

Unit 7: Production Rule Learning

In this unit we will discuss how new production rules are learned. As we will see, new production rules can be acquired by collapsing two production rules that apply in succession into a single rule. In the process one can move knowledge that is stored declaratively into a procedural form. We call this process of forming new production rules **production compilation**.

7.1 The Basic Idea

A good pair of productions for illustrating production compilation is the two that fire in succession to retrieve a paired associate in the **paired** model from Unit 4:

```
(p read-probe
  =goal>
    isa      goal
    state    attending
    arg1     nil
  =visual>
    isa      text
    value    =val
  =visual-state>
    isa      module-state
    modality free
==>
  +retrieval>
    isa      goal
    state    associated
    arg1     =val
  =goal>
    arg1     =val
    state    testing
  -visual>
)

(p recall
  =goal>
    isa      goal
    arg1     =val
    state    testing
  =retrieval>
    isa      goal
    arg1     =val
    arg2     =ans
  =manual-state>
    ISA      module-state
    modality free
==>
  +manual>
    isa      press-key
    key      =ans
  =goal>
    state    read-study-item
)
```

If these two productions fired and retrieved the paired-associate zinc-9, production compilation would combine these two rules into the following single production:

```
(p recall-zinc
  =goal>
    isa GOAL
    state Attending
    arg1 nil
  =visual-state>
    isa MODULE-STATE
    modality Free
  =visual>
    isa TEXT
    value "zinc"
  =manual-state>
    isa MODULE-STATE
    modality Free
  ==>
  =goal>
    state Read-Study-Item
    arg1 "zinc"
  -visual>
  +manual>
    isa PRESS-KEY
    key "9"
)
```

Essentially, this production combines the work of the two and has built into it the paired associate. In the next two subsections we will describe generally the principles for combining two production rules together and the factors that control how these productions compete in the conflict resolution process.

7.2 Forming a New Production

The basic idea behind forming a new production is to combine the tests in the two conditions into a single set of tests that will recognize when the pair of productions will apply and combine the two actions into a single action that has the effect of both. Since the conditions consist of a set of buffer tests and the actions consist of a set of buffer transformations (either direct changes or new requests) this can be done largely on a buffer-by-buffer basis. The complications occur when there is a buffer transformation in the action of the first production and either a test of that buffer in the condition of the second production or another transformation of the same buffer in the action of the second productions. The productions above illustrate both complications with respect to the goal buffer. First, **read-probe** sets the state slot of the goal to **testing**. Then **recall** tests for that value in the slot. In this case, one can simply omit the setting of state to **testing** in the composed production. Also, the goal is changed in both productions but the resulting production can just produce the final changes. The result of the overlap in the goal buffer is just a simplification of the production rule but in other cases other responses are necessary.

7.2.1 Perceptual-Motor Buffers

Let us first consider the compilation policy for the perceptual-motor buffers. The buffers in question are **visual-location**, **visual**, **manual**, **aural**, and **vocal**. There are two things that one can do with each of these buffers. One is to test their contents on the left hand side and the other is to request an action on the right hand side. If the first production makes a request of one of these buffers then it is not possible to compose it with the second production if that production also mentions the same buffer. If both productions make a request, then there is a danger of jamming. If the first production makes a request and the second tests the value of the buffer after the request, then we cannot predict the outcome of the test in the future. Thus, points where a request is made of a perceptual-motor buffer are points where there are natural breaks in the compilation and one cannot compose the production that makes the request with any later production that operates on the same buffer.

7.2.2 Retrieval Buffer

Next let us consider the compilation policy for the retrieval buffer. Because it is an internal buffer (i.e., not subject to the whims of the outside world) it is more predictable and so offers an opportunity for economy. The interesting opportunity for economy occurs when the first production requests a retrieval and the second tests for the successful outcome of that retrieval. In this case, one can delete the request and test and instead specialize the two productions -- by replacing throughout the production any variables in the retrieval request by the constants they are bound to. This was what happened in the example production above where the retrieved paired-associate zinc-9 was built into the **recall-zinc** production. There is one case, however, where it is not possible to drop out the retrieval request and retrieval test. This is when the first production requests a retrieval and the second test for a retrieval error. This cannot be composed because declarative memory grows monotonically and it is not safe to predict that in the future there will be a retrieval error. This suggests that it is preferable, if possible, to write production rules that do not depend on retrieval failures.

7.2.3 Goal Buffer

The goal buffer is also an internal buffer allowing economies to be achieved. I will treat separately the cases where the first production involves requesting a new goal (the action contains **+goal>**) and when it does not (the action contains **=goal>**).

7.2.3.a First Production does not request a new goal.

Let C1 and C2 be the goal buffer conditions for the first and second production and A1 and A2 be the corresponding productions' goal buffer modification actions. Then, the goal buffer test for the combined production is essentially $C1+(C2-A1)$ where $C2-A1$ specifies those things tested in C2 that were not created in A1. The modification for the combined production is $A2+(A1\sim A2)$ where $(A1\sim A2)$ indicates those things that were in A1 that are not undone by A2. If the second production requests a new goal (**+goal>**) that can just be kept.

7.2.3.b First production requests a new goal.

This case breaks down into two subcases depending on whether the second production also changes the goal:

The second production does not also request a new goal. In this case the second goal test can be deleted since its satisfaction is guaranteed by the first production. Let C1 be the goal buffer condition of the first production, A1 be the goal buffer modification action of the first production, N1 be the new goal request in the first production, C2 be the goal condition in the second production and A2 be the goal modification of the second production. Then the goal test of the composed production is just C1, the goal modification is just A1, and the new goal request is $A2+(N1\sim A2)$.

The second production also requests a new goal. This is an ambiguous situation. A conservative policy would be to just block the compilation. A totally aggressive policy would just skip over the intermediate goal (i.e, the one created by the first and tested by the second production). What we have implemented is a special case that occurs when the goal set in the second production is exactly the same as the goal tested in the first. Then, the composed production would have test C1 and modification $N2+(A1\sim N2)$ with no new goal request. The consequence of this policy is that the chunk that represented the intermediate goal is not created. This might be viewed as automating behavior and dropping out intermediate declarative representations.

While these are the basic principles of compilation it is the case that there are numerous technical details needed to assure that the proper references are made to variables and constants in the composed productions. Some of these details will be illustrated in the example of Section 9.4.

7.3 Conflict Resolution

So far we have discussed how production rules are created but not how they are selected. When a new production **New** is composed from old productions **Old1** and **Old2**, it is the case that whenever **New** could apply **Old1** could also apply (Note because **New** might be specialized it does not follow that whenever **Old1** could apply **New** could also apply.) The choice between **New**, **Old1**, and whatever other productions might apply will be determined by their utilities. The utility of a production *i* is calculated as P_iG-C_i where *G* is the value of the goal, P_i is the expected probability that *i* will achieve the goal and C_i is the expected cost to achieve that goal. As we learned in Unit 8, the values P_i and C_i are learned from experience. However, there is no experience associated with the new production **New**. Therefore, how are these values to be assigned to that production? The probability of a production is defined in terms of its successes and failures as

$$P = \frac{\text{Successes}}{\text{Successes} + \text{Failures}} \quad \text{Probability Learning Equation}$$

where **Successes** and **Failures** are the number of experienced successes and failures. Therefore, the question of assigning an initial value of P comes down to the question of how to assign initial values to Successes and Failures. Similarly, the cost associated with a production is defined as

$$C = \frac{\text{Efforts}}{\text{Successes} + \text{Failures}} \quad \text{Cost Learning Equation}$$

where **Efforts** is the accumulated time over all the successful and failed applications of this production rule. Therefore, the question of assigning an initial value to C requires that we also assign an initial value to **Efforts**.

Roughly speaking the value of P and C for a new production **New** should be set based on the values of P and C for the production **Old1** that it competes with. The simple idea would be that the same estimates for P and C should apply. This is roughly the principle that applies but there are a couple of complications.

What we will do is to have a component in the equation that represents the experience of **Old1**, and a component that reflects the rule's own experience. For P this amounts to:

$$P = \frac{n * \text{priorP} + \text{Successes}}{n + \text{Successes} + \text{Failures}}$$

When the rule is first created, we set priorP to 0. Each time the rule is recreated, however, the value of priorP is increased so that it approaches the P value of **Old1**, on the basis of the formula:

$$\text{priorP} = \text{priorP}_{\text{previous}} + \alpha(\text{Old1P} - \text{priorP}_{\text{previous}})$$

This learning system has two parameters, n, which determines how many experiences the priorP value is worth (default 10, can be set by the :ie keyword in the sgp command), and α (default .05, which can be set by a (setf *pc-speed* value) command), which determines the speed of learning.

