj
      attribute    category                  involving an attribute category
      value        =cat                      with the value =cat
==>                                   Then
   =goal>                                    modify the goal
      judgment     yes                       so that the judgment is yes
)
```

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

1.7.3 Chaining Through Category Links

A slightly more complex case occurs when the category is not an immediate super ordinate of the queried 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 query is defined in chunk **g2**:

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

One can change the goal to **g2** by the command **(goal-focus g2)**. This can be done by editing the model and reloading 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 involves an extra production, **chain-category**, which retrieves the category in the case that an attribute has been retrieved which does not allow a decision to be made. Here is that production:

(P chain-category	**English Description**
`   =goal>`	If the goal is
`      ISA         is-member`	to judge the membership
`      object      =obj1`	of =obj1
`      category    =cat`	in =cat
`      judgment    pending`	and the judgment is pending
`   =retrieval>`	and a chunk has been retrieved
`      ISA         property`	of type property
`      object      =obj1`	for the object =obj1
`      attribute   category`	involving an attribute category
`      value       =obj2`	with a value =obj2
`    - value       =cat`	and this is not the same as =cat
`==>`	Then
`   =goal>`	change the goal
`      object      =obj2`	to judge the category membership of =obj2
`   +retrieval>`	and request a retrieval
`      ISA         property`	of a chunk of type property
`      object      =obj2`	for the object =obj2
`      attribute   category`	involving the attribute category
)	

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

1.7.4 The Failure Case

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

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

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

The production **fail** applies and fires when a retrieval attempt fails. This production uses a condition on the LHS that we have not yet seen.

1.7.5 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 state of the buffer or the module which controls it. This is done using a "**?**" operator instead of an "**=**" before the name of the buffer. A module may have a number of different queries to which it will respond, but there are some to which all buffers and modules will always respond. The queries that are always valid are whether there is a chunk in the buffer or not, whether or not that chunk was requested by the procedural module, and for one of three possible states of the module: free, busy, or error. The result of a query will always be either true or false. If any query tested in a production has a result which is false, then the production does not match. Here are examples of the possible queries with respect to the **retrieval** buffer.

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

```
  ?retrieval>
     buffer      full
```

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

```
   ?retrieval>
      buffer     empty
```

This query will be true if the chunk in the **retrieval** buffer was requested by a production (which should always be true for the **retrieval** buffer) or false if the chunk was not requested by a production:

```
   ?retrieval>
      buffer     requested
```

This query will be true if the chunk in the **retrieval** buffer was placed there other than by a request in a production and false if the chunk was the result of a production's request:

```
   ?retrieval>
      buffer     unrequested
```

This query will be true if the **retrieval** buffer's module (the declarative memory module) is not currently working to retrieve a chunk and false if it is currently working on retrieving a chunk:

```
   ?retrieval>
     state       free
```

This query will be true if the declarative memory module is working on retrieving a chunk and false if it is not:

```
   ?retrieval>
     state       busy
```

This query will be true if there was an error in the last request made to the **retrieval** buffer and false if there was no error:

```
   ?retrieval>
      state       error
```

For retrieval requests, an error means that no chunk could be found that matched the request.

It is also possible to make multiple queries at the same time. For instance this query would check if the module was not currently handling a request and that there was currently a chunk in the buffer:

```
   ?retrieval>
      state       free
      buffer      full
```

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 retrieving a chunk:

```
   ?retrieval>
      state       free
```
or
```
   ?retrieval>
     - state      busy
```

1.7.6 The fail production

Now, here is the production that fires in response to a category request not being found.

```
(P fail                                 English Description
   =goal>                               If    the goal is
      ISA         is-member                   to judge membership
      object      =obj1                       of =obj1
      category    =cat                        in =cat
      judgment    pending                     and the judgment is pending
   ?retrieval>                            and the retrieval
      state       error                       has failed
==>                                     Then
   =goal>                                     change the goal
      judgment    no                          so that the judgment slot is no
)
```

Note the testing for a retrieval 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 that matches the specification requested, the buffer's state indicates an error. 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.

1.7.7 Model Warnings

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

```
#|Warning: Creating chunk CATEGORY of default type chunk |#
#|Warning: Creating chunk PENDING of default type chunk |#
#|Warning: Creating chunk YES of default type chunk |#
#|Warning: Creating chunk NO of default type chunk |#
```

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", looking at the warnings would have shown something like this:

