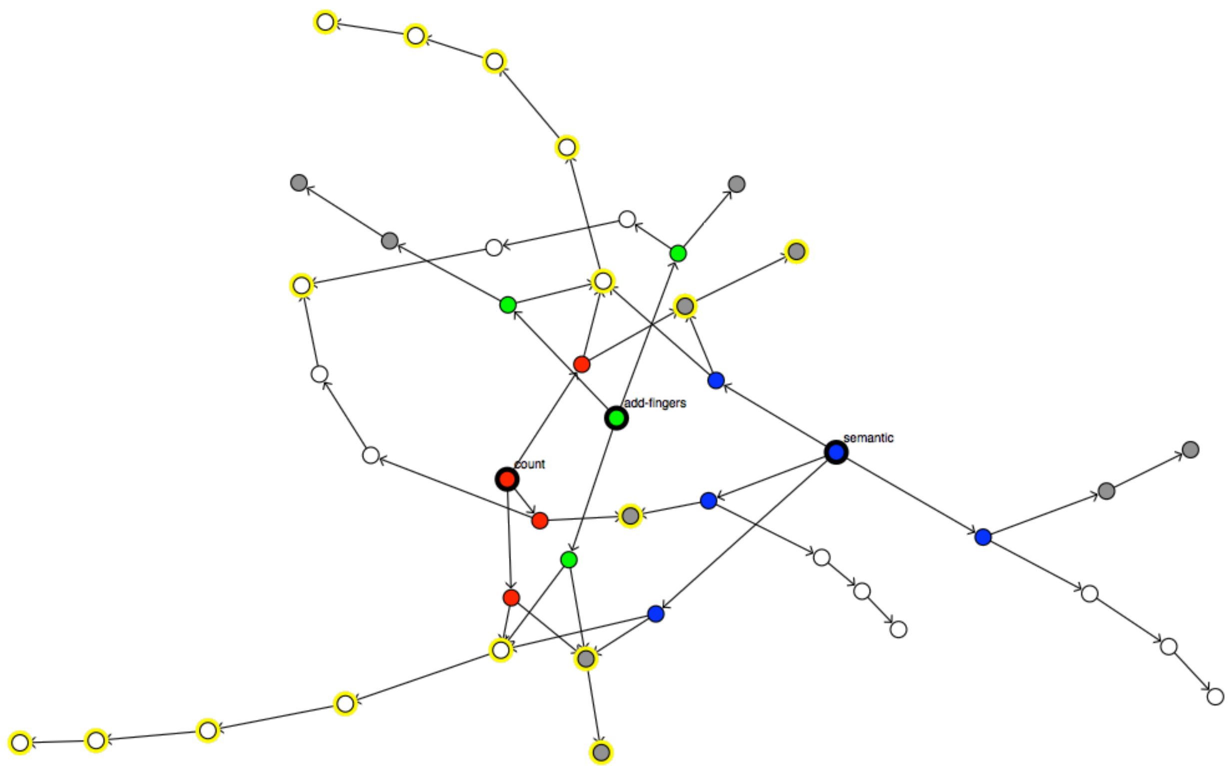
PRIMS TUTORIAL



Niels Taatgen
March 2017

Funded by ERC StG
283597 MULTITASK

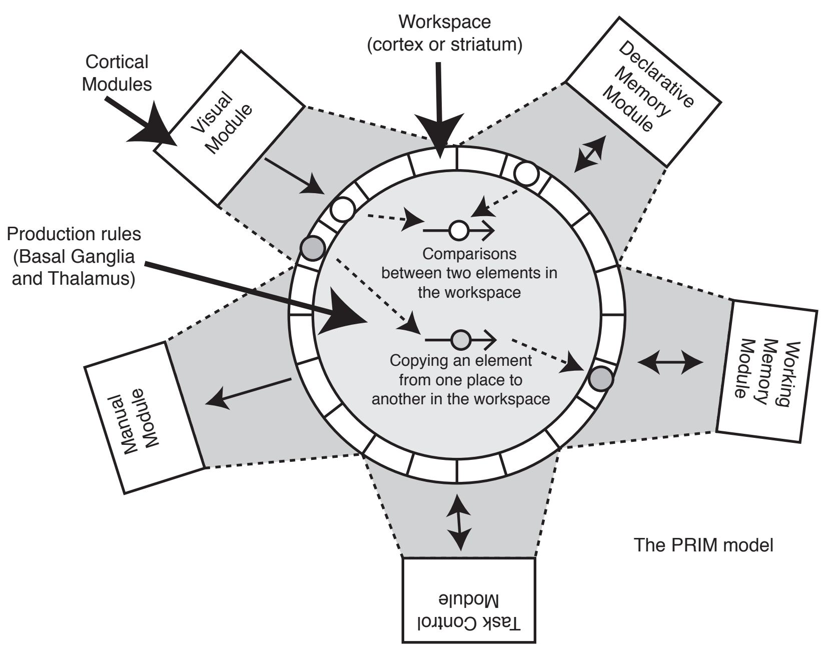
UNIT 1

Introduction to PRIMs

The PRIMs cognitive architecture has evolved from the ACT-R cognitive architecture (Anderson, 2007). Although familiarity with ACT-R is useful, no prior ACT-R knowledge is necessary for this tutorial. We will start the tutorial with some concepts that have been adapted from ACT-R before starting with the structure of a PRIMs model.

Main concepts

Like many other current cognitive architectures, PRIMs is modular. It assumes the cognitive system has specific modules for vision, motor control, goal focus, declarative memory, working memory, etc. Modules can operate in parallel, but can only do one thing at a time. Communication between these modules is carried out through buffers. Each buffer has a number of slots through which it can exchange information with other buffer, where each slot can hold a single piece of information. All the buffers together comprise the (global) workspace of the system.



The PRIM model

The core PRIMs theory focusses on how information is exchanged within this workspace. The process of information exchange is carried out by production rules that do simple comparisons and copy operations between slots in the workspace. However, the control of this process is determined by declarative knowledge. Declarative memory contains both knowledge of facts (3 +4 = 7, an elephant is a mammal, my birthday is 6 May, etc.) as well as knowledge on how to do things (how to solve a multi-column addition, how to count a number of marbles, how to ride a bicycle), in other words procedural knowledge that many other architectures (including ACT-R) assume resides in a specific procedural memory.

Given the importance of declarative memory, we discuss it in a bit more detail.

Declarative Memory

The elements in declarative memory are called *chunks*. A chunk is a unit of information that consists of a name and a set of attribute-value pairs. A typical chunk that might represent the fact that 3 + 4 = 7 may look like:

```
FACT347
        SLOT1 ADDITION-FACT
        SLOT2 THREE
        SLOT3 FOUR
        SLOT4 SEVEN
```

In this example the name of the chunk is `FACT347`. The name is in this case fairly meaningless, but sometimes we want to give chunks a meaningful name so that it is easier for us to read a model. The chunk has four attributes. Contrary to most cognitive architectures, attributes do not have names, but just numbers. Nevertheless the order of the attributes matters, because in this particular case `SLOT2` and `SLOT3` contain the numbers that are added, while `SLOT4` holds the answer. This fact is not represented anywhere explicitly, but is implicit in the knowledge that knows how to do calculations with numbers (as we will see later on). The values connected to the attributes refer to other chunks, `ADDITION-FACT`, `THREE`, `FOUR` and `SEVEN`. Most of the time chunks refer to other chunks, although a value may also be a number.