Similar equations determine the behavior of C:

$$C = \frac{n * \text{priorC} + \text{Efforts}}{n + \text{Successes} + \text{Failures}}$$

Here priorC is initially set to G (default 20), and is updated each time the rule is recreated by:

$$priorC = priorC_{previous} + \alpha(Old1C - priorC_{previous})$$

This leaves us with a system that is biased against new productions. However, each time a rule is recreated, its utility will get closer to the utility of the parent rule it competes with, until it is selected due to noise in the utility evaluation. Should the new rule prove to be better than the old, which is often the case, its conflict resolution parameters will come to dominate the old rule and so it will eventually become favored in the conflict resolution.

7.4 Learning from Instruction

Generally production compilation allows a problem to be solved with fewer productions over time and therefore performed faster. In addition to this speed-up, production compilation results in the drop-out of declarative retrieval as part of the task performance. As we saw in the example of the previous section, production rules are produced that just "do it" and do not bother retrieving the intervening information. The classic case of where this applies in experimental psychology is in the learning of experimental instructions. These instructions are told to the participant and initially the participant needs to interpret these declarative productions. However, with practice the participant will come to embed these instructions into productions that directly perform the task. These productions will be like the productions we normally write to model participant performance in the task. Essentially these are productions that participants learn in the warm-up phase of the experiment. The **paired** model for this assignment contains an example of a system that interprets instructions about how to perform a paired associate task and learns productions that do the task directly.

In the model are a set of chunks that encode declaratively the following knowledge about the structure of a paired associate task:

1. To do the experiment you are to read the stimulus, associate the stimulus with the response, act on the response, and repeat.
2. To associate a response with a stimulus, wait and read the response.
3. To act on an item, if you are still in the stimulus stage, type the item, and read the response.
4. Otherwise to act on an item just pass.

Below are a set of "rules" that encode this information. It would be too much of a digression from the main point of this lesson (production compilation) to give a total exposition of the system. This is done in the Appendix at the end of this unit. Nonetheless, the following are the most significant points:

A. Each point in the instructions above is represented as a rule for achieving a goal by trying to satisfy a sequence of clauses. Thus, `RULE102` below states that to achieve the experiment goal one satisfies the clauses of reading the stimulus (`P102`), associating the

stimulus with the response (P103), acting on the response (P104), and then repeating (P105).

B. Some clauses can be directly achieved and others require that one apply other rules. Thus, `RULE105` specifies how to associate the stimulus and response should it not be possible to retrieve the response. Rules `RULE106` and `RULE109` specify two ways to act on the response depending on whether one is in the stimulus or the response phase of the experiment.

```
(RULE102 ISA HEAD RELATION DO-EXPERIMENT PRIOR START)
(P102 ISA CLAUSE RELATION READ PRIOR RULE102 ARG1 VAR1)
(P103 ISA CLAUSE RELATION ASSOCIATE PRIOR P102 ARG1 VAR1 ARG2 VAR2)
(P104 ISA CLAUSE RELATION ACT PRIOR P103 ARG1 VAR2)
(P105 ISA CLAUSE RELATION REPEAT PRIOR P104)
(RULE105 ISA HEAD RELATION ASSOCIATE PRIOR START ARG1 VAR1 ARG2 VAR2)
(P106 ISA CLAUSE RELATION READ PRIOR RULE105 ARG1 VAR2)
(P107 ISA CLAUSE RELATION done PRIOR P106)
(RULE106 ISA HEAD RELATION ACT PRIOR START ARG1 VAR1)
(P108 ISA CLAUSE RELATION STILL-STIMULUS? PRIOR RULE106)
(P109 ISA CLAUSE RELATION TYPE PRIOR P107 ARG1 VAR1)
(P110 ISA CLAUSE RELATION READ PRIOR P108 ARG1 VAR2)
(P111 ISA CLAUSE RELATION done PRIOR P110)
(RULE109 ISA HEAD RELATION ACT PRIOR RULE102 ARG1 VAR1)
(P111 ISA CLAUSE RELATION done PRIOR RULE109)
```

The **paired** model contains a set of productions for interpreting instructions like this rather than productions for actually doing the **paired-associate** task. Otherwise, it is nearly identical to the paired model you had for Unit 4. It can be run either with production compilation on or off. This is controlled by an additional parameter to the functions **collect-data** and **do-experiment**. Below is a contrast of the behavior with compilation off (second argument to `collect-data` nil) and compilation on (second argument to `collect-data` t):

```
? (collect-data 10 nil)
```

Latency:

```
CORRELATION: 0.958
MEAN DEVIATION: 0.360
Trial      1      2      3      4      5      6      7      8
          0.000  2.176  2.177  2.063  2.010  2.077  1.920  1.962
```

Accuracy:

```
CORRELATION: 0.990
MEAN DEVIATION: 0.082
Trial      1      2      3      4      5      6      7      8
          0.000  0.555  0.655  0.730  0.825  0.785  0.860  0.830
```

NIL

```
? (collect-data 10 t)
```

Latency:

```
CORRELATION: 0.975
MEAN DEVIATION: 0.146
```

Trial	1	2	3	4	5	6	7	8
	0.000	2.015	1.983	1.962	1.829	1.764	1.535	1.205

Accuracy:

CORRELATION: 0.986

MEAN DEVIATION: 0.086

Trial	1	2	3	4	5	6	7	8
	0.000	0.575	0.580	0.725	0.740	0.805	0.880	0.905

As can be seen, whether learning is off or on has relatively little impact on the accuracy of recall but turning it on greatly increases the speed of recall. This is because we are cutting out productions and retrievals.

If you set the :v trace to t you will see the system print out its production compilations. For instance, the following is a fragment of the trace when we executed the command `(do-experiment 1 8 t)` -- to study one paired-associate for 8 trials with production compilation turned on.

```

Time 0.285: Read-Bind-Var1 Selected
Time 0.335: Read-Bind-Var1 Fired
Time 0.335: P6419 Retrieved
Time 0.335: Retrieve-*Var1-Var2 Selected
Compiling Production Production6439.
(p Production6439
  =goal>
    isa TASK
    arg1 Var1
    relation Read
    step Reading
    clause P6418
    var2 nil
  =visual>
    isa TEXT
    value =val
==>
  =goal>
    relation Associate
    arg1 =val
    arg2 Var2
    clause P6419
    step Retrieval-Harvest
    var1 =val
  +retrieval>
    isa TASK
    relation Associate
    arg1 =val
    - arg2 Var2
    - step Retrieval-Harvest
  -visual-location>
)
Parameters for production Production6439:
:Chance 1.000
:Effort 0.050
:P 1.000
:C 5.000
:PG-C 15.000
:Successes (10.0)
:Failures (0.0)
:Efforts (50.0)

```

```
:Success    nil
:Failure    nil
```

The production that was learned, **Production6439**, is a compilation of two of the original productions:

```
(P read-bind-var1
  "read-bind-var1"
  =goal>
    ISA      task
    arg1     var1
    relation read
    step     reading
    clause   =clause
  =visual>
    ISA      text
    value    =val
==>
  -visual-location>
  =goal>
    step     done
    relation nil
    arg1     nil
    var1     =val
  +retrieval>
    ISA      clause
    prior    =clause
)

(P retrieve-*var1-var2
  "retrieve-*var1-var2"
  =goal>
    ISA      task
    step     done
    var1     =val1
    var2     nil
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var1
    arg2     var2
==>
  =goal>
    relation =relation
    arg1     =val1
    arg2     var2
    clause   =retrieval
    step     retrieval-harvest
  +retrieval>
    ISA      task
    relation =relation
    arg1     =val1
  - arg2     var2
  - step     retrieval-harvest
)
```

The first production, **Read-Bind-Var1**, encodes the stimulus, puts that stimulus in its var1 slot, and retrieves the chunk that gives the next subgoal. This is the chunk **P6419**,

which asserts that we need to find what var1 is associated with. Production, **Retrieve-Var1-Var2**, attempts to retrieve that element.

```
P6419    7.921
  isa CLAUSE
  relation Associate
  arg1 Var1
  arg2 Var2
  prior P6418
```

Declarative retrieval of the instruction chunk **P6419** is compiled out and the resulting production just goes from the stimulus to a goal to test memory with that stimulus.

