

Unit 1: Understanding Production Systems

Section 1.1: The ACT-R Production System

ACT-R is a production system theory that tries to explain human cognition by developing a model of the knowledge structures that underlie cognition. 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** whereas procedural knowledge is represented in **productions**. Thus chunks and productions are the basic building blocks of an ACT-R model.

The function of Unit 1 is to present the formal notation for specifying chunks and production rules and to describe how the two types of knowledge interact to produce cognition. You will get some directed practice in interpreting and writing production systems.

Throughout the tutorial there will be fill-in-the-blank questions like the following:

The fundamental units of declarative knowledge are _____.

The fundamental units of procedural knowledge are _____.

These are included to help you in learning and understanding the material.

1.1.1 Declarative Units 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 **type** and its **slots**. One can think of types as categories (e.g., birds) and slots as category attributes (e.g., color or size). Below are chunks that encode the facts that *the dog chased the cat* and that $4+3=7$. The type of the first chunk is **chase** and its slots are **agent** and **object**. The **isa** slot gives 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

```

Fill in the slots for the **addition-fact** that **1+4=5**:

```

Fact1+4:
  isa      _____
  addend1  _____
  addend2  _____
  sum      _____

```

1.1.2 Production Rules in ACT-R

A production rule is a statement of a particular contingency that controls behavior. 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 rule (the IF part) consists of a specification of the chunks in various **buffers**. The action of a production rule (the THEN part) basically involves the modifications of those chunks or requests for other chunks. The above are informal English specifications of production rules. They give an overview of what the production does in the context of the declarative memory structures used, but do not necessarily detail everything that needs to happen within the production. You will learn the syntax for precise production specification within the ACT-R system.

A production rule specifies an _____ to be taken when a _____ is met.

1.1.3 Production Rule Format

A production rule is a condition-action pair. The condition (also known as the left-hand side) specifies a pattern of chunks that must be present in the buffers for the production rule to apply. The action (right-hand side) specifies some actions to take.

The buffers are the interface between the procedural memory system and the other components (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. 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 declarative memory module. Later chapters will introduce other buffers and modules.

The general form of a production rule is:

```
(p Name
  list of buffer tests
==>
  list of buffer changes
)
```

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 be selected, and from all the selected productions one will be chosen to fire and that production's actions will be performed. The process of choosing a production from those that are selected 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, selection and conflict resolution will again be performed and that will continue until the model has finished.

The specification of a production above involves a special ACT-R command. All ACT-R commands are Lisp functions and therefore are specified in parenthesis. The first term in the parenthesis is the command name.

The command name for specifying a production is the single letter _____.

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 a particular number

	English Description
(P example-counting	
=goal>	If the goal is
isa count	to count
state counting	the current state is counting
number =num1	there is a number we will call =num1
=retrieval>	and a chunk has been retrieved
isa count-order	of type count-order
first =num1	where the first number is =num1
second =num2	and it is followed by another
	number 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 fact
first =num2	for the number that follows =num2
)	

1.1.4 ACT-R's Condition Form

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

A pattern starts by naming which buffer is to be tested followed by ">". 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 these tests), 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    counting
```

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 **counting**.

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 general conditions. They can be used to test that a slot holds any value, that two slots hold the same value or that two slots hold different values. 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.

The first time a variable is used in a production it gets assigned (bound to) the value of the specified slot from the chunk in the buffer. If the slot does not have a value, then the pattern does not match. Further uses of that variable within the production will be tests against the specific value to which it is bound.

So, this slot test from the goal pattern:

```
number      =num1
```

causes the variable called **=num1** to be bound to the current value of the **number** slot from the chunk in the goal buffer, if it has a value.

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 bound in the goal test this is testing that this slot has that same value. Finally, it tests the **second** slot which will bind its value to the **=num2** variable.

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

Assume that the count-order chunks correctly encode the numerical order. If this production is selected and fires and **=num1** is bound to 3, then **=num2** will be bound to _____.

1.1.5 ACT-R's Action Side

The right-hand side (RHS -- the part after the arrow) or action side of a production consists of a small set of actions. The typical actions are to change the contents of the buffers as in our example:

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

The actions are specified similarly to the conditions. They start with the name of a buffer followed by ">" and then any number of slot and value specifications.