Each chunk has an activation value. The activation value is comprised of two main components. The first is *base-level activation* that represents how frequent and recent the chunk has been used, and therefore represents the history of a chunk. Chunks that have been used more frequently and recently receive a higher activation, so base-level activation represents the natural decay and repetition effects of memory. The second component is spreading activation from the current context. Chunks can have a strength of association with each other. Chunks that are already in the workspace and that are associated with chunks in declarative memory spread activation to those chunks. For example, if `THREE` and `FOUR` are already in the workspace, then `FACT347` will receive extra activation because `THREE` and `FOUR` each have a positive association with `FACT347`.

Activation has three purposes. The first is *selection*. If two chunks both match the retrieval request, the chunk with the highest activation will be chosen. On each cycle, some noise will be added/subtracted from the activation, so selection is a stochastic process.

The second is *forgetting*. If the activation of a chunk drop below a certain retrieval threshold, the chunk cannot be retrieved from memory.

The third is that activation determines the time that is necessary to retrieve the chunk. A higher activation translates into a short retrieval time.

The details of how activations are calculated can be find at the end of this unit.

The structure of a PRIMs model

A PRIMs model consists of several components. It contains the name of the task with some parameters, the specification of a set of operators to perform the task, possibly additional facts necessary to carry out the task, and a script that simulates the environment or experiment.

The example we will use consists of two models: count and semantic. We will see that there can be quite a bit of transfer between the two tasks, even though they are not very similar at first glance.

Let us look at the file `count.prims`. It starts by specifying what task we are going to carry out:

```
define task count {
  initial-goals: (count)
  default-activation: 1.0
  ol: t
  rt: -2.0
  lf: 0.2
  default-operator-self-assoc: 0.0
  egs: 0.05
  retrieval-reinforces: t
}
```

In PRIMs, a task can be implemented by several goals, but the count task is only implemented by one goal, also named count. The `initial-goals: (count)` line specifies this.

The remaining lines in the task definition are model parameters, some of which are straight from ACT-R (`ol:` optimized learning, `rt:` retrieval threshold, `lf:` latency factor, `egs:` utility noise). The model's parameters are not particularly critical for the operation of this model, so we will not discuss them yet.

The next part of the model defines operators to carry out the task. Operators are organized within goals. We have only one goal, the count goal, so we define it as:

```
define goal count {
… Operator definitions …
}
```

The result of this organization is that all the operators within a goal definition will be associated with that goal, that is, whenever a goal is in one of the goal buffer slots, the associated operators receive spreading activation and are therefore more likely to be retrieved.

We then see definitions of the three operators that are needed to count. The first is:

```
operator start-count {
 V1 <> nil  // There has to be a visual input with the starting number
 WM1 = nil  // Imaginal should be empty
==>
 V1 -> WM1 // Copy the start number to working memory
 count-fact -> RT1 // Start retrieving the next number
 V1 -> RT2
 say -> AC1 // Say the current number
 V1 -> AC2
 }
```

As the name implies, this operator initiates counting. Each operator consists of one or more condition PRIMs, and one or more action PRIMs, each on a separate line. PRIMs always refer to two specific slots in two buffers (or one slot in one buffer and nil). In between is either a comparison (= or <>), or a copy operation (->). An arrow ==> separates the conditions and actions, and anything after `//` is ignored.

`nil` is used to denote that a slot is empty, and can therefore be used to check whether a slot is empty or not empty, and can also be used to clear a slot (e.g., `nil -> WM1`).

The letters indicate the buffer, and the number the slot number within that buffer. In this example, we will use the following buffers:

- `V`: Visual, or input buffer. This buffer contains the visual input. In this particular model, `V1` has the starting number for counting, and `V2` the ending number
- `WM`: Working memory or imaginal buffer. This buffer is used to store intermediate results. Here we only use one slot to represent the current count.
- `RT`: Retrieval. Used to retrieve items from declarative memory. In this model it is used to retrieve count-facts from memory.
- `AC`: Action. Used specify actions the model takes. This model will use it to “say” numbers.
- `G`: Goal. Goal slots are used to activate operators that are relevant for the current goal. Because of our initial-goals declaration, `count` will be put into `G1`.

- `C`: Constant. This is not actually a buffer, and it also doesn't show up in the operator definition. However, each time a PRIM in an operator has a constant in it, that is, a specification that is not a buffer/slotnumber combination (and not `nil`), that constant is put in one of the slots of the operator chunk. In our example, `count-fact` and `say` are both constants that will be put into `C1` and `C2`, respectively. Although it appears in the syntax as if we can also use constants in PRIMs, "under the hood" they are replaced by buffer slots as well.

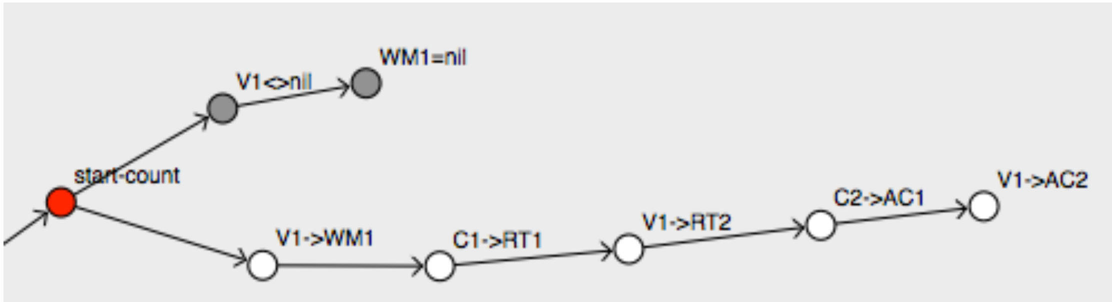
If an operator is selected, its conditions are checked first. In this case, a check is made whether `V1` is not empty, denoted by `V1 <> nil`. The second check is to see whether `WM1` is empty, indicating we haven't started counting yet.

If these conditions are satisfied, the model will start carrying out its actions. Typically, actions involve updating `WM` slots and specifying actions that are carried out by the modules. In this example, we first store the input into working memory in order to maintain a counter: `V1 -> WM1`. We then specify a retrieval request, which consists of two PRIMs: `count-fact -> RT1` specifies `slot1` of the retrieval, and `V1 -> RT2` `slot2` of the retrieval. In other words, we want a count-fact about the starting number. Furthermore, we want to say that number, which is specified by `say -> AC1` and `V1 -> AC2`.