After 7 trials on this single-paired associate, the following represents the trace of the model on the eighth trial:

```
Time 70.000: Read-Attend Selected
Time 70.050: Read-Attend Fired
Time 70.050: Module :VISION running command MOVE-ATTENTION
Time 70.135: Module :VISION running command FOCUS-ON
Time 70.135: Read-Bind-Var1 Selected
Time 70.185: Read-Bind-Var1 Fired
Time 70.185: P6419 Retrieved
Time 70.185: Retrieve-*Var1-Var2 Selected
Recreating Production Production6439.
Time 70.235: Retrieve-*Var1-Var2 Fired
Time 71.025: Goal6442 Retrieved
Time 71.025: Production6523 Selected
Recreating Production Production6577.
Parameters for production Production6577:
:Chance 1.000
:Effort 0.050
:P 1.000
:C 17.000
:PG-C 3.000
:Successes (10.0)
:Failures (0.0)
:Efforts (169.9997973604053)
:Success nil
:Failure nil

Time 71.075: Production6523 Fired
Time 71.075: P6427 Retrieved
Time 71.075: Type-Var1 Selected
Compiling Production Production6598.
(p Production6598
 =goal>
  isa TASK
  arg2 Var2
  step Retrieval-Harvest
  clause P6419
 =retrieval>
  isa TASK
  arg2 =val
  !eval! (and (stimulus =goal) (equal (length =val) '1))
 ==>
 =goal>
  relation Act
  arg1 =val
```

```

    arg2 nil
    step Subgoaled
    clause P6420
    var2 =val
+goal>
    isa TASK
    relation nil
    arg1 nil
    clause P6427
    step Done
    rule Rule6425
    var1 =val
    parent =goal
+retrieval>
    isa CLAUSE
    prior P6427
+manual>
    isa PRESS-KEY
    key =val
)
Parameters for production Production6598:
:Chance 1.000
:Effort 0.050
:P 1.000
:C 6.000
:PG-C 14.000
:Successes (10.0)
:Failures (0.0)
:Efforts (60.00003213993572)
:Success nil
:Failure nil

Time 71.125: Type-Var1 Fired
Time 71.125: P6428 Retrieved
Time 71.125: Module :MOTOR running command PRESS-KEY
Time 71.125: Ready-To-Read Selected
Recreating Production Production6482.
Time 71.175: Ready-To-Read Fired
Time 71.175: Module :VISION running command CLEAR
Time 71.225: Module :MOTOR running command PREPARATION-COMplete
Time 71.225: Module :VISION running command CHANGE-STATE
Time 71.275: Module :MOTOR running command INITIATION-COMplete
Time 71.375: Device running command OUTPUT-KEY

<< Window "Paired-Associate Experiment" got key #\9 at time 71375 >>

Time 71.525: Module :MOTOR running command FINISH-MOVEMENT
Time 75.000: * Running stopped because time limit reached.
Time 75.000: Read-Attend Selected
Time 75.050: Read-Attend Fired
Time 75.050: Module :VISION running command MOVE-ATTENTION
Time 75.135: Module :VISION running command FOCUS-ON
Time 75.135: Read-Bind-Var2 Selected
Time 75.185: Read-Bind-Var2 Fired
Time 75.185: P6429 Retrieved
Time 75.185: Go-Back-1 Selected
Recreating Production Production6486.
Time 75.235: Go-Back-1 Fired
Time 75.622: Goal6590 Retrieved
Time 75.622: Go-Back-Side-Effect Selected
Merging chunk Goal6596 into chunk Goal6475
Time 75.672: Go-Back-Side-Effect Fired
Time 75.672: Production6462 Selected

```

```

Time 75.722: Production6462 Fired
Time 75.722: Experiment Retrieved
Time 75.722: Repeat-0-Arg Selected
Recreating Production Production6532.
Parameters for production Production6532:
:Chance 1.000
:Effort 0.050
:P 1.000
:C 5.550
:PG-C 14.450
:Successes (10.0)
:Failures (0.0)
:Efforts (55.49974863812315)
:Success t
:Failure nil

```

```

Merging chunk Goal6590 into chunk Start
Time 75.772: Repeat-0-Arg Fired
Time 75.772: Rule6417 Retrieved
Time 75.772: Production6437 Selected
Recreating Production Production6516.
Time 75.822: Production6437 Fired
Time 75.822: Module :VISION running command CLEAR
Time 75.872: Module :VISION running command CHANGE-STATE
Time 80.000: * Running stopped because time limit reached.

```

Notice that it involves a mixture of productions from the original production set plus new learned ones. Also we see that sometimes it is choosing old productions and forming the same new one. This occurs whenever there is a recreate message presented. The system avoids duplicating productions just as it avoids duplicating chunks.

Where the system to get to the point of maximal learning it would produce a trace like:

```

Time 70.000: Read-Attend Selected
Time 70.050: Read-Attend Fired
Time 70.050: Module :VISION running command MOVE-ATTENTION
Time 70.135: Module :VISION running command FOCUS-ON
Time 70.135: Production6657 Selected
Time 70.185: Production6657 Fired
Time 70.185: P6428 Retrieved
Time 70.185: Module :MOTOR running command PRESS-KEY
Time 70.185: Ready-To-Read Selected
Time 70.235: Ready-To-Read Fired
Time 70.235: Module :VISION running command CLEAR
Time 70.285: Module :MOTOR running command PREPARATION-COMplete
Time 70.285: Module :VISION running command CHANGE-STATE
Time 70.335: Module :MOTOR running command INITIATION-COMplete
Time 70.435: Device running command OUTPUT-KEY

```

<< Window "Paired-Associate Experiment" got key #\9 at time 70435 >>

```

Time 70.585: Module :MOTOR running command FINISH-MOVEMENT
Time 75.000: * Running stopped because time limit reached.
Time 75.000: Read-Attend Selected
Time 75.050: Read-Attend Fired
Time 75.050: Module :VISION running command MOVE-ATTENTION
Time 75.135: Module :VISION running command FOCUS-ON
Time 75.135: Production6486 Selected
Time 75.185: Production6486 Fired
Time 75.916: Goal6787 Retrieved

```

```

Time 75.916: Go-Back-Side-Effect Selected
Merging chunk Goal6792 into chunk Goal6475
Time 75.966: Go-Back-Side-Effect Fired
Time 75.966: Production6623 Selected
Merging chunk Goal6787 into chunk Start
Time 76.016: Production6623 Fired
Time 76.016: Module :VISION running command CLEAR
Time 76.066: Module :VISION running command CHANGE-STATE
Time 76.752: Goal6475 Retrieved
Time 80.000: * Running stopped because time limit reached.

```

The newly learned productions are:

```

(p Production6657
  =goal>
    isa TASK
    arg1 Var1
    relation Read
    step Reading
    clause P6418
    var2 nil
  =visual>
    isa TEXT
    value "zinc"
  !eval! (stimulus =goal)
==>
  =goal>
    relation Act
    arg1 "9"
    arg2 nil
    step Subgoaled
    clause P6420
    var2 "9"
    var1 "zinc"
  +goal>
    isa TASK
    relation nil
    arg1 nil
    clause P6427
    step Done
    rule Rule6425
    var1 "9"
    parent =goal
  +retrieval>
    isa CLAUSE
    prior P6427
  -visual-location>
  +manual>
    isa PRESS-KEY
    key "9"
)
(p Production6486
  =goal>
    isa TASK
    arg1 Var2
    relation Read
    step Reading
    clause P6428
    parent =oldgoal
  - parent Experiment
  =visual>
    isa TEXT
    value =val

```

```

==>
  =goal>
    step Go-Back
    parent nil
    relation nil
    arg1 nil
    var2 =val
  +retrieval>
    =oldgoal
  -visual-location>
)
(p Production6623
  =goal>
    isa TASK
    step Subgoaled
    clause P6420
  - arg1 Var1
  - arg1 Var2
  - arg2 Var1
  - arg2 Var2
    parent Experiment
    rule =rule
==>
  =goal>
    step Repeat
    relation nil
    arg1 nil
    arg2 nil
  +goal>
    isa TASK
    relation Read
    arg1 Var1
    arg2 nil
    clause P6418
    step Ready-To-Read
    rule Rule6417
    parent Experiment
  -visual>
)

```

The first production, **Production6657**, goes directly from the word "zinc" on the screen to the typing of the response. It is no longer necessary to retrieve the declarative chunk. If productions like these were always operative it would only take .435 seconds to respond. The other two productions are responsible for resetting the system after it has read the response to be ready for the stimulus on the next trial.