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

```
=goal>
  start      =num2
```

changes the value of the **start** slot of the chunk in the goal buffer to the value of the **=num2** variable.

If the buffer name is prefixed with "+" then the action is a request to the buffer's module. Typically this results in the module replacing the chunk in the buffer with a different one. 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 declarative memory that matches the specification provided and to place that chunk into the retrieval buffer. Different modules may handle different types of requests and may respond in other ways. Future units will introduce some of the other modules and their functionality.

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 value bound to **=num2** and place it into the retrieval buffer. If there exists such a chunk, then it will be placed into the **retrieval** buffer.

Section 1.2: The Count Model

We will be going through a series of examples to illustrate how a production system works and to introduce you to the ACT-R environment. All of your work in the ACT-R tutorial will probably involve using some variant of the ACT-R environment. The environment is a GUI for writing, running and debugging ACT-R models. It can be run with Lisp implementations from multiple vendors in a variety of operating systems or without a Lisp application installed under Windows or Mac OS X. More information on the environment is available on the ACT-R website (<http://act-r.psy.cmu.edu/software>).

The first example is a simple production system that counts up from one number to another - for example it will count up from 2 to 5 -- 2,3,4,5. This is called the Count production system or the Count model. It is included with the models for unit 1 which you should have gotten from the ACT-R website along with the tutorial.

To see the Count model in action, start the ACT-R environment (see the instruction in the environment's manual for the version you are using) and follow these directions:

- Find and select the environment's "Control Panel" Window
- Press the "Open Model..." button
- Open count (which is located in the unit1 folder)
- A confirmation dialog will indicate a successful load
- Several edit windows will be opened
- You should find the Listener window (MCL, LispWorks, or standalone environment), Debug window (ACL for Windows) or terminal window (OpenMCL or ACL for Linux/Unix) of the Lisp application and position it so that you can see it (this will be referred to as the Listener window from now on in the tutorial).

If you press the run button on the control panel, you should see the following in the Listener window:

```
Time 0.000: Start Selected
Time 0.050: Start Fired
Time 0.100: C Retrieved
Time 0.100: Increment Selected
2
Time 0.150: Increment Fired
Time 0.200: D Retrieved
Time 0.200: Increment Selected
3
Time 0.250: Increment Fired
Time 0.300: E Retrieved
Time 0.300: Increment Selected
4
Time 0.350: Increment Fired
Time 0.350: Stop Selected
Time 0.400: F Retrieved
5
Time 0.400: Stop Fired
Time 0.400: Checking for silent events.
Time 0.400: * Nothing to run: No productions, no events.
```

This display is called the trace of the model. The trace lists the productions that fired and the chunks that were retrieved from declarative memory. It also lists the numbers that were printed out as the productions fire. As more features of the system are used additional information will also be shown in the trace.

1.2.1 Declarative Memory for the Count Example

The chunks with which the count example begins are contained in the **chunk** window of the environment. There you will find the following chunks:

```
(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 5 step start)
```

Each list specifies one chunk. The details of how to specify chunks will be discussed later in the unit, but we will briefly discuss the details of the above chunks here. The first five define counting facts named **b-f**. They are all of the type **count-order**. 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 system 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 5 (in slot **end**). It is declared to be the current goal (placed into the goal buffer) by the command (**goal-focus first-goal**) in the **command** window. Note that the step slot of **first-goal** is set to **start** at the beginning. It will be set to **counting** as the counting progresses and to **stop** when the counting is over. This use of a slot in the goal to maintain a current state (keeping track of what is happening now) is a common practice when writing ACT-R models. It provides a way to limit which productions are appropriate at any particular time, as will be shown in the following descriptions. It is not necessary to do so however, and there are other means by which the same control structure can be accomplished, but many of the models in the tutorial will use a state slot in the goal to make the production sequencing clear.