Although an operator does not really look like declarative chunk, it consists of a small structure of chunks. The PRIMs parser takes care of the translation process, which will look like:

```
start-count
  isa  operator
  slot1  count-fact
  slot2  say
  condition  V1<>nil;WM1=nil
  action  V1->WM1;C1->RT1;V1->RT2;C2->AC1;V1->AC2
```

The chunks in the condition and action slots represent the first in a list of PRIMs. `V1->WM1;C1->RT1;V1->RT2;C2->AC1;V1->AC2` is a PRIM that carries out `V1->WM1` and points to `C1->RT1;V1->RT2;C2->AC1;V1->AC2`, which in turn carries out `C1->RT1`, and points to `V1->RT2;C2->AC1;V1->AC2`, etcetera. The following figure visualizes the structure of the operator, in which each node is a chunk:



The second and third operator are as follows:

```
operator iterate {
  RT2 = WM1
  V2 <> WM1
 ==>
  RT3 -> WM1
  count-fact -> RT1
  RT3 -> RT2
  say -> AC1
  RT3 ->AC2
}

operator final {
  V2 = WM1
 ==>
  say -> AC1
  stop -> AC2
  stop -> G1
}
```

They do the rest of the counting: the iterate operator iterates as long as the final number has not been reached (`V2 <> WM1`), and the third operator terminates the count. By putting `stop` in `G1` we signal the simulation that we have reached the end.

The next definition in the file defines facts that will be put into declarative memory. A fact definition consists of lists of values enclosed in parentheses. Each of these lists is translated into a chunk

```
define facts {
(cf1 count-fact  one  two)
(cf2 count-fact  two  three)
(cf3  count-fact  three  four)
(cf4  count-fact  four  five)
(cf5  count-fact  five  six)
…
}
```

The first item in the list will become the name of the new chunk, while the remaining items will be put into slots. For example, `(cf1 count-fact one two)` will be translated into the following chunk:

```
cf1
  isa  fact
  slot1  count-fact
  slot2  one
  slot3  two
```

Although we have specified in the model that it should "say" things, we haven't really specified what it means to say something. In this case we do not really care, but we do want the action to take a certain amount of time. The following declaration takes care of this:

```
define action say {
  latency: 0.3
  noise: 0.1
  distribution: uniform
  output: Saying
}
```

It specifies that the latency to say something is 0.3 ± 0.1 seconds, with a uniform distribution of the noise. The action will show up in the trace as "Saying".

The final part is a script that runs the task.

```
define script {
    digits = ["one","two","three","four","five","six","seven",
        "eight","nine","ten"]
    start = random(3)
    end = start + 1 + random(3)
    print("Counting from",digits[start],"to",digits[end])
    screen(digits[start],digits[end])
    run-until-action("say","stop")
    issue-reward()
    trial-end()
}
```

We will discuss in more detail how to build scripts, but the syntax is similar to C-style languages, so should speak for itself (to some extent). The script here first defines an array of the numbers one through ten, then picks a random start number and end number. The screen function puts the selected numbers in the V-buffer (so `digits[start]` ends up in `V1` and `digits[end]` in `V2`). It then tells the model to run until it the action "say stop" is carried out. It then issues a reward to the model, and ends the trial.

The PRIMs execution cycle

Operators only move around information within the workspace. It is therefore up to the different modules to carry out actions that result from this. The execution cycle is therefore as follows:

1. Retrieve the operator with the highest activation
2. Check the conditions of that operator. If a condition is not satisfied, retrieve the operator with the next highest activation and try again.
3. Carry out the actions of the operator (i.e., move information between slots in the workspace).
4. Based on the new state of the workspace, let all the modules perform their function in parallel. This can be: retrieval from memory, consolidation of the item working memory, and/or carry out an action. The result of this may change the content of the buffers in the workspace again: a declarative retrieval can be placed in the retrieval buffer, and some actions may result in a change in input.

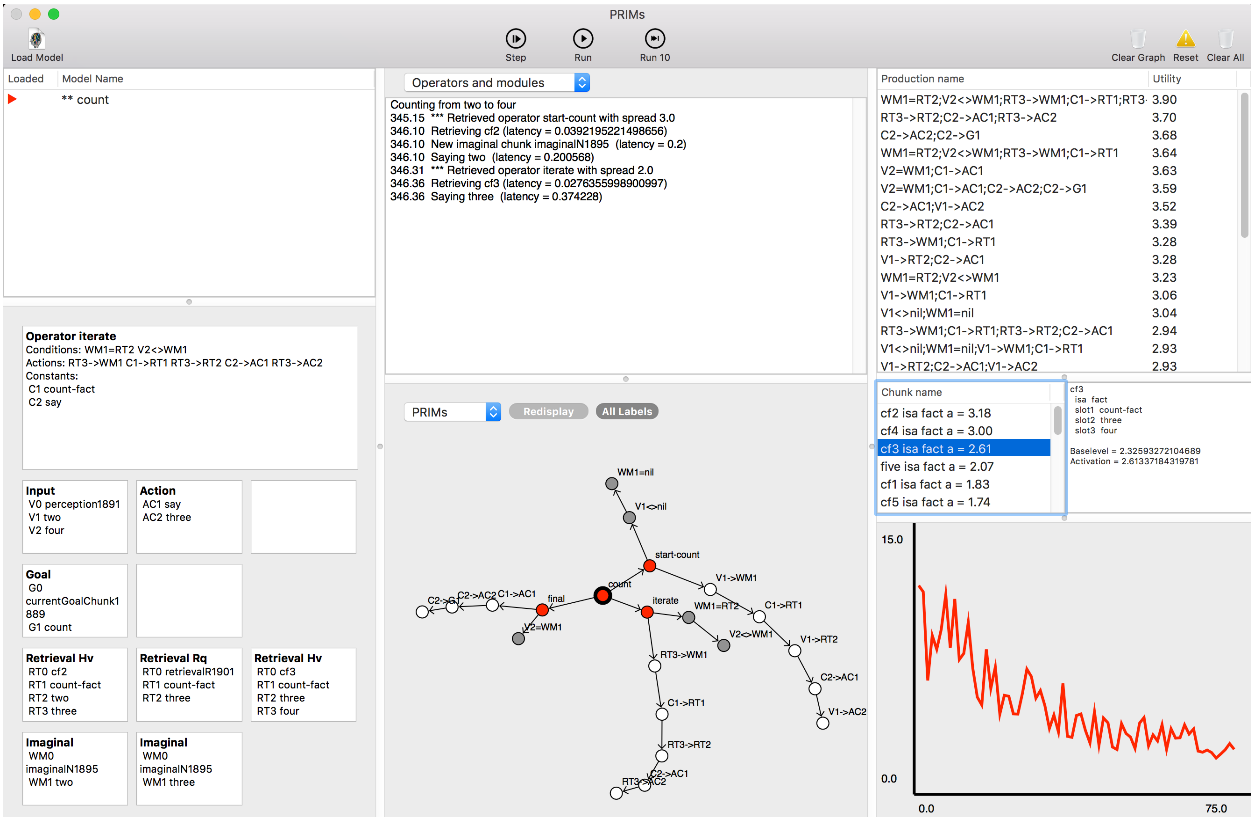
Running a model

To run the count model, start the PRIMs application, and use the "Load Model" button or "Open..." menu option to load the file. Select `count.prims`, after which the top-middle panel will show some information about what is loaded. If there is an error in the model, it will show up in that panel, you will find it in that panel (unless the program crashed first.... Not everything is fool-proof yet).

Once the model has loaded, you can run it with the three buttons in the menu bar (Step, Run and Run 10). Various other parts of the window can help you make sense of what is going on in the model. Let us have a look after the model has run a couple of times (Figure 1).