7.5 Assignment

Your assignment is to make a model that learns the past tense of verbs in English. The learning process of the English past tense is characterized by the so-called U-shaped learning in the learning of irregular verbs. That is, at a certain age children inflect irregular verbs like "to break" correctly, so they say "broke" if they want to use the past tense. But at a later age, they overgeneralize, and start saying "brokeed". At an even later stage they again inflect irregular verbs correctly. Some people, such as Pinker and Marcus, interpret this as evidence that a rule is learned to create regular past tense (add -ed to the stem). According to Pinker and Marcus, after this rule has been learned, it is overgeneralized so that it will also produce regularized versions of irregular verbs.

Part of the model is already given in the file past-tense. The assignment is to make a model that learns both the regular rule for the past tense, and particular rules for particular irregular verbs. So eventually it should learn rules like:

IF the goal is to make the past tense of a verb
THEN copy that verb and add –ed

IF the goal is to make the past tense of the verb have
THEN the past tense is had

The code that is provided does two things. It adds correct past tenses to declarative memory, reflecting the fact that a child hears and then encodes correct past tenses in the environment. It also creates goals to generate the past tense of a verb and then runs the model to generate one.

Here are examples of correctly formed past tenses:

WORD2323
 isa past-tense
 verb have
 stem had
 suffix blank
 status nil

is a correct encoding of the irregular verb have and:

WORD4323
 isa past-tense
 verb use
 stem use
 suffix ed
 status nil

is a correct encoding of the regular verb use.

A goal to generate a new past tense will look like this:

GOAL332
 isa past-tense
 verb get
 stem nil
 suffix nil
 status start

Your model has to fill in the stem and suffix slots of the goal to indicate the past tense form of the verb and set the status slot to done. Then, one of the three production rules given will fire to simulate the final encoding and “use” of the word. There are three possible cases. The first is an irregular inflection, in which the suffix is blank. There is a regular inflection, in which the stem is the same as the verb and the suffix is ed, and finally, there is a non-inflection case, in which both the stem and suffix are nil. The non-inflection case applies when the model cannot come up with a past tense at all, either because it has no example to retrieve or no rule or strategy to come up with anything else. The regular case and the non-inflection case each receive a cost penalty in the form of an additional effort added to the rule. The extra effort added to the regular rule reflects the fact that regular forms would take longer on average to say, and the penalty on the non-inflection rule reflects the fact that the past tense has to be indicated by some other method, for example by adding “yesterday” or some other explicit reference to time when it is actually used.

One important thing to notice is that all three of the provided productions are marked as a success. The model receives no feedback as to whether the past tenses it produces are correct – any past tense is considered a success. The only feedback it has are the correctly constructed verbs that it hears from the environment.

You can run the model with the **do-it** command. It takes as an argument the number of words you want the model to generate:

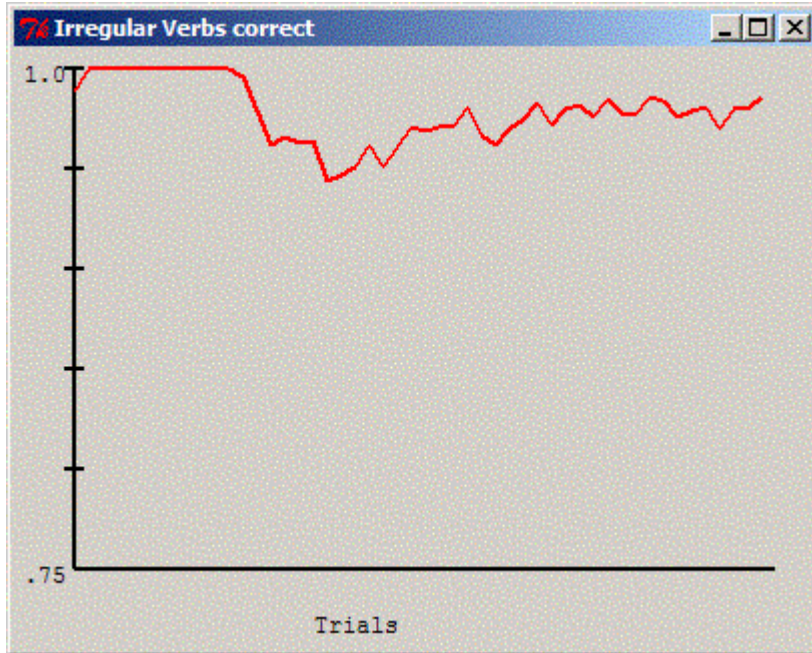
```
(do-it 5000)
```

As optional parameters you can specify whether or not you want ACT-R to be verbose (:v t), or whether ACT-R should continue with the run you started in an earlier **do-it** (:cont t).

During the run the simulation will display four numbers on each row, reflecting the results of the last 100 words. The first number is the proportion correct of irregular verbs. The second number is the proportion of irregular verbs that are inflected regularly. An increase in this number indicates a regular rule is active i.e. irregular verbs are having ed added to them. The third number is the proportion of irregular verbs that are not inflected at all. The fourth number is the proportion of inflected irregular verbs that are inflected correctly (the non-inflected verbs are not counted for this measure). It is in this last column that you should see a U-shape.

It often requires much more than 5000 trials to see the effect, and taking 30000 to 50000 trials is not uncommon. The :cont (continue) option in the **do-it** function allows you to run more trials without resetting the model. After the model is done, the **report-irreg** function gives a report where results are summarized for 1000 trials at a time. The 100 trial summaries displayed during the **do-it** function run are more to see the model is still doing things, and in what direction the results are going. If you have loaded the ACT-R

environment then you can pass `t` as a parameter to the **report-irreg** function to have it generate a graph of the data. What you are looking for from the model is a graph that looks something like this:



It starts out with a high percentage correct, dips down, and then goes back up. That is the U-shaped learning result.

This model differs from other models in the tutorial in that it does not model a particular experiment, but rather some long term development. This has a couple of consequences for the model. One of those is that using perceptual/motor modules does not contribute much to the objective of the model. Thus things like the "hearing" of past tenses and the eventual generation of the verb in speech are not modeled for the purpose of this exercise. It could be modeled, but it is not what the model and exercise are about. Therefore explicitly adding already processed perceived past tenses to declarative memory and just adding extra time for generation of certain classes of verb tenses serves as a reasonable compromise.

The other big consequence is that runs of the model may differ considerably. On the one hand this is not so bad, as children also differ with respect to U-shaped learning. One reason for the relative unpredictability is the fact that this simulation runs with a very limited vocabulary (extending the vocabulary results in a model that runs extremely slowly which is not beneficial as an exercise), but the effect of the noise in the model also has an impact that can create noticeable effects over the long term running of the model. This also makes comparing this model to data difficult, and hard data on the phenomenon are scarce, although the phenomenon of the U-shape is reported often. A few children have been followed in a longitudinal study, and there is a spreadsheet included with the unit materials (`data.xls`) that shows those results for comparison.

So, in terms of the assignment the objective is to write a model that learns the appropriate productions for producing past tenses. There is no parameter adjustment or data fitting required. The key to a successful model is to implement both a retrieval strategy and a simple analogy strategy. The model can either remember a correct past tense or the model attempts to generate a past tense based on another retrieved verb. The productions you write should make no explicit reference to either ed or blank because that is what the model is to eventually learn i.e. you do not write a production that says add ed, but though the compilation mechanism such a production is created. Although the experiment code is only outfitted with a limited set of words, the frequency that the words are presented to the model is in accordance with the frequency they appear in real life. So, if your model learns the proper productions it should generate the U-shaped learning automatically, but not necessarily on every run.

7.6 Appendix on Learning from Instruction

The basic idea behind the representation of instructions is that they should be represented as sequences of instructions about how to achieve goals. The instruction about how to achieve a goal is represented as a sequence of clauses that need to be achieved. Below in Prolog format is a set of instructions about how to perform a simple paired associate task. They encode essentially the following knowledge:

1. To do the experiment you are to read the stimulus, associate the stimulus with the response, act on the response, and repeat.
2. To associate a response with a stimulus, wait and read the response.
3. To act on an item, if you are still the stimulus stage, type the item, and read the response.
4. Otherwise to act on an item just pass.