1.2.2 The Start Production

The productions are contained in the **production** window. The count model has three productions. The first production to apply is called **start**:

	English Description
(p start	If the goal is
=goal>	to count from
ISA count-from	the number =num1
start =num1	and the step is start
step start	Then
==>	change the goal
=goal>	to note that one is now counting
step counting	and request a retrieval
+retrieval>	of a count-order fact
ISA count-order	for the number that follows =num1
first =num1	
)	

This production responds to the fact that the goal is at the start step and schedules a retrieval of the number that follows the initial number. It also changes the goal state to note that one is now counting.

1.2.3 The Increment Production

The production that iterates through the counting is the **increment** production:

```
(P increment
  =goal>
    ISA      count-from
    start    =num1
  - end      =num1
    step     counting
  =retrieval>
    ISA      count-order
    first    =num1
    second   =num2
==>
  =goal>
    start    =num2
  +retrieval>
    ISA      count-order
    first    =num2
  !output!   (=num1)
)
```

English Description

If the goal is
to count from
the number =num1
and =num1 is not the final digit
and one is counting
and a chunk has been retrieved
of type count-order
where the first number is =num1
and it is followed by =num2

Then
change the goal
to continue counting from =num2
and request a retrieval
of a count-order fact
for the number that follows =num2
and output =num1 in the trace

There are two new things used in this production. On the LHS we see the slot test modifier "-" being used. This 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 **start** slot.

On the RHS it uses the !output! (pronounced bang-output-bang) command to get the number printed out. !output! is a special command which can be used on the RHS of a production to display information in the trace. It must be followed by a list of things that will be printed in the trace when the production fires. 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). The list will be displayed on one line in the trace between the selected and the fired lines of the trace for the production in which it occurs.

If the goal has 4 in its start slot the production **increment** will print out ___ and change the start slot of the goal to ___.

1.2.4 The Stop Production

The other production in the **production** window is **stop**:

```
(P stop
  =goal>
    ISA      count-from
    start    =num
    end      =num
    step     counting
==>
```

English Description

If the goal is
to count from
the number =num
to the number =num
and one is counting

Then

```

=goal>                                change the goal
  step                                to note a stop
!output!                               and output the number =num
)

```

Production **stop** will fire when the value of the _____ slot is the same as the value of the _____ slot of the chunk in the goal buffer.

1.2.5 Pattern Matching Exercise

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

1. Click the **Reload** button or the **Reset** button in the **Control Panel** to prepare the model to run again.
2. Click on the **Stepper** button in the **Control Panel** to bring up the **Stepper** window. In general, this window will stop the model before every operation it performs. For this model, each time a production is selected and again when it is fired it will show that production with all of the variables bound and wait for you to press the **step** button before going on. It will also stop and show which chunk is retrieved when there is a retrieval request. It is a very useful tool for debugging your models because it allows you to see what is happening at every step of the way. More details on using the Stepper can be found in the environment's manual.

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** window, click on 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 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. If you select **goal** you will see that the goal buffer contains the chunk **first-goal**.
4. Click the **Run** button in the **Control Panel** to start the assignment. The first production to apply, **Start**, will appear in the **Stepper**. Its structure will be displayed in the lower right pane of the window and all of the variables will be highlighted. Your task is to go through the production rule replacing all 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 current contents of the goal buffer. This can be found in the **Buffer window**.
 - **=retrieval** will always bind to the chunk in the retrieval buffer. This can also be found in the **Buffer window**.
 - If a variable has been bound to a value, it must be assigned the same value throughout the matching. The bound values are displayed in the middle right pane of the **Stepper**.
 - 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 at the top of the **Stepper** window. The **Stepper** will then stop at the firing of that production and after you step past that you will see the retrieval event for the count-order chunk **C**.
7. Then, at the risk of too much repetition, you will need to instantiate three 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, press the **Close Model button** on the **Control Panel** and then go on to the next section of this unit, which will describe the next model.

Section 1.3: The Addition Model

The second example uses the same database 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 open the **addition** model from the **unit1** folder in the same way as you opened 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 the numbers up to 10. The goal now encodes the starting number (**arg1**) and the number to be added (**arg2**):

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

There are two other slots in the goal called count and sum. Since they are not specified, they are empty, which is indicated with the default value of **nil**. They will be used to hold the results of the counting and the total result so far.