This model window has the following panels:

- The top-left panel shows a list of tasks that the system currently knows, and are proceeded by a colored triangle if the associated model is actually loaded. The color of the triangle also identifies the task in some of the other panels. The task with ** is the currently active task, clicking a different task activates it.
- The bottom-left panel shows the contents of the different buffers when the model is running. Shown are the currently selected operator, the contents of several buffers before



the operator is executed (left column), how buffers are changed by the operator (middle column), and how buffers are changed by module actions (right column).

- The top-center panel typically shows the model trace. We will discuss this in a little more detail in a moment. The pop-menu at the top of this panel can be used to select the desired amount of detail in the trace. For now, select “Operators and productions”.
- The bottom-center panel shows a graph of the PRIMs. If multiple tasks are loaded, operators will show different colors, and PRIMs that are used by multiple tasks will receive a yellow halo. The popup menu can be used to make graphs of different aspects of the models.
- The top-right panel shows learned production rules with their utilities. Production rules are always combinations of PRIMs.
- The middle-right panel shows all the chunks in declarative memory with their activation. If you click on one, it will show in more detail on the right.
- Finally, the bottom-right panel shows a graph of model results. In the example, the model has run about 50 times, and has improved its speed considerably. Multiple graphs can show after running multiple models.

- The buttons on the top-right of the window do the following: Clear Graph clears the graph on the bottom-right. Reset removes all models from memory and reloads the one that is selected on the top-left (Note: it doesn't actually reload it from the file, just from a stored representation, so if you have changed the model you should load it back explicitly). Clear All, finally, removes everything.

Once you have loaded the model, you can step through it with the Step button, run it all the way with the Run button, or run it 10 times with the Run 10 button (no trace will show in that case). The trace will detail what is going on. You can select several levels of detail with the popup menu. Let the Count run model once, and select "All" in the popup-menu. The trace will show something like:

```
Counting from two to four
0.00  *** Retrieved operator start-count with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.35  Firing V1->WM1
0.65  Firing C1->RT1
0.95  Firing V1->RT2
1.25  Firing C2->AC1
1.55  Firing V1->AC2
1.85  Retrieving cf2 (latency = 0.0457923987659046)
1.85  New imaginal chunk imaginalN6  (latency = 0.2)
1.85  Saying two  (latency = 0.363782)
2.22  *** Retrieved operator iterate with spread 2.0
2.22  Firing WM1=RT2
2.27  Firing V2<>WM1
2.57  Firing RT3->WM1
2.87  Firing C1->RT1
3.17  Firing RT3->RT2
3.47  Firing C2->AC1
3.77  Firing RT3->AC2
4.07  Retrieving cf3 (latency = 0.0868078620404207)
4.07  Saying three  (latency = 0.339786)
       4.42  *** Retrieved operator start-count with spread 2.0
       4.42  Firing V1<>nil
       4.47  Firing WM1=nil
       4.77  Operator start-count15 failed
4.78  *** Retrieved operator iterate with spread 1.5
4.78  Firing WM1=RT2
4.83  Firing V2<>WM1
5.13  Firing RT3->WM1
5.43  Firing C1->RT1
5.73  Firing RT3->RT2
6.03  Firing C2->AC1
6.33  Firing RT3->AC2
6.63  Retrieving cf4 (latency = 0.0693551797048966)
6.63  Saying four  (latency = 0.230098)
       6.88  *** Retrieved operator start-count with spread 2.0
       6.88  Firing V1<>nil
       6.93  Firing WM1=nil
       7.23  Operator start-count23 failed
7.23  *** Retrieved operator final with spread 2.0
```

```
7.23  Firing V2=WM1
7.28  Firing C1->AC1
7.58  Firing C2->AC2
7.88  Firing C2->G1
8.18  Saying stop  (latency = 0.380044)
```

The number in the left column represents time. We can see that at time `0`, the model retrieves the `start-count` operator. It will then start carrying out that operator by firing a production rule for each individual PRIM. After the last PRIM has been carried out, modules do their actions in parallel. In this case, three modules become active: the retrieval module retrieves `cf2`, the action module says two, and the imaginal (working memory) module makes a new chunk to store the count. They each have their own latency, but the longest counts (in this case 0.36 seconds for saying two). Note that anything in the trace that is indented belongs to an operator that failed, so it has no impact on execution, except for taking up time. As we can see, the model is pretty slow to count from two to four, taking about eight seconds.

The popup menu at the top of the panel can be used to change the level of detail. For example, selecting All will reveal:

```
Counting from two to four
0.00  Conflict Set
0.00    start-count A = 5.6670079060538
0.00    iterate A = 5.52215155396268
0.00    final A = 4.90347259519914
0.00  *** Retrieved operator start-count with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.05  Compiling V1<>nil;WM1=nil
0.35  Firing V1->WM1
0.35  Compiling WM1=nil;V1->WM1
0.65  Firing C1->RT1
0.65  Compiling V1->WM1;C1->RT1
0.95  Firing V1->RT2
0.95  Compiling C1->RT1;V1->RT2
1.25  Firing C2->AC1
1.25  Compiling V1->RT2;C2->AC1
1.55  Firing V1->AC2
1.55  Compiling C2->AC1;V1->AC2
1.85  Retrieving cf2 (latency = 0.0457923987659046)
1.85  New imaginal chunk imaginalN6  (latency = 0.2)
1.85  Saying two  (latency = 0.363782)
```

For each retrieved operator it will show all the competing operators with their activations. Of those competing operators, only those will be tried in which the first production that checks conditions matches. Initially, the only productions are productions that check or execute a single PRIM. But the learning mechanism of production compilation will learn productions that execute combinations of PRIMs.

Production Compilation

The trace also shows production compilation in action. After `V1<>nil` and `WM1=nil` have been checked, a rule is learned that makes both comparisons in one step. This rule will not be used right away. The reason is that all rules have a utility value, and the rule with the highest utility value will be used. Initially, there are only rules that each carry out a single PRIM, and they all have a utility of `2.0` (you can give this a different value with the `primU:` parameter). Newly learned productions receive a utility of `0.0` (you can change that as well with the `nu:` parameter). Whenever PRIMs needs to select a production rule, it will add logistic noise to the utility values (varied by the `egs:` parameter, by default `0.05`), and will pick the rule with the highest value.

Whenever a rule successfully leads to the execution of an operator, its utility will be adjusted according to the following equation:

$$U_{t+1} = U_t + \alpha(\text{Payoff} - U_t)$$

Payoff is equal to the Reward parameter (`procedural-reward:`, defaults to `4.0`) minus the time needed after executing the production to finish the operator. The α parameter controls the learning speed (`alpha:` by default `0.1`). Learning is only used for rules that consist of multiple PRIMs: the utility of rules that carry out a single PRIM remains fixed at `2.0` (or whatever value you select for `primU`).

Obviously, a new rule has little chance to compete with the existing rule that it was learned from. Therefore, every time a production is relearned, its utility is increased by the following equation:

$$U_{t+1} = U_t + \alpha(U_{\text{parent}} - U_t)$$

This is the same equation as the regular equation, except that the payoff is replaced by the utility of the parent. This ensure that utility gradually approaches the utility of the parent, increasing the odds that it will eventually be selected (and evaluated).