The Prolog clauses are:

```
do-experiment :- read(Stimulus),associate(Stimulus,Response),
                act(Response), repeat.
associate(Probe,Answer):- read(Answer).
act(Item):- still-stimulus?, type (Item), read (Answer).
act(Item).
```

The main goal of do-experiment breaks down into a sequence of 4 clauses of reading the stimulus, finding the association between the stimulus and response, acting on the response, and a special goal of repeating. If it is not possible to retrieve the response to the stimulus, there is a special rule that says to simply read the answer. There is an ordered set of rules for acting on the response. The first rule tests that one is still in the stimulus stage, and if so types the item and read the answer. Otherwise if it is too late one just passes on any action.

This small example reflects most of the significant structure of the representation of instruction.

1. A rule for a goal is represented as an ordered sequence of clauses. Should it be not possible to satisfy one clause, the rule immediately fails. There is no backup (unlike Prolog).
2. The rules for a goal are tried in strict sequence and in this way one can arrange to have default rules tried only after special case rules.
3. Iteration is achieved with a special case repeat goal.
4. The terms capitalized above are variables. Rules have at most two variables.
5. Relations have at most two arguments.

The first three constraints enable a relatively simple control structure for instruction interpretation. The last two constraints minimize the number of cases we have to deal with. It does appear that this is adequate for full expressive power.

Obviously, we do not encounter instruction as prolog clauses. The above representation is just for my (and perhaps your) own conceptual clarity. I have developed a means of representing these prolog clauses as LISP structure and parsing them into internal ACT-R structures. The following is the LISP encoding of the above clauses:

```
(setf instructions '(
  (do-experiment read (stimulus) associate (stimulus response)
    act (response) repeat)
  (associate (probe answer) read (answer) done)
  (act (item) still-stimulus? type (item) read (answer) done)
  (act (item) done)))
```

Note that each clause requires a specification of whether it should be repeated (repeat) or whether it terminates (done). There is a LISP function parse that will convert these into ACT-R structure:

```
? (parse instructions)
```

The heads of these rules are represented as structures of type head and the clauses as structures of type clause. The following are the type definitions used in the instruction follower:

```
(chunk-type clause relation arg1 arg2 prior)
(chunk-type head relation arg1 arg2 prior)
(chunk-type task parent relation arg1 arg2 step rule clause var1 var2)
```

Heads and clauses have slots for their relation and (possibly as many as) 2 arguments plus a prior slot. The prior slot keeps track of the order of rules in the case of heads and the order of clauses in the case of clauses. The last chunk type is the task type that keeps track of the interpretation of a rule. It has a parent slot to keep track of what goal called it. It has slots to represent the relation and arguments it is currently interpreting and a

step slot to represent where it is in the interpretation of the clause. It has a rule slot to represent which rule it is working on and a clause slot to represent which clause it is working on within the rule. Finally it has two slots to keep track of the binding of the variables.

Interpretation of Instructions

Initiation of a Rule

A goal starts out as task with its relations and arguments filled in and the step set to achieve. The following production is responsible for selecting the first rule to apply to achieve that relation:

```
(P retrieve-rule
  "retrieve-rule"
  =goal>
    ISA      task
    relation  =relation
    step      achieve
==>
  +retrieval>
    ISA      head
    relation  =relation
    prior    start
  =goal>
    step     rule
)
```

Then, depending on the number of arguments that the relation takes, one of the following three rules applies:

```
(P instantiate-rule-0-args
  "instantiate-rule-0-args"
  =goal>
    ISA      task
    relation  =relation
    step     rule
  =retrieval>
    ISA      head
    arg1     nil
==>
  =goal>
    step     done
    relation nil
    rule     =retrieval
  +retrieval>
    ISA      clause
    prior    =retrieval
)
```

```
(P instantiate-rule-var1
  "instantiate-rule-var1"
  =goal>
    ISA      task
    relation  =relation
    arg1     =val
    arg2     nil
    step     rule
  =retrieval>
```

```

        ISA      head
        arg1     var1
        arg2     nil
==>
+retrieval>
  ISA      clause
  prior   =retrieval
=goal>
  step     done
  relation nil
  arg1     nil
  rule     =retrieval
  var1     =val
)

(P instantiate-rule-var1-var2
 "instantiate-rule-var1-var2"
=goal>
  ISA      task
  relation =relation
  arg1     =val1
  arg2     =val2
  step     rule
=retrieval>
  ISA      head
  arg1     var1
  arg2     var2
==>
+retrieval>
  ISA      clause
  prior   =retrieval
=goal>
  step     done
  relation nil
  arg1     nil
  arg2     nil
  var1     =val1
  var2     =val2
  rule     =retrieval
)

```

Note that these productions bind the arguments to variable slots in the goal so that they can be remembered while instantiating the clauses of that rule. The first argument in the head will always be bound to the var1 slot and the second argument to the var2 slot. The relation and argument slots will change as the focused clause changes and so they are set to nil. The var1 and var2 slots will remember the variable bindings. As we will see at the end of achieving a goal, the relation and argument slots will be set back to values for the head. A retrieval request is made for the first clause and the step slot is set to done as a signal that it is time to go onto the next clause.

There are three basic ways a clause can be satisfied. First, there might be some special productions for achieving that clause like how to read a word or move a mouse. Second, it is possible that the clause can be satisfied by retrieving some fact in the data base. Third, it may be necessary to try to apply some other declarative rule for achieving the clause. These three options are ordered.

Failure to Find a Rule

It may be the case that there are no rules for achieving that goal or that all the rules have been exhausted. In that case one needs to return failure the higher goal that called the current goal. This higher goal is contained in the parent slot of the current goal. The following production retrieves that goal noting that there has been a failure to achieve it:

```
(P retry-higher
  "retry-higher"
  =goal>
    ISA      task
    parent   =parent
  - parent   experiment
    step     rule
  =retrieval>
    ISA      error
==>
  +retrieval> =parent
  =goal>
    step     pop-failure
)
```

The term "experiment" in this situation refers to the top goal and the negative test in the above rule is a test to prevent popping out of the task.

To deal with the failure the parent goal has to be re-established and the system checks whether there are any other rules that might achieve the clause it was at. This process takes place in two steps. First a copy of the parent goal is created from which to try again:

```
(P pop-failure
  "pop-failure"
  =goal>
    ISA      task
    step     pop-failure
  =retrieval>
    ISA      task
    parent   =grandparent
    rule     =rule
==>
  +retrieval> =grandparent
  +goal>
    ISA      task
    step     try-again
    rule     =rule
    parent   =grandparent
)
```

The parent (grand parent goal if you like) of this parent goal is retrieved from which one can recreate the initial state of the goal being retried. There are three productions to deal with this depending on whether the parent goal has zero, one, or two arguments:

```
(P retry-0-arg
  "retry-0-arg"
  =goal>
    ISA      task
    step     try-again
    rule     =rule
```

```

=retrieval>
  ISA      task
  relation =rel
  arg1     nil
==>
+retrieval>
  ISA      head
  prior    =rule
=goal>
  relation =rel
  arg1     nil
  step     rule
)

(P retry-1-arg
 "retry-1-arg"
=goal>
  ISA      task
  step     try-again
  rule     =rule
=retrieval>
  ISA      task
  relation =rel
  arg1     =arg1
  arg2     nil
==>
+retrieval>
  ISA      head
  prior    =rule
=goal>
  relation =rel
  arg1     =arg1
  arg2     nil
  step     rule
)

(P retry-2-args
 "retry-2-args"
=goal>
  ISA      task
  step     try-again
  rule     =rule
=retrieval>
  ISA      task
  relation =rel
  arg1     =arg1
  arg2     =arg2
==>
+retrieval>
  ISA      head
  prior    =rule
=goal>
  relation =rel
  arg1     =arg1
  arg2     =arg2
  step     rule
)

```

In all cases a request is made to retrieve the rule that follows the rule last tried.

Special Instructions for Achieving a Clause

One can provide special productions for achieving any clause. The following is an example of the productions we use in the paired-associate model for typing

```
(P type-var1
  "type-var1"
  =goal>
    ISA      task
    step     done
    var1     =val
  =retrieval>
    ISA      clause
    relation type
    arg1     var1
    arg2     nil
    !eval!   (equal (length =val) 1)
  ==>
  +manual>
    ISA      press-key
    key      =val
  =goal>
    relation nil
    arg1     nil
    clause   =retrieval
    step     done
  +retrieval>
    ISA      clause
    prior    =retrieval
)
```

In this case there is a single production but sometimes it takes multiple productions to achieve a clause. All cases have to begin with a rule like this inspecting the retrieved clause and taking appropriate action. All have to end with a rule like this, setting the step slot to done and retrieving the next clause.