If you run this model (make sure you close the **Stepper** first) you will see:

```

Time 0.000: Initialize-Addition Selected
Time 0.050: Initialize-Addition Fired
Time 0.100: F Retrieved
Time 0.100: Increment-Sum Selected
Time 0.150: Increment-Sum Fired
Time 0.200: A Retrieved
Time 0.200: Increment-Count Selected
Time 0.250: Increment-Count Fired
Time 0.300: G Retrieved
Time 0.300: Increment-Sum Selected
Time 0.350: Increment-Sum Fired
Time 0.400: B Retrieved
Time 0.400: Increment-Count Selected
Time 0.450: Increment-Count Fired
Time 0.450: Terminate-Addition Selected
Time 0.500: H Retrieved
Time 0.500: Terminate-Addition Fired
Time 0.500: Checking for silent events.
Time 0.500: * Nothing to run: No productions, no events.

```

In this sequence we alternate between incrementing the count from 0 to 2 and incrementing the sum from 5 to 7. **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 stop.

1.3.1 The Initialize-Addition and Terminate-Addition Production

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.

<pre> (P initialize-addition =goal> ISA add arg1 =num1 arg2 =num2 sum nil ==> =goal> sum =num1 count 0 +retrieval> isa count-order first =num1)</pre>	<p>English Description</p> <p>If the goal is to add the arguments =num1 and =num2 but the sum has not been set</p> <p>Then change the goal by setting the sum to =num1 and setting the count to 0 and request a retrieval of a chunk of type count-order for the number that follows =num1</p>
--	---

This production initializes the sum to be the first digit and the count to be zero. It schedules a retrieval for the number that follows =**num1**.

If the value of the arg1 slot is 4 this production will initialize the sum slot to ____ and the count slot to ____.

Pairs of productions will apply after this to keep incrementing the sum and the count until the count equals the second argument at which time terminate-addition applies:

	English Description
(P terminate-addition	
=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 (nil is the value of an empty slot). This causes the model to stop because all of the productions require that the count slot contain a value. So, after this production fires none of the productions will match the chunk in the goal buffer.

1.3.2 The Increment-Sum and Increment-count Productions

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

	English Description
(P increment-sum	
=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 =sum	where the first number is =sum
second =newsum	and it is followed by =newsum
==>	Then

```

=goal>
  sum      =newsum
+retrieval>
  isa      count-order
  first    =count
)

```

change the goal
so that the sum is =newsum
and request a retrieval
of a chunk of type count-order
for the number that follows =count

If the value of the count slot of the goal is 2 and the sum slot is 6 this production will change the sum slot to ___ and request a retrieval for the number following ___.

```

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

```

English Description
If the goal is
to add
and the sum is =sum
and the count is =count
and a chunk has been retrieved
of type count-order
where the first number is =count
and it is followed by =newcount
Then
change the goal
so that the count is =newcount
and request a retrieval
of a chunk of type count-order
for the number that follows =sum

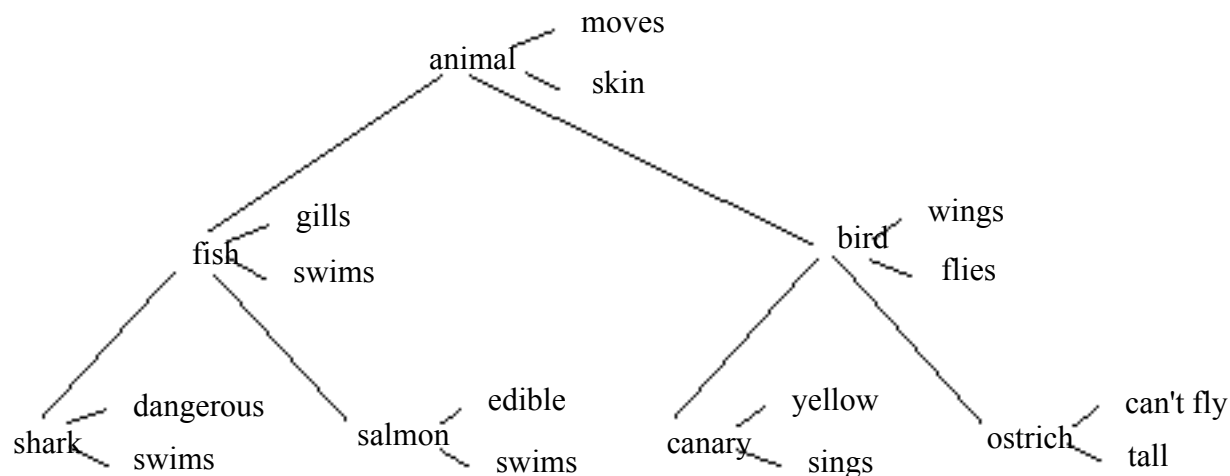
If the value of the count slot is 2 and the sum slot is 6 this production will change the count slot to ___ and request a retrieval for the number following ___.