Production compilation speeds up the execution of the model. Carrying out an operator takes the amount of time needed to retrieve the operator (typically this is very short) and the time needed to fire production rules that carry out the operator. The first of these takes 50 ms to carry out, but each subsequent one takes 300 ms (you can change these values with the `dat:` and `production-prim-latency:` parameters, respectively). After sufficient learning, a production is learned that checks all conditions and carries out all actions with a single rule. Try running the count model often enough to see this (newly learned productions show up in the top-right of the window).

Modeling Transfer

One of the goals of PRIMs is to model transfer. The general idea of transfer is that productions that are learned for one task can be reused for another task. We therefore need to specify at least one additional model. The example is the Semantic model from unit 1. The goal of the semantic model is to judge relationships between animals and animal categories, and answer questions like "Is a lion an animal"? The facts the model uses are:

```
define facts {
    (sem1  property  lion  mammal)
    (sem2  property  mammal  animal)
    (sem3  property  animal living-thing)
    (sem4  property  plant living-thing)
    (sem5  property  tulip plant)
    (sem6  property  bird animal)
    (sem7  property  tweety bird)
    (sem8  property  robin bird)
}
```

Answering the question involves two steps: first to retrieve that a lion is a mammal, and then that a mammal is an animal. Even though this model is semantically different from count, it shares the same type of iteration. The operators in the model are therefore similar:

```
define goal semantic {
    operator start-semantic {
    "Start semantic reasoning"
        V1 <> nil
        WM1 = nil
       ==>
        V1 -> WM1
        property -> RT1
        V1 -> RT2
        subvocalize -> AC1
        V1 -> AC2
    }

    operator move-up-tree {
    "Move up the tree"
        RT2 = WM1
        V2 <> WM1
       ==>
        RT3 -> WM1
        property -> RT1
        RT3 -> RT2
        subvocalize -> AC1
        RT3 -> AC2
    }

    operator say-yes {
    "Say yes when found"
        V2 = WM1
       ==>
        say -> AC1
        yes -> AC2
        stop -> G1
```

```
    }

    operator say-no {
    "Say no on retrieval failure"
        V2 <> WM1
        RT1 = error
       ==>
        say -> AC1
        no -> AC2
        stop -> G1
    }
}
```

The first two operators in this model are the same as in the counting model, with the exception that the slot labels are different. But once the operators are translated into declarative memory, the conditions and actions are identical. The last two operators are different. If a match between the target category and the retrieved category is found, the model should answer "yes", which is slightly different from finalizing the count. Also, if the proposition does not hold, the model will hit a retrieval error at some point. On a retrieval error, the first slot of the retrieval buffer will be set to error (which is, in the example, matched by `RT1 = error`).

Here is an example of a run of the model:

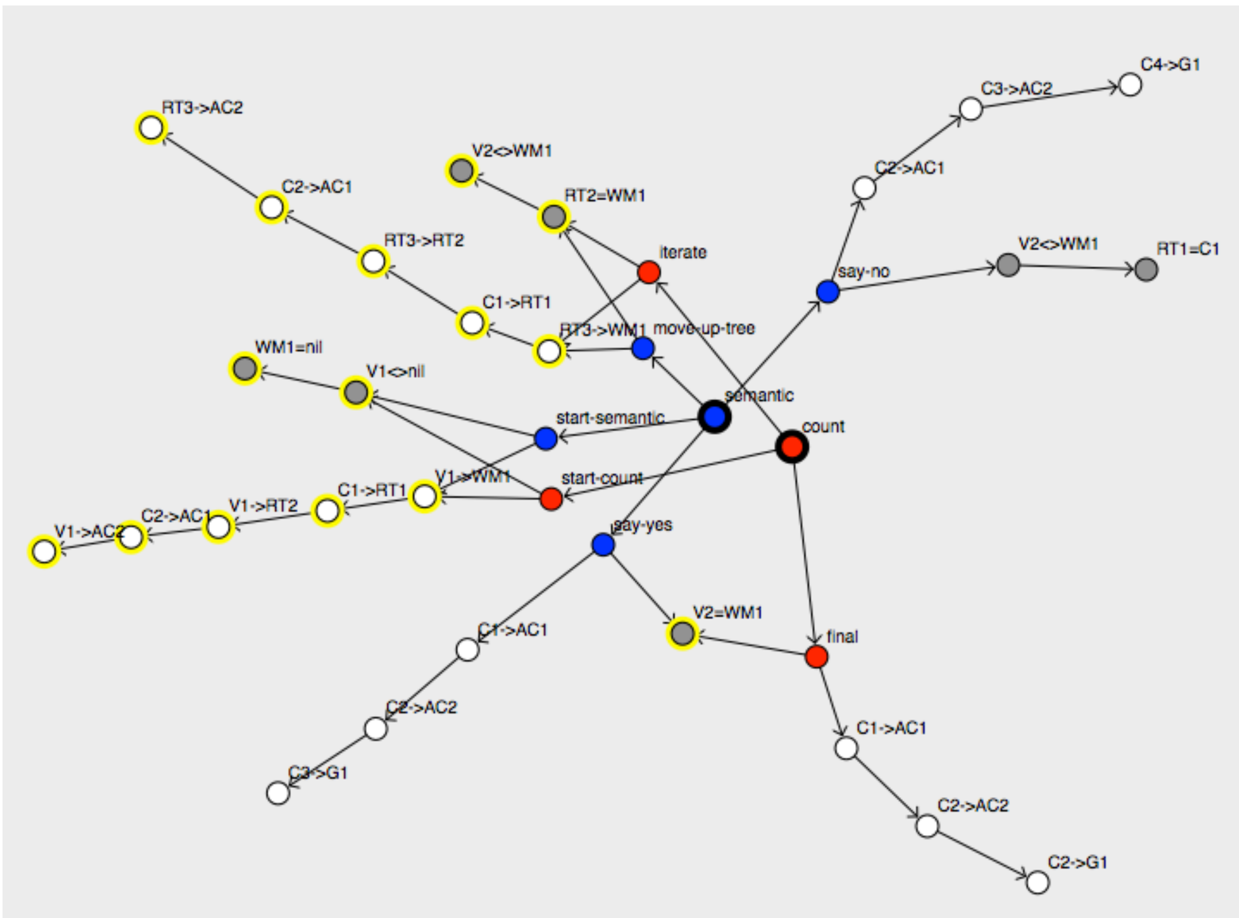
```
0.00  *** Retrieved operator start-semantic with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.35  Firing V1->WM1
0.65  Firing C1->RT1
0.95  Firing V1->RT2
1.25  Firing C2->AC1
1.55  Firing V1->AC2
1.85  Retrieving sem1 (latency = 0.0458280985167914)
1.85  New imaginal chunk imaginalN6  (latency = 0.2)
1.85  Subvocalizing lion (latency = 0.3)
2.15  *** Retrieved operator move-up-tree with spread 2.0
2.15  Firing RT2=WM1
2.20  Firing V2<>WM1
2.50  Firing RT3->WM1
2.80  Firing C1->RT1
3.10  Firing RT3->RT2
3.40  Firing C2->AC1
3.70  Firing RT3->AC2
4.00  Retrieving sem2 (latency = 0.0750171372159093)
4.00  Subvocalizing mammal (latency = 0.3)
      4.31  *** Retrieved operator start-semantic with spread 2.0
      4.31  Firing V1<>nil
      4.36  Firing WM1=nil
      4.66  Operator start-semantic16 failed
      4.67  *** Retrieved operator say-no with spread 2.0
      4.67  Firing V2<>WM1
      4.72  Firing RT1=C1
      5.02  Operator say-no19 failed
```

```
5.03  *** Retrieved operator move-up-tree with spread 2.0
5.03  Firing RT2=WM1
5.08  Firing V2<>WM1
5.38  Firing RT3->WM1
5.68  Firing C1->RT1
5.98  Firing RT3->RT2
6.28  Firing C2->AC1
6.58  Firing RT3->AC2
6.88  Retrieving sem3 (latency = 0.0460701809949314)
6.88  Subvocalizing animal (latency = 0.3)
      7.19  *** Retrieved operator start-semantic with spread 2.0
      7.19  Firing V1<>nil
      7.24  Firing WM1=nil
      7.54  Operator start-semantic28 failed
      7.55  *** Retrieved operator say-no with spread 2.0
      7.55  Firing V2<>WM1
      7.60  Firing RT1=C1
      7.90  Operator say-no30 failed
7.90  *** Retrieved operator move-up-tree with spread 2.0
7.90  Firing RT2=WM1
7.95  Firing V2<>WM1
8.25  Firing RT3->WM1
8.55  Firing C1->RT1
8.85  Firing RT3->RT2
9.15  Firing C2->AC1
9.45  Firing RT3->AC2
9.75  Retrieval failure
9.75  Subvocalizing living-thing (latency = 0.3)
11.24  *** Retrieved operator say-yes with spread 2.0
11.24  Firing V2=WM1
11.29  Firing C1->AC1
11.59  Firing C2->AC2
11.89  Firing C3->G1
12.19  Saying yes (latency = 0.3)
```