Retrieval to Instantiate a Clause

Sometimes the clause will have one argument bound and the other not. This is the situation in which we first try to retrieve another clause in memory that will fill in the missing value. There are eight possibilities created by whether it is the first or second variable we are trying to bind, by whether it is first or second argument, and by whether the other value is just given or bound to a variable:

```
(P retrieve-var1-val
  "retrieve-var1-val"
  =goal>
    ISA      task
    step     done
    var1     nil
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var1
    arg2     =val2
  - arg2     var2
  ==>
  =goal>
    relation =relation
```

```

    arg1      var1
    arg2      =val2
    clause    =retrieval
    step      retrieval-harvest
+retrieval>
  ISA        task
  relation   =relation
-  arg1      var1
  arg2      =val2
-  step      retrieval-harvest
)

```

```

(P retrieve-var1-*var2
 "retrieve-var1-*var2"
=goal>
  ISA        task
  step       done
  var1       nil
  var2       =val2
=retrieval>
  ISA        clause
  relation   =relation
  arg1       var1
  arg2       var2
==>
=goal>
  relation   =relation
  arg1       var1
  arg2       =val2
  clause     =retrieval
  step       retrieval-harvest
+retrieval>
  ISA        task
  relation   =relation
-  arg1       var1
  arg2       =val2
-  step       retrieval-harvest
)

```

```

(P retrieve-var2-val
 "retrieve-var2-val"
=goal>
  ISA        task
  step       done
  var2       nil
=retrieval>
  ISA        clause
  relation   =relation
  arg1       var2
  arg2       =val2
-  arg2       var1
==>
=goal>
  relation   =relation
  arg1       var2
  arg2       =val2
  clause     =retrieval
  step       retrieval-harvest
+retrieval>
  ISA        task
  relation   =relation
-  arg1       var2
  arg2       =val2

```

```

- step      retrieval-harvest
)

(P retrieve-var2-*var1
  "retrieve-var2-*var1"
  =goal>
    ISA      task
    step     done
    var1     =val2
    var2     nil
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var2
    arg2     var1
==>
  =goal>
    relation =relation
    arg1     var2
    arg2     =val2
    clause   =retrieval
    step     retrieval-harvest
  +retrieval>
    ISA      task
    relation =relation
  - arg1     var2
    arg2     =val2
  - step     retrieval-harvest
)

(P retrieve-val-var1
  "retrieve-val-var1"
  =goal>
    ISA      task
    step     done
    var1     nil
  =retrieval>
    ISA      clause
    relation =relation
    arg2     var1
    arg1     =val1
  - arg1     var2
==>
  =goal>
    relation =relation
    arg2     var1
    arg1     =val1
    clause   =retrieval
    step     retrieval-harvest
  +retrieval>
    ISA      task
    relation =relation
    arg1     =val1
  - arg2     var1
  - step     retrieval-harvest
)

(P retrieve-*var2-var1
  "retrieve-*var2-var1"
  =goal>
    ISA      task
    step     done
    var1     nil

```

```

    var2      =val2
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var2
    arg2     var1
==>
  =goal>
    relation =relation
    arg1     =val2
    arg2     var1
    clause   =retrieval
    step     retrieval-harvest
+retrieval>
    ISA      task
    relation =relation
    arg1     =val2
- arg2     var1
- step     retrieval-harvest
)

(P retrieve-val-var2
 "retrieve-val-var2"
=goal>
  ISA      task
  step     done
  var2     nil
=retrieval>
  ISA      clause
  relation =relation
  arg2     var2
  arg1     =vall
- arg1     var1
==>
  =goal>
    relation =relation
    arg2     var2
    arg1     =vall
    clause   =retrieval
    step     retrieval-harvest
+retrieval>
  ISA      task
  relation =relation
  arg1     =vall

- arg2     var2
- step     retrieval-harvest
)

(P retrieve-*var1-var2
 "retrieve-*var1-var2"
=goal>
  ISA      task
  step     done
  var1     =vall
  var2     nil
=retrieval>
  ISA      clause
  relation =relation
  arg1     var1
  arg2     var2
==>
  =goal>

```

```

    relation    =relation
    arg1        =vall
    arg2        var2
    clause      =retrieval
    step        retrieval-harvest
+retrieval>
  ISA          task
  relation     =relation
  arg1         =vall
-  arg2        var2
-  step        retrieval-harvest
)

```

All of these productions are given lower utilities to make sure they are only tried when there are no special case rules for that relation.

The following productions are responsible for harvesting successful retrievals. They deal with the four possibilities created by whether var1 or var2 is to be bound and whether it is arg1 or arg2:

```

(P harvest-var1a
  "harvest-var1a"
  =goal>
    ISA          task
    arg1         var1
    step         retrieval-harvest
    clause       =clause
  =retrieval>
    ISA          task
    arg1         =val
==>
  =goal>
    step         done
    relation     nil
    arg1         nil
    arg2         nil
    var1         =val
  +retrieval>
    ISA          clause
    prior        =clause
)

(P harvest-var2a
  "harvest-var2a"
  =goal>
    ISA          task
    arg1         var2
    step         retrieval-harvest
    clause       =clause
  =retrieval>
    ISA          task
    arg2         =val
==>
  =goal>
    step         done
    relation     nil
    arg1         nil
    arg2         nil
    var2         =val
  +retrieval>

```

```

        ISA      clause
        prior   =clause
    )
(P harvest-var1b
  "harvest-var1b"
  =goal>
    ISA      task
    arg2     var1
    step     retrieval-harvest
    clause   =clause
  =retrieval>
    ISA      task
    arg2     =val
==>
  =goal>
    step     done
    relation nil
    arg1     nil
    arg2     nil
    var1     =val
  +retrieval>
    ISA      clause
    prior   =clause
)

(P harvest-var2b
  "harvest-var2b"
  =goal>
    ISA      task
    arg2     var2
    step     retrieval-harvest
    clause   =clause
  =retrieval>
    ISA      task
    arg2     =val
==>
  =goal>
    step     done
    relation nil
    arg1     nil
    arg2     nil
    var2     =val
  +retrieval>
    ISA      clause
    prior   =clause
)

```

The following rule transitions to setting a subgoal to find the variable should the retrieval fail:

```

(P fail-harvest-var
  "fail-harvest-var"
  =goal>
    ISA      task
    relation =relation
    step     retrieval-harvest
  =retrieval>
    ISA      error
==>

```

```

+retrieval>
  ISA      head
  relation =relation
  prior    start
=goal>
  step      subgoal-var
)

```

Subgoaling Variables

In addition to the case of failure of retrieval, we will set a subgoal to find a value of a variable immediately when there is only a single unbound variable argument. There are two production rules corresponding to whether it is the first or second variable. As in the case of the productions that request retrieval these are given lower utility so that special case rules for the relation will be tried first.

```

(P subgoal-rule-var1
  "subgoal-rule-var1"
  =goal>
    ISA      task
    step      done
    var1      nil
  =retrieval>
    ISA      clause
    relation =relation
  - relation read
    arg1      var1
    arg2      nil
==>
  +retrieval>
    ISA      head
    relation =relation
    prior    start
  =goal>
    relation =relation
    arg1      var1
    step      subgoal-var
    clause    =retrieval
)

```

```

(P subgoal-rule-var2
  "subgoal-rule-var2"
  =goal>
    ISA      task
    step      done
    var2      nil
  =retrieval>
    ISA      clause
    relation =relation
  - relation read
    arg1      var2
    arg2      nil
==>
  +retrieval>
    ISA      head
    relation =relation
    prior    start
  =goal>
    relation =relation
    arg1      var2
    step      subgoal-var
    clause    =retrieval
)

```

)