1.3.3 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. You will have to set the tutor flag in that window again. When you are finished working with this model you should close it before going on to the next model.

Section 1.4: The Semantic Model

The last example for this unit is the **semantic** model in **unit1**. 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.



1.4.1 Encoding of the Semantic Network

Each of the links in this network is 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 should inspect the chunks in the **chunks** window to see how the rest of the semantic network is encoded. Fill in the slots below to encode that a dog runs

```
dog-runs
  isa      _____
  object   _____
  attribute _____
  value    _____
```

1.4.2 Queries about Category Membership

Queries about category membership are encoded by goals of the **is-member** type. There are 3 goals encoded in the initial declarative memory. 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 decided if a canary is a bird. The judgment slot is nil reflecting the fact that the decision has yet to be made whether it is true. If you run the model with **g1** in the goal buffer you will see the following trace:

```
Time 0.000: Initial-Retrieve Selected
Time 0.050: Initial-Retrieve Fired
Time 0.100: P14 Retrieved
Time 0.100: Direct-Verify Selected
Time 0.150: Direct-Verify Fired
Time 0.150: Checking for silent events.
Time 0.150: * Nothing to run: No productions, no events.
```

This is about the simplest case possible and involves retrieval of the property

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

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

English Description

```
If the goal is
to judge membership
of =obj
in the category =cat
and the judgment has not begun
Then
change the goal
so that the judgment is pending
and request a retrieval
of a chunk of type property
for the object =obj
involving the attribute category
```

```
(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
)
```

English Description

If the goal is to judge the membership of =obj in the category =cat and the judgment is pending and a chunk has been retrieved of type property for the object =obj involving an attribute category with the value =cat

Then modify the goal so that the judgment is yes

Direct-verify will recognize that salmon is a _____.

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

1.4.3 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 of this is given in the following goal.

```
(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 **command** window and reloading (using the **Reload** button on the **Control Panel**) or by typing (**goal-focus g2**) into the listener window after reloading. Running the model with that goal will result in the following trace:

```
Time 0.000: Initial-Retrieve Selected
Time 0.050: Initial-Retrieve Fired
Time 0.100: P14 Retrieved
Time 0.100: Chain-Category Selected
Time 0.150: Chain-Category Fired
Time 0.200: P20 Retrieved
Time 0.200: Direct-Verify Selected
Time 0.250: Direct-Verify Fired
Time 0.250: Checking for silent events.
Time 0.250: * Nothing to run: No productions, no events.
```

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:

<code>(P chain-category</code>	English Description
<code>=goal></code>	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
)	

If the goal is to verify that a salmon is an animal, **chain-category** will change the object slot of the goal from _____ to _____.

You should reload the model, make sure the goal is set to **g2** and go through the pattern matching exercise in the **Stepper**.

1.4.4 The Fail Production

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

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

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

```
Time 0.000: Initial-Retrieve Selected
Time 0.050: Initial-Retrieve Fired
Time 0.100: P14 Retrieved
Time 0.100: Chain-Category Selected
Time 0.150: Chain-Category Fired
Time 0.200: P20 Retrieved
Time 0.200: Chain-Category Selected
Time 0.250: Chain-Category Fired
Time 0.300: Failure Retrieved
```