However, if you first run the count model for a while, productions have been compiled that can be reused:

```
1108.01  *** Retrieved operator start-semantic with spread 3.0
1108.01  Firing V1<>nil;WM1=nil;V1->WM1;C1->RT1
1108.06  Firing V1->RT2;C2->AC1;V1->AC2
1108.36  Retrieving sem2 (latency = 0.0557285933220927)
1108.36  New imaginal chunk imaginalN12151  (latency = 0.2)
1108.36  Subvocalizing mammal (latency = 0.3)
      1108.67  *** Retrieved operator say-no with spread 2.0
      1108.67  Firing V2<>WM1
      1108.72  Firing RT1=C1
      1109.02  Operator say-no12156 failed
1109.03  *** Retrieved operator move-up-tree with spread 2.0
1109.03  Firing RT2=WM1;V2<>WM1;RT3->WM1;C1->RT1;RT3->RT2;C2->AC1;RT3->AC2
1109.08  Retrieving sem3 (latency = 0.0922408709785375)
1109.08  Subvocalizing animal (latency = 0.3)
      1109.39  *** Retrieved operator say-no with spread 2.0
      1109.39  Firing V2<>WM1
      1109.44  Firing RT1=C1
      1109.74  Operator say-no12166 failed
1109.75  *** Retrieved operator move-up-tree with spread 2.0
```

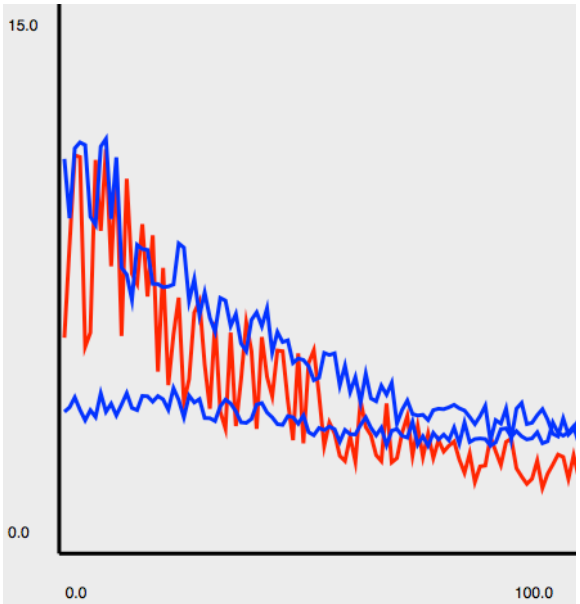
```
1109.75  Firing RT2=WM1;V2<>WM1;RT3->WM1;C1->RT1;RT3->RT2;C2->AC1;RT3->AC2
1109.80  Retrieval failure
1109.80  Subvocalizing living-thing (latency = 0.3)
1111.29  *** Retrieved operator say-yes with spread 2.0
1111.29  Firing V2=WM1
1111.34  Firing C1->AC1
1111.64  Firing C2->AC2
1111.94  Firing C3->G1
1112.24  Saying yes (latency = 0.3)
```



How do we assess transfer between these two models? A first option is to look at the overlap of chunks in declarative memory between the two models. The figure on the previous page gives an impression of this overlap (all the yellow halos), which is considerable.

To get a real sense of the amount of transfer, we have to run the model. We first run the Count model a number of times, and then the Semantic model. Then we run the Semantic model without prior training.

To do this in a simple way, do the following. Load in both the Count and Semantic model. Click on the count model in the top-left panel to activate it. Click the “Run 10” button ten times to run the model for 100 times. You will see the learning curve in the right bottom corner. Now activate semantic, and run it 100 times. The new curve will show semantic after learning count. Now push the Reset button, and run the semantic model again for 100 times. Your graph should look something like the one here.



The top blue curve is semantic without transfer from count, and the bottom curve is the one with transfer.

We can also collect this type of data by automatizing this. Choose the "Run Batch…" option from the Run menu, and select the "testbatch.bprims" as input in the file dialog. Then choose a filename of your choice as output. The system will now run what we just did by hand ten times. It will also generate a datafile that we can analyze with R, Excel or any other packages that takes data tables.

Perception and Action

PRIMs uses a simplified version ACT-R's perception and motor modules. This means that some of the precision is lost, but on the other hand that it is easier to program the experiment part of the model.

For the current assignment, we are going to build two models of addition by counting. The first one will be done through mental operations, but the second model assumes that one of the two counters that has to be maintained will use fingers. The assumption of the model is that each time we say a number, we also stick up an additional finger, and the total number of fingers is available to perception. In order to do this we need a somewhat more elaborate script that changes perception on the basis of actions:

```
define script {
      digits =
["zero","one","two","three","four","five","six","seven","eight","nine","ten"]
    num1 = random(4) + 1
    num2 = random(4) + 1
    print("Adding",digits[num1],"and",digits[num2])
    screen(digits[num1],digits[num2])
    done = 0
    fingers = 0
    while (!done) {
             run-until-action()
```

```
            ac = last-action()
            if (ac[0] == "say") {
                  fingers = fingers + 1
                  screen(digits[num1],digits[num2],digits[fingers])
            }
            if (ac[0] == "answer") {
                  done = 1
            }
      }
    issue-reward()
    trial-end()
}
```

This script picks two random numbers between one and four, and displays them to the model. It will start a loop that ends when the model uses "answer" as an action. More in particular, the `run-until-action()` function will run the model until it takes any action. The `last-action()` function then holds an array with the last action the model took. If the first element in this array is "say", it means we need to put up an extra finger, and add this to the input (it goes into `V3`). If the first element is "answer", it means we are done (the script language has no Booleans, but uses `0` as false and any non-zero number as true).