Both of these productions request retrieval of a rule associated with that relation. The next productions deal with the processing of the retrieved rule. If there are two arguments for this retrieved rule, one will be bound upon the calling of the rule. In the productions below we create a new goal (+goal) and use the parent slot of the new goal to point to the original goal.

```
(P subgoal-find-first-arga
  "subgoal-find-first-arga"
  =goal>
    ISA      task
    relation =relation
    step     subgoal-var
  =retrieval>
    ISA      head
    arg1     var1
    arg2     nil
==>
  +retrieval>
    ISA      clause
    prior    =retrieval
  =goal>
    step     subgoal-ed
  +goal>
    ISA      task
    parent   =goal
    step     done
    rule     =retrieval
)
```

```
(P subgoal-find-first-argb
  "subgoal-find-first-argb"
  =goal>
    ISA      task
    relation =relation
    arg2     =val
  - arg2     var1
  - arg2     var2
    step     subgoal-var
    parent   =parent
  =retrieval>
    ISA      head
    arg1     var1
    arg2     var2
==>
  +retrieval>
    ISA      clause
    prior    =retrieval
  +goal>
    ISA      task
    var2     =val
    parent   =goal
    step     done
    rule     =retrieval
  =goal>
    step     subgoal-ed
)
```

```
(P subgoal-find-second-arg
  "subgoal-find-second-arg"
```

```

=goal>
  ISA      task
  relation =relation
  arg1     =val
-  arg1    var1
-  arg1    var2
  step     subgoal-var
  parent   =parent
=retrieval>
  ISA      head
  arg1     var1
  arg2     var2
==>
+retrieval>
  ISA      clause
  prior    =retrieval
+goal>
  ISA      task
  var1     =val
  parent   =goal
  step     done
  rule     =retrieval
=goal>
  step     subgoaled
)

```

Finally, we have to deal with the case when there is no rule that applies:

```

(P fail-subgoal-var
 "fail-subgoal-var"
=goal>
  ISA      task
  step     subgoal-var
  parent   =parent
=retrieval>
  ISA      error
==>
=goal>
  step     try-again
  relation nil
  arg1     nil
  arg2     nil
+retrieval> =parent
)

```

Returning from Subgoals

The following is the production that recognizes generally when a goal has been achieved. This occurs when one reaches the ending done clause:

```

(P go-back-1
 "go-back-1"
=goal>
  ISA      task
  step     done
  parent   =oldgoal
-  parent   experiment
=retrieval>
  ISA      clause
  relation done
==>

```

```

+retrieval>   =oldgoal
=goal>
  step        go-back
  parent      nil
)

```

For our purposes "experiment" is considered to be the top goal and the test is just to prevent us from jumping out of the experiment. This production requests retrieval of the parent goal.

Before returning to the parent goal an attempt is made to set the subgoal to a state that contains the answer (so it can be retrieved next time and so avoid the need for a subgoal). This is determined by inspecting the parent goal and filling in the variable. There are 4 cases if the variable is in the first slot depending on whether it is var1 or var2 and whether the second argument is filled or not. If the variable is in the second slot there are two cases depending on whether it is var1 or var2

```

(P go-back-arg1a
  "go-back-arg1a"
  =goal>
    ISA      task
    step     go-back
    var1     =arg1
  =retrieval>
    ISA      task
    relation =rel
    arg1     var1
    arg2     =arg2
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     =arg2
  +retrieval> =goal
  +goal>     =retrieval
)

```

```

(P go-back-arg1b
  "go-back-arg1b"
  =goal>
    ISA      task
    step     go-back
    var1     =arg1
  =retrieval>
    ISA      task
    relation =rel
    arg1     var2
    arg2     =arg2
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     =arg2
  +retrieval> =goal
  +goal>     =retrieval
)

```

```

(P go-back-arg1c
  "go-back-arg1c"
  =goal>
    ISA      task
    step     go-back
    var1     =arg1
  =retrieval>
    ISA      task
    relation =rel
    arg1     var1
    arg2     nil
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     nil
  +retrieval> =goal
  +goal>     =retrieval
)

```

```

(P go-back-arg1d
  "go-back-arg1d"
  =goal>
    ISA      task
    step     go-back
    var1     =arg1
  =retrieval>
    ISA      task
    relation =rel
    arg1     var2
    arg2     nil
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     nil
  +retrieval> =goal
  +goal>     =retrieval
)

```

```

(P go-back-arg2a
  "go-back-arg2a"
  =goal>
    ISA      task
    step     go-back
    var2     =arg2
  =retrieval>
    ISA      task
    relation =rel
    arg1     =arg1
    arg2     var2
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     =arg2
  +retrieval> =goal
  +goal>     =retrieval
)

```

```

(P go-back-arg2b
  "go-back-arg2b"
  =goal>
    ISA      task
    step     go-back
    var2     =arg2
  =retrieval>
    ISA      task
    relation =rel
    arg1     =arg1
    arg2     var1
    step     subgoaled
==>
  =goal>
    relation =rel
    arg1     =arg1
    arg2     =arg2
  +retrieval> =goal
  +goal>     =retrieval
)

```

Each of these productions switches attention back to the parent goal and requests that the subgoal be put in the retrieval buffer to be processed by the next production. This next productions deal with harvesting when we return from the subgoal. There are 4 cases to be dealt with depending on whether the variable was in arg1 or arg2 slot and whether that variable was var1 or var2. In the productions below the chunk retrieved (bound to =retrieval) is always the subgoal that we had created. If the variable is in arg1 of the goal its value will be in the var1 slot of the subgoal and if it is in arg2 its value will be in the var2 slot:

```

(P harvest-subgoal-var1a
  "harvest-subgoal-var1a"
  =goal>
    ISA      task
    arg1     var1
    step     subgoaled
    clause   =clause
  =retrieval>
    ISA      task
    var1     =val
==>
  +retrieval>
    ISA      clause
    prior    =clause
  =goal>
    arg1     =val
    var1     =val
    step     done
)

```

```

(P harvest-subgoal-var1b
  "harvest-subgoal-var1b"
  =goal>
    ISA      task
    arg2     var1
    step     subgoaled
    clause   =clause
  =retrieval>
    ISA      task

```

```

    var2      =val
==>
+retrieval>
  ISA      clause
  prior    =clause
=goal>
  arg2     =val
  var1     =val
  step     done
)

(P harvest-subgoal-var2a
 "harvest-subgoal-var2a"
=goal>
  ISA      task
  arg2     var2
  step     subgoalied
  clause   =clause
=retrieval>
  ISA      task
  var2     =val
==>
+retrieval>
  ISA      clause
  prior    =clause
=goal>
  arg2     =val
  var2     =val
  step     done
)

(P harvest-subgoal-var2b
 "harvest-subgoal-var2b"
=goal>
  ISA      task
  arg1     var2
  step     subgoalied
  clause   =clause
=retrieval>
  ISA      task
  var1     =val
==>
+retrieval>
  ISA      clause
  prior    =clause
=goal>
  arg1     =val
  var2     =val
  step     done
)

```

In all of these cases we retrieve the next clause to be achieved for that goal.

Subgoals as Side Effects

The other case for subgoaling is when a clause is reached for which all the arguments are specified and for which there are no special-case production rules. This happens when some subgoal is created to achieve some side effect (like typing a word). We need separate rules corresponding to whether the relation takes 0, 1, or 2 arguments. Only one rule is needed for zero arguments:

```

(P subgoal-0-arg
  "subgoal-0-arg"
  =goal>
    ISA      task
    step     done
  =retrieval>
    ISA      clause
    relation =relation
  - relation done
  - relation still-stimulus
    arg1     nil
==>
  =goal>
    step     subgoaled
    clause   =retrieval
    relation =relation
    arg1     nil
    arg2     nil
  +goal>
    ISA      task
    relation =relation
    parent   =goal
    step     achieve
)

```

In the case of 1 argument that argument can be given outright or as the value of var1 or var2 leading to three separate rules:

```

(P subgoal-1-given
  "subgoal-1-given"
  =goal>
    ISA      task
    step     done
  =retrieval>
    ISA      clause
    relation =relation
    arg1     =val
  - arg1     var1
  - arg1     var2
    arg2     nil
==>
  =goal>
    relation =relation
    arg1     =val
    arg2     nil
    step     subgoaled
    clause   =retrieval
  +goal>
    ISA      task
    relation =relation
    arg1     =val
    parent   =goal
    step     achieve
)

```

```

(P subgoal-1-var1
  "subgoal-1-var1"
  =goal>
    ISA      task
    step     done
    var1     =val
  =retrieval>