```
#|Warning: Creating chunk PENDING of default type chunk |#
#|Warning: Creating chunk PNEDING of default type chunk |#
```

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 could be displayed and you should always read them to make sure that there are no serious problems before running the model.

1.8 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 tutorial models 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 added 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).

1.8.1 Chunk-types

The first thing we need to do is define the chunk types that will be used by 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 1.2.1.

1.8.1.1 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**.

1.8.1.2 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**.

After you have created those chunk-types you have completed the chunk type portion of this model. Now it is time to create the chunks that allow the model to perform addition along with an initial goal to do such an addition.

1.8.2 Chunks

We are now going to define the chunks that will allow the model to solve the problem 36 + 47. This is done using the **add-dm** command which is described in section 1.2.2.

1.8.2.1 The Addition Facts

You need to add the addition facts to encode the following math facts:

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

They will be of 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 appropriate numbers.

1.8.2.3 The Initial Goal

You should now create a chunk called **goal** which encodes that the goal is to add 36+47. 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 specify a particular chunk-type, and only chunks having that chunk-type will be displayed. Try selecting **addition-fact** 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 like the filter in the declarative memory viewer. Its parameters are a chunk description (similar to a retrieval request) and it prints out only those chunks from the model's declarative memory which match that description. For example, **(sdm isa addition-fact)** will print out only those chunks which are of type addition-fact and **(sdm isa addition-fact addend1 3)** will print only the addition facts that have the value **3** in the **addend1** slot.

You are now ready to begin the next section, where you will write productions.

1.8.2 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 equivalents of the production rules described below in English, which can perform multi-column addition. To do that you will need to use the ACT-R **p** command to specify the productions as described in sections 1.2.5 and 1.3.

Here are the English descriptions of the six productions needed for this task.

START-PAIR

```
IF the goal is to add a pair of numbers
   and the ones digits of the pair are available
   but the ones digit of the answer is nil
THEN note in the one-ans slot that you are busy computing
      the answer for the ones digit
   and request a retrieval of the sum of the ones digits.
```

ADD-ONES

```
IF the goal is to add a pair of numbers
   and you are busy waiting for the answer for the ones digit
   and the sum of the ones digits has been retrieved
THEN store the sum as the ones answer
   and note that you are busy checking the answer for the carry
   and request a retrieval to determine if the sum equals 10
     plus a remainder.
```

PROCESS-CARRY

```
IF the goal is to add a pair of numbers
   and the tens digits are available
   and you are busy working on the carry
   and the one-ans equals a value which a retrieval
    finds is the sum of 10 plus a remainder
THEN make the ones answer the remainder
   and note that the carry is 1
   and note you are busy computing the sum of the tens digits
   and request a retrieval of the sum of the tens digits.
```

NO-CARRY

```
IF the goal is to add a pair of numbers
   and the tens digits are available
   and you are busy working on the carry
   and the one-ans equals a sum
   and there has been a retrieval failure
THEN note that the carry is nil
   and note you are busy computing the sum of the tens digits
   and request a retrieval of the sum of the tens digits.
```

ADD-TENS-DONE

```
IF the goal is to add a pair of numbers
   and you are busy computing the sum of the tens digits
   and the carry is nil
   and an addition-fact has been retrieved
THEN note the answer for the tens digits is the sum of that
   retrieved chunk.
```

ADD-TENS-CARRY

```
IF the goal is to add a pair of numbers
   and the tens digits are available
   and you are busy computing the sum of the tens digits
   and the carry is 1
   and the sum of the tens digits has been retrieved
THEN set the carry to nil
   and request a retrieval of 1 plus that sum.
```

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.

1.8.3 Incremental Creation of Productions

It is 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 run 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.

1.8.4 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 what condition on the LHS of a production is causing it to fail to match. It can be called from the Lisp prompt with the name of a production, or it is also available as a button on the "Procedural Viewer" of the Control Panel. To use the ACT-R environment's version you should select a production from the list of defined productions in the Procedural Viewer window and then press the "Whynot" 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 it lets you watch everything that the model does. You can also use **whynot**, the Buffer viewer, and the Declarative viewer windows while using the Stepper to better understand what is happening in the model at any specific point in its run.