```

Time 0.300: Fail Selected
Time 0.350: Fail Fired
Time 0.350: Checking for silent events.
Time 0.350: * Nothing to run: No productions, no events.

```

The production **Fail** applies and fires when a retrieval attempt fails:

(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
isa 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 will indicate 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 reload the model and make sure your goal is set to **g3**, and then go through the pattern matching exercise using the **Stepper**.

Section 1.5: Creating Declarative Structure

Up to this point, we have had you simply interpret complete models. Now we are going to have you work through the steps of creating your own. The first step in this regard is creating the declarative structure. Recall that chunks are intended to represent pieces of knowledge that a person might be expected to have when they solve a problem. The basic elements of the representation are **types** and **slots**.

Here are chunks that encode the facts that the *dog chased the cat*, and that $3+4=7$. Fill in the missing slots:

```

Action1:
  isa chase
  agent _____
  object _____

```

```
Fact3+4:
  isa addition-fact
  addend1 _____
  addend2 _____
  sum      _____
```

1.5.1. Using Chunk-Type to Create New Chunk Types

To create a new type of chunk like “bird” or “addition problem”, you need to specify a frame for the chunk using the **chunk-type** command. This requires that you give the name of the chunk type and the names of the slots that it will have. The general notational specification is of the form (**chunk-type type slot-1 slot-2 ... slot-n**). For example,

```
(chunk-type bird wings feathers beak flies)
(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 which can each hold one value. The remaining arguments in the **chunk-type** specification are the names of the slots for that type of chunk.

Create below the chunk type for an addition fact with slots addend1, addend2, and sum:

```
(chunk-type _____ _____ _____ _____)
```

1.5.2. Creating Declarative Chunks

Just as there is a special syntax for creating chunk types, there is also a special syntax for creating a set of chunks. The command to add a set of chunks to declarative memory is **add-dm**. It takes any number of chunk specifications as its arguments. Here is an example from the **count** 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 5 step start))
```

Each chunk is specified by a list. The first element of the list is the name of the chunk. The name can be anything you want (with the exception of some chunk names that are already used by the system) and should be different for each chunk. 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. 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. 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 default to the value nil. One thing to note about that is that a variable in a production cannot be bound to the value nil. If such a binding is attempted in a production the match will fail.

The command for adding chunks to declarative memory is _____.

1.5.3. Tutor Exercises on Declarative Memory

We would like you to do a series of exercises with the ACT-R tutor to help you get a sense of the syntax of ACT-R structures. If you have access to the ACT-R environment you should open tutor model 1.1. The instructions that follow will guide you through using the tutor. However, if you do not have access to the ACT-R environment you might simply try going through these exercises outside of the tutor and running them in ACT-R.

To open the tutor model select **Tutor 1.1** from the option menu next to the **Open Model** button on the **Control Panel** of the ACT-R Environment. Then, press the **Open Model** button. It will prompt you for the name of your model, which it will create. You may enter any name, but we recommend calling it **tutor1-1**. Then it will open up a number of windows, including a **chunk-type** window where you are to do your first exercises below. You interact with the tutor by typing the appropriate expansion for the currently highlighted node in the window. You should do these problems in the order listed. There is no constraint on the order you give the slots for a problem. When interacting with the tutor if at any time you get stuck you can ask for help by pressing the **F1** key.

Also, please note that you should not close your model until you have completed entering all of the required code. If you do close the model before you finish, you will be able to continue to work on your model later, but you will not receive any guidance from the tutor.

Problem 1.

Write a chunk-type declaration for representing an addition-fact with slots **addend1**, **addend2**, and **sum**. Here is how you do this: Select the **chunk-type** window. The current node highlighted is {name}. You should type **addition-fact** for the name of this chunk-type and then hit the **enter** key or the **spacebar**. The next node highlighted is {slot} and the names of the slots for this chunk-type are **addend1**, **addend2**, and **sum**. Type each of those followed by the **enter** key or the **spacebar**. Once you have entered all of those the tutor will create a new chunk-type form for you to fill in.

Problem 2.

Write a chunk-type, called **add-pair**, representing the goal to add a pair of two-digit numbers with slots for the tens digit of the first number (**ten1**), ones digit of the first number (**one1**), tens digit of the second number (**ten2**), ones digit of the second number (**one2**), tens digit of the answer (**ten-ans**), ones digit of the answer (**one-ans**), and information about carry (**carry**).

Problem 3.

Create a chunk-type, **number**, with a slot to hold the **value** of the number.

After you have created those three chunk-types you have completed the chunk-type portion of this tutorial model and there will no longer be a highlighted node in the **chunk-type** window. Now go to the **chunk** window to create the chunks specified in the exercises below:

Problem 4 (a-d).

Write chunks for the numbers **three**, **four**, **seven**, and **eight** giving them their English names and the values 3, 4, 7, and 8 respectively.

Problem 5 (a-c).

Write chunks of type **addition-fact**, encoding the addition facts $3+4=7$, $4+3=7$, and $4+4=8$ using the chunk names for the numbers. You should name these chunks **fact34**, **fact43**, and **fact44** respectively.

Problem 6.

Create a chunk called **goal** which encodes the goal to add $36+47$.

When you are done...

A dialog will pop-up and inform you that the model is complete. To take the environment out of tutor mode you should close the model by pressing the **Close Model** button on the **Control Panel**. Then you should open the model by selecting **Existing** from the option menu next to the **Open Model** button and press the **Open Model** button and select the model you saved. To see the chunks you have created click the **Declarative Viewer** button on the **Control Panel**, which opens up 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. However, there are many chunks already defined which may make it difficult to find the chunks you just added. The filter at the top of the declarative memory viewer (the recessed button that defaults to saying **none**) 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 are displayed.

You are now ready to begin the next section, where you will learn how to write production rules.

Section 1.6: Writing Productions

A production rule is a condition-action pair where the condition specifies some buffer patterns that must be matched for the production rule to apply and the action specifies some changes to the buffers. The general form of a production rule is:

```
(p Name
  list of buffer patterns
  ==>
  list of buffer changes to make
)
```

where **p** is the command for creating a production and **Name** is the name of the production. It is also possible to add an optional string after the name of the production as a comment.

1.6.1. Condition Form

The condition of a production rule consists of a specification of the necessary contents of various buffers. The following is a production that matches the goal to add two numbers with attention on the ones digits, harvests a retrieval of the sum of the ones digits, notes the sum, and requests a retrieval to check whether the sum requires processing a carry:

```
(p add-ones
  =goal>
    isa add-pair
    one1 =num1
    one2 =num2
    one-ans waiting
  =retrieval>
    isa addition-fact
    addend1 =num1
    addend2 =num2
    sum =sum
  ==>
  =goal>
    one-ans =sum
    carry waiting
  +retrieval>
    isa addition-fact
    addend1 ten
    sum =sum
)
```

English Description

If the goal
is to add a pair of numbers
and =num1 is the ones digit of the first
and =num2 is the ones digit of the second
and the ones digit of the answer is waiting
and the retrieved chunk
is of type addition-fact stating
=num1 plus
=num2 equals
=sum

Then
change the goal so that
the ones answer is =sum
and note that a carry is being processed
and request a retrieval
of a chunk of type addition-fact
with addend1 being ten
and the sum being =sum

Let's examine the patterns for the goal buffer and the retrieval buffer from the condition part of the production rule above:

```
=goal>
  isa add-pair
  one1 =num1
  one2 =num2
  one-ans waiting
=retrieval>
  isa addition-fact
  addend1 =num1
  addend2 =num2
  sum =sum
```

Symbols prefixed by "=", like =**goal**, =**num1**, and =**num2** are **variables** that can match to any element, except nil which is the indication that a slot has no value, while elements like **waiting** are **constants**. ACT-R distinguishes between constants and variables. If constants are used in the definition, then the match is highly specific -- the production will fire if the value of the slot of the chunk and the constant match exactly. A variable allows you to relax the criteria for matching, so that, for example, the value of a particular slot of a chunk can be anything.

It is also possible to have negative tests on slot values and to repeat slots. Consider the following goal buffer pattern from a production in a model that can do the Tower of Hanoi task:

```
=goal>
  isa check-disk
  size =size
- size 1
  goal =peg
  current =peg
- covered 1
```

This condition checks for a disk which does not have a size of 1, which is on the same peg in the goal and current state (both have value =**peg**), and which is not covered by a disk of size 1.

Write a pattern to test that the two numbers being added are not the same

```
=goal>
  isa add-column
  number1 =num1
- number2 _____
```

1.6.2. The Action Side of Productions

The RHS of a production consists of actions, which are usually changes to the buffers. Generally, all of the variables on the right-hand side must have been bound on the left-hand side. Let's consider the RHS of the add-ones production above:

```
=goal>
  one-ans =sum
  carry waiting
+retrieval>
  isa addition-fact
  addendl ten
  sum =sum
```

This modifies the current chunk in the goal buffer (since **=goal** is used) and requests a new chunk be retrieved (since a '+' is used to indicate an action on the **retrieval** buffer).

Section 1.7: Creating a Production System

So far, we have focused on writing single production rules. However, production systems get their power through the interaction of different production rules. Essentially, one production will set the condition for another production rule to fire. Typically the sequence of production rules starts with the setting of a goal to perform a particular task. One can set an initial goal by the command **goal-focus**.

Thus, if you want to have the system focus on a goal chunk called **addition-problem**, one would use the command:

```
( _____ )
```

Exercise 1.2 involves coding a production system to do mental addition of 2 digit numbers. It involves the following chunk types which you have already created in your preceding tutor exercise:

1. (chunk-type addition-fact addendl addendl2 sum)

This chunk type encodes the addition facts.

2. (chunk-type add-pair ten1 one1 ten2 one2 carry ten-ans one-ans)

This chunk type encodes the goal with the first digit represented in the **ten1** and **one1** slots, the second digit represented in the **ten2** and **one2** slots, and the answer to be stored in the **ten-ans** and **one-ans** slots. The **carry** slot will be important in processing carries from the ones place to the tens place.

3. (chunk-type number value)

This encodes numbers and their values.

Your task is to write the ACT-R equivalents of the production rules described below, which can perform multi-column addition. This lesson also uses the tutor in the ACT-R environment. To open the tutor model select **Tutor 1.2** from the option menu next to the **Open Model** button. Then, press the **Open Model** button. It will prompt you for the name of your model, which it will create. You may enter any name and we recommend calling it **tutor1-2**. This tutor model already comes with the declarative memory (addition facts and numbers) defined. It also contains a definition of the goal of the problem, which is to add 36 and 47:

```
(goal ISA add-pair ten1 three one1 six ten2 four one2 seven tens-ans nil
      ones-ans nil carry nil)
```

The tutor operates as it did for the last exercise, except now you will be entering productions. As before, you can press the **F1** key for help.

One tip: The slot test pattern **slotname nil** can be used to test that a chunk does not have a value for **slotname**.

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 "waiting" on 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 waiting on
   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 "waiting" on 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 waiting on a retrieval for the carry
   and the one-ans equals a sum
   and it has been retrieved that the one-ans equals ten
   plus a remainder
THEN make the ones answer the remainder
   and note that the carry is one
   and that you are "waiting" on a retrieval of 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 waiting on a retrieval for the carry
and the one-ans equals a sum
and there has been a retrieval failure
THEN note that the carry is nil
and that you are "waiting" on a retrieval of 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 waiting on retrieval of the tens digits
and the carry is nil
and the sum of the tens digits has been retrieved
THEN note the sum of tens digits.

ADD-TENS-CARRY

IF the goal is to add a pair of numbers
and the tens digits are available
and you are waiting on retrieval of the tens digits
and the carry is one
and the sum of the tens digits has been retrieved
THEN set the carry to nil
and request a retrieval of one plus the sum.

When you are finished entering the productions, save your model, reload it, and then run it. If you would like to make changes (for instance to try a different goal) you will have to close it and then open it to disable tutoring mode. Using the **Buffer Viewer** you can watch the goal buffer as the model solves the problem 36 plus 47. You can also use the **Stepper** to step through the individual production firings.