```

```

        ISA      clause
        relation =relation
-       relation read
-       relation type
        arg1     var1
        arg2     nil
==>
=goal>
        relation =relation
        arg1     =val
        arg2     nil
        step     subgoaled
        clause   =retrieval
+goal>
        ISA      task
        relation =relation
        arg1     =val
        parent   =goal
        step     achieve
)

(P subgoal-1-var2
  "subgoal-1-var2"
  =goal>
        ISA      task
        step     done
        var2     =val
  =retrieval>
        ISA      clause
        relation =relation
-       relation read
-       relation type
        arg1     var2
        arg2     nil
==>
=goal>
        relation =relation
        arg1     =val
        arg2     nil
        step     subgoaled
        clause   =retrieval
+goal>
        ISA      task
        relation =relation
        arg1     =val
        parent   =goal
        step     achieve
)

```

In the case of two arguments there are nine (3x3) logical possibilities:

```

(P subgoal-2-given-given
  "subgoal-2-given-given"
  =goal>
        ISA      task
        step     done
  =retrieval>
        ISA      clause
        relation =relation
        arg1     =val1
-       arg1     var1
-       arg1     var2

```

```

    arg2      =val2
  - arg2      var1
  - arg2      var2
==>
=goal>
  relation    =relation
  arg1        =val1
  arg2        =val2
  step        subgoaled
  clause      =retrieval
+goal>
  ISA         task
  relation    =relation
  arg1        =val1
  arg2        =val2
  parent      =goal
  step        achieve
)

```

```

(P subgoal-2-given-var1
  "subgoal-2-given-var1"
=goal>
  ISA         task
  step        done
  var1        =val2
=retrieval>
  ISA         clause
  relation    =relation
  arg1        =val1
  - arg1      var1
  - arg1      var2
  arg2        var1
==>
=goal>
  relation    =relation
  arg1        =val1
  arg2        =val2
  step        subgoaled
  clause      =retrieval
+goal>
  ISA         task
  relation    =relation
  arg1        =val1
  arg2        =val2
  parent      =goal
  step        achieve
)

```

```

(P subgoal-2-given-var2
  "subgoal-2-given-var2"
=goal>
  ISA         task
  step        done
  var2        =val2
=retrieval>
  ISA         clause
  relation    =relation
  arg1        =val1
  - arg1      var1
  - arg1      var2
  arg2        var2
==>
=goal>

```

```

        relation    =relation
        arg1        =val1
        arg2        =val2
        step        subgoaled
        clause      =retrieval
+goal>
    ISA            task
    relation       =relation
    arg1           =val1
    arg2           =val2
    parent        =goal
    step           achieve
)

(P subgoal-2-var1-given
  "subgoal-2-var1-given"
=goal>
    ISA            task
    step           done
    var1           =val1
=retrieval>
    ISA            clause
    relation       =relation
    arg1           var1
    arg2           =val2
-   arg2           var1
-   arg2           var2
==>
=goal>
    relation       =relation
    arg1           =val1
    arg2           =val2
    step           subgoaled
    clause         =retrieval
+goal>
    ISA            task
    relation       =relation
    arg1           =val1
    arg2           =val2
    parent        =goal
    step           achieve
)

(P subgoal-2-var1-var1
  "subgoal-2-var1-var1"
=goal>
    ISA            task
    step           done
    var1           =val2
    var1           =val1
=retrieval>
    ISA            clause
    relation       =relation
    arg1           var1
    arg2           var1
==>
=goal>
    relation       =relation
    arg1           =val1
    arg2           =val2
    step           subgoaled
    clause         =retrieval
+goal>

```

```

        ISA      task
        relation =relation
        arg1     =val1
        arg2     =val2
        parent   =goal
        step     achieve
    )

(P subgoal-2-var1-var2
  "subgoal-2-var1-var2"
  =goal>
    ISA      task
    step     done
    var2     =val2
    var1     =val1
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var1
    arg2     var2
==>
  =goal>
    relation =relation
    arg1     =val1
    arg2     =val2
    step     subgoaled
    clause   =retrieval
  +goal>
    ISA      task
    relation =relation
    arg1     =val1
    arg2     =val2
    parent   =goal
    step     achieve
)

(P subgoal-2-var2-given
  "subgoal-2-var2-given"
  =goal>
    ISA      task
    step     done
    var2     =val1
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var2
    arg2     =val2
  - arg2     var1
  - arg2     var2
==>
  =goal>
    relation =relation

    arg1     =val1
    arg2     =val2
    step     subgoaled
    clause   =retrieval
  +goal>
    ISA      task
    relation =relation
    arg1     =val1
    arg2     =val2
    parent   =goal

```

```

        step      achieve
    )
(P subgoal-2-var2-var1
  "subgoal-2-var2-var1"
  =goal>
    ISA      task
    step     done
    var1     =val2
    var2     =val1
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var2
    arg2     var1
==>
  =goal>
    relation =relation
    arg1     =val1
    arg2     =val2
    step     subgoaled
    clause   =retrieval
  +goal>
    ISA      task
    relation =relation
    arg1     =val1
    arg2     =val2
    parent  =goal
    step     achieve
)

(P subgoal-2-var2-var2
  "subgoal-2-var2-var2"
  =goal>
    ISA      task
    step     done
    var2     =val2
    var2     =val1
  =retrieval>
    ISA      clause
    relation =relation
    arg1     var2
    arg2     var2
==>
  =goal>
    relation =relation
    arg1     =val1
    arg2     =val2
    step     subgoaled
    clause   =retrieval
  +goal>
    ISA      task
    relation =relation
    arg1     =val1
    arg2     =val2
    parent  =goal
    step     achieve
)

```

All of the above productions are given lower utilities to reflect the fact that they should only be called when there are not special case rules. Note that this is unlike the case when subgoals are created to bind a variable. In those cases we retrieved a rule in the

production that subgoal. In this case the goal is just set with an achieve step and control is tossed back to the top. Thus, these subgoals are treated as full goals like the original goals.

There is one production that returns control from such a side effect effort:

```
(P go-back-side-effect
  "go-back-side-effect"
  =goal>
    ISA      task
    step     go-back
  =retrieval>
    ISA      task
  - arg1    var1
  - arg1    var2
  - arg2    var1
  - arg2    var2
    step     subgoal
==>
  +retrieval> =goal
  +goal>      =retrieval
)
```

The following production recognizes when we have returned from such a side effect goal (because there are no variables to be bound) and moves on to the next clause.

```
(P forward-subgoal-default
  "forward-subgoal-default"
  =goal>
    ISA      task
    step     subgoal
    clause   =clause
  - arg1    var1
  - arg1    var2
  - arg2    var1
  - arg2    var2
==>
  =goal>
    step     done
    relation nil
    arg1     nil
    arg2     nil
  +retrieval>
    ISA      clause
    prior    =clause
)

(spp forward-subgoal-default :success t)
```

This is set as a success because it is recognized as the sign that a significant goal has been achieved.

Repeating

Repeat is a special clause that resets the goal. It finds the parent goal of the current goal and reinstantiates the clause in that parent goal. The first production retrieves the parent

goal and then there are three second productions to deal with the different number of arguments that this clause may have had:

```
(P repeat
  "repeat"
  =goal>
    ISA      task
    step     done
    parent   =parent
  =retrieval>
    ISA      clause
    relation repeat
  ==>
  =goal>
    step     repeat
  +retrieval>
    =parent
)
```

```
(P repeat-0-arg
  "repeat-0-arg"
  =goal>
    ISA      task
    parent   =parent
    step     repeat
    rule     =rule
  =retrieval>
    ISA      task
    relation =rel
    arg1     nil
    arg2     nil
  ==>
  +retrieval>
    ISA      head
    relation =rel
    prior    start
  +goal>
    ISA      task
    relation =rel
    arg1     nil
    arg2     nil
    step     rule
    parent   =parent
)
```

```
(P repeat-1-arg
  "repeat-1-arg"
  =goal>
    ISA      task
    parent   =parent
    step     repeat
    rule     =rule
  =retrieval>
    ISA      task
    relation =rel
    arg1     =arg1
    arg2     nil
  ==>
  +retrieval>
    ISA      head
    relation =rel
    prior    start
  +goal>
```

```

        ISA      task
        relation =rel
        arg1     =arg1
        arg2     nil
        step     rule
        parent   =parent
    )

(P repeat-2-args
  "repeat-2-args"
  =goal>
    ISA      task
    parent   =parent
    step     repeat
    rule     =rule
  =retrieval>
    ISA      task
    relation =rel
    arg1     =arg1
    arg2     =arg2
  ==>
  +retrieval>
    ISA      head
    relation =rel
    prior    start
  +goal>
    ISA      task
    relation =rel
    arg1     =arg1
    arg2     =arg2
    step     rule
    parent   =parent
  )

```