Assignment

The assignment is to add a model of addition by counting to the existing two models, and to assess transfer between counting and addition.

There are two possible solutions to explore. The first involves keeping track of two numbers in WM. The two numbers to be added are perceptual input, and two `WM` slots are needed to hold the current count and the current sum.

Your first operator has to initialize the process by putting the first addend (which is in `V1`) into `WM1`, and zero in `WM2`. It then has to make a retrieval request for the a count-fact about the number in `V1`.

The second operator harvests a retrieval count-fact that matches `WM1`, updates the number in `WM1`, and issues a count-fact retrieval request for `WM2`.

A third operator harvest a count-fact retrieval that matches `WM2`, checks whether `WM2` is not equal to the second addend (in `V2`), and issues a count-fact retrieval for a count-fact about `WM1`.

A fourth operator checks whether the count in `WM2` equals `V2`, and gives the contents of `WM1` as an answer.

Implement this model, and check whether it works. The script is a more simple version of the one above: replace `done = 0, finger = 0` and the `while` loop by `run-until-action("answer")`. Make sure the last action of your model is "answer"! Although we haven't yet gone over the details of the script (they are in the appendix to this unit), it should be relatively easy to understand and edit.

Once you have made sure the model works, you can check how fast it learns. You can then also see how much transfer there is from the count model, using the same method as with semantic. If you want to use the method with the batch file, you should edit the `.bprims` file so that it uses your new model instead of semantic (hopefully the structure of the batch file speaks for itself).

You will probably conclude that transfer between counting and this addition model is limited. You can therefore try to build an alternative model using fingers. This model only requires three operators, and should show more transfer than the "standard" model.

Details of Declarative Memory

The standard ACT-R equation for calculating the activation of a chunk is as follows:

$$A_i = B_i + \sum_k^{\text{sources}} \sum_j^{\text{slots}} S_{ji} W_k + \text{noise}$$

In this equation B_i is the base-level activation that reflects how often the chunk has been recreated in the past. The double summation calculated the spreading activation from the different buffers (sources) in the workspace. Each of these buffers can spread activation, the amount of which is represented by the W_k parameter that is set as a global parameter (e.g., `:imaginal-activation` for the imaginal buffer, see the list of PRIMs parameters).

Each buffer has a number of slots, and the chunk in each slot is the potential source of spreading, assuming it has a non-zero strength of association (S_{ji}). This strength of association can be set explicitly in the model, but also receives a value by default whenever a chunk in the buffer is in one of the slots of a chunk in declarative memory. For example, if the chunk `three` has a positive association with the chunk that represents `3 + 4 = 7`, because the `three` chunk is in one of the slots of `3 + 4 = 7`.

The base-level activation is calculated using the standard base-level activation formula:

$$B_i(t) = \ln \sum_k (t - t_k)^{-d}$$

In this equation, the t_k's are all the moments in time that the chunk was created or accessed, and d is a decay parameter that is 0.5 by default. In standard ACT-R, both a recreating of a chunk or a retrieval from memory reinforces the chunk. In PRIMs, only recreating a chunk in the working memory buffer reinforces it. However, if the `retrieval-reinforces:` parameter is set to `t,` retrieval also reinforces the activation.

Because the standard formula is not very efficient computationally, an optimized version can be used (and is in fact the default, controlled by the `ol:` parameter):

$$B_i(t) = \ln(\frac{n}{1-d}) - d \ln(t - t_0)$$

In this version of the equation, t_0 is the creation time of the chunk, and n is the number of accesses it has had during its lifetime.

As mentioned above, the strength of association (S_{ji}) can be set by hand in the model. It is zero by default, except when chunk j is in a slot of chunk i. In that case the value is calculated by the following equation:

$$S_{ji} = S - \ln(fan_j)$$

S is the maximum associative strength parameter (`:mas`), and *fan*$_j$ is the number of chunks in which chunk *j* appears. The idea behind this is that as chunk *j* is less unique for chunk *i* (because it appears in many other chunks), its association is weaker.

Whenever the total activation of a chunk drops below the retrieval threshold τ, retrieval fails. If retrieval succeeds, the time required for a retrieval is:

$$t_{\text{retrieval}} = Fe^{-A_i}$$

F is a global parameter (`:lf`). In the case of a retrieval failure, the time spent on that failure is:

$$t_{\text{retrieval}} = Fe^{-\tau}$$

PRIMs adds a few things to the standard activation equations. First, it handles the spreading activation of goals slightly different from standard ACT-R. We will discuss this in a later unit, but the equation is:

$$A_i = B_i + \sum_k^{\text{buffers}} \sum_j^{\text{slots}} S_{ji} W_k + \sum_k^{\text{goals}} S_{ki} A_k + \text{noise}$$

The basic idea is that every active goal spreads an amount of activation that is equal to its own activation.

Furthermore, PRIMs adds the convenience of a “default activation”. In most models you add facts that you assume are more or less permanently available to the model. In the count model you don’t want the model to forget the count-facts. The default activation puts a lower bound on the activation of a chunk. If that activation is well over the retrieval threshold, and the noise in your model is not too large, you can be sure that the chunk will always be retrieved successfully. Here is how this is integrated into the equations:

$$B_i(t) = \ln(e^{\text{fixed}} + \sum_k (t - t_k)^{-d})$$

$$B_i(t) = \ln(e^{\text{fixed}} + \frac{n}{1-d}) - d\ln(t - t_0)$$

PRIMs parameters

imaginal-delay:

Time it takes to put a new chunk in the imaginal buffer (default 0.2)

egs:

Utility noise (default 0.05)

alpha:

Learning parameter for production compilation (default 0.1)

procedural-reward:

Reward for procedural learning (default 4.0)

nu:

Utility for newly compiled productions (default 0.0)

primU:

Utility of productions that handle a single PRIM (default 2.0)

dat:

Default time to fire the first production to handle an operator (default 0.05)

production-prim-latency:

Default time to fire subsequent productions to handle an operator. The idea is that this takes longer because it also involves retrieving a chunk from memory. The current implementation doesn't actually do this, but the time this would take should be accounted for. The consequence is that a fully proceduralized operator only takes de default action time (dat:) to carry out, but if multiple productions have to fire it takes substantially longer. (default 0.3)

bll:

Base-level decay (d) (default 0.5)

ol:

Optimized learning (default t). Set to nil for the standard equation.

mas:

Maximum associative strength (default 3.0)

rt:

Retrieval Threshold (default -2.0)

lf:

Latency Factor (default 0.2)

mp:

Mismatch Penalty (default 5.0).

pm:

Use partial matching to retrieve facts. Off by default (nil), switch to t to use.

ans:

Activation noise (default 0.2)

ga:

Spreading activation (`W`) from the goal (default 1.0)

input-activation:

Spreading activation (`W`) from the input (default 0.0)

retrieval-activation:

Spreading activation (`W`) from the retrieval buffer (default 0.0)

imaginal-activation:

Spreading activation (`W`) from working memory (default 0.0)

default-activation:

In most models, the chunks you specify as part of the model can be assumed to exist for a while, so they probably have reasonable stable activation values. When you give a value to default-activation, that value becomes the lower-bound of baselevel activation for all chunks you specify in the model (including operators). There is no default for this parameter, because when you omit it there is no lower-bound, and any chunk you specify will have a single reference.

default-operator-assoc:

S_{ji} between an operator and the goal it is defined in. This ensures that operators relevant to one of the goals in the goal buffer are more likely to be retrieved instead of other operators. (default 4.0)

goal-chunk-spreads:

Normally, the amount of spreading from the goal is equal to the ga parameter divided by the number of chunks is the goal. If you set this parameter to t, the amount of spreading will be equal to the activation of the goal, allowing you to give goals more or less priority. (default nil) default-inter-operator-assoc:

S_{ji} between an operator and other operators for the same goal. Make it more likely that an operator that is associated to the same goal as the previous operator is selected. (default 1.0)

default-operator-self-assoc:

S_{ji} between an operator and itself. Should be negative to make it less likely that an operator will fire repeatedly. (default -1.0)

perception-action-latency:

Default time for any action that is not defined explicitly (default 0.2)

retrieval-reinforces:

In standard ACT-R, each time you retrieve a fact from memory it receives an extra reference, increasing its baselevel activation. In PRIMs this is not the case by default. If you want standard ACT-R behavior, set this parameter to t. (default: nil)

goal-operator-learning:

Experimental mechanism for a task to find its own operators. When set to t (true), the mechanism will be active. What the mechanism does is try to learn associations between the goal in `G1` and operators it retrieves using reinforcement learning. Whenever the model successfully completes a task, all the operators responsible are updated. The next three parameters also need to be specified. (default: nil)

beta:

Learning speed for goal-operator learning (similar to alpha). (default: 0.1)

reward:

Reward used in goal-operator learning. The reward also maximizes the time that the model will try to reach the goal (default is 0.0, which switches it off, so if you want to use it you have to give it a sensible value, i.e., something slightly longer than the time necessary to reach the goal).

explore-exploit:

Goal-operator learning adds extra noise to goal-operator combinations it hasn't tried very often yet. This parameter scales how fast noise on the goal-operator association is reduced with more experience. A higher value corresponds to a longer period of exploration (default 0.0). Still very experimental, so off by default.

Script Syntax

The scripting language has the following types: integers, reals, strings and arrays. Arrays can hold items of different types. Types are all implicit. There is no boolean type, but instead the integer 0 is used for false, and all other integers for true.

A script consists of a sequence of statements. Statements can be an assignment, a while loop, a for loop, and if clause, or a function call.

Assignment

An assignment assigns a value to a variable, or to an element in a arrays. Examples of the former are:

```
a = (10 + 5) * 8
b = [“letter”, 8, 4.5, random(10)]
```

Examples of the latter are:

```
b[2] = 10
b[3] = [1, 1, 2, 3, 5, 8]
```

Arrays always start at index `0`. If an assignment is made to an array element that is out of bounds, the array is automatically expanded. For example, the following loop creates an array with 5 random numbers:

```
a = []
for i in 0 to 4 {
       a[i] = random(10)
}
```

If, While and For

If statements are very similar to most programming languages. The syntax is

```
if (condition) { then-statements } else { else-statements }
```

The else statement is optional, but both parentheses and braces are required. The condition can be a combination of comparisons, using `&&` for and, `||` for or, and `!` for negation.

New variables within an if or else clause are local to that clause.

While statements have the following syntax:

```
while (condition) { statements }
```

Like with the if-clause, parentheses and braces are required, and new variables within the loop are local.

Finally, there are two kinds of for loops, one that iterates over an integer from a starting to an ending value, and one that iterates over an array:

`for` *`index`* `in` *`startindex`* `to` *`endindex`* `{` *`statements`* `}`

Iterates *index* from *startindex* to *endindex* (both inclusive).

`for` *`index-variable`* `in` *`array`* `{` *`statements`* `}`

Iterates over all elements in *array*.

FUNCTION CALLS

The script module has a set of functions calls that can be used to inspect, change and run the model, and several miscellaneous purposes. The functions are all detailed in the next section.

Script types

Each model has a main script that defines what happens when you push the run button. Optionally, a model can have a second script that is run when the model is loaded, an initialization script. This can be useful to add larger amounts of knowledge to declarative memory, or other things that only need to be done once. For example, an alternative means to add facts to declarative memory in the counting model would be to add the following script:

```
define init-script {
    print("Init script: Adding count-facts to memory")
    digits = ["one","two","three","four","five","six","seven",
    "eight","nine","ten"]
    for i in 0 to 8 {
        name = random-string("count-fact")
        add-dm(name, "count-fact", digits[i], digits[i + 1])
        set-activation(name, 1.0)
    }
 }
```

Script Functions

RUNNING THE MODEL

run-step()

Run the model for one operator.

run-until-action(values to be compared to slot values in action)

Run until the model takes a certain action. Only the slots compared in the call are compared, so run-until-action("say") will stop at any "say" action. run-until-action() will run until any action.

run-relative-time(time)
: Run the model for a specified number of seconds.

run-absolute-time(time)
: Run the model until a particular time. The function time() provides the current model time.

run-until-relative-time-or-action(time, values to be compared to slot values in action)
: Runs until either the specified action is taken by the model, or the specified amount of time has elapsed.

run-absolute-time-or-action(time, values to be compared to slot values in action)
: Same as previous, but now until a particular time.

Perception and Action

screen(items)
: Put the specified items in the input buffer. First argument is put into `V1`, second in `V2`, etc. The argument can also be a single chunk in the visicon, we will explain this in a later Unit.

last-action()
: Returns an array with the last action (`AC1` is the first item, `AC2` the second, etc.). If there is no last action, it returns an array with a single empty string (this is useful in any of the run functions that runs until a time or an action).

Run Control of the Model

trial-start()
: Set the start-time of the model back to zero. You only need to call this if you run multiple trials in a single script run.

trial-end()
: Indicates the end of a trial. Sets the model up for the next trial, and writes a line to the output file in the case of batch running. Always include this.

issue-reward(optional number)
: Give the model a reward that it can use to reinforce connections to successful operators. Gives the reward set in the main parameters of the model if no value is specified, or the amount specified in the parameter.

sleep(amount of time in seconds)

Move the model time forward for the specified amount of time without running the model. This mainly affects memory decay.

Modification and inspection of the model

time()

returns the current model time

add-dm(chunk-name, slot values)

Add a chunk to memory in the same way as add-fact does this. The first parameter is the name of the chunk, and the remaining are values to be put in slots slot1..slotn. Use the function random-string to generate chunk-names if necessary.

set-activation(chunk-name, value)

Set the lower-bound default activation of a chunk to value.

set-sji(chunkj, chunki, value)

Set the Sji value between two chunks

sgp(parameter, value)

Set a global parameter to a certain value. Example: `sgp("lf:", 0.3)`

Other commands and functions

print(arguments)

Print the arguments in the trace field

shuffle(array)

Returns an array with the elements of the given array in random order.

length(array)

Returns the number of elements in the array

set-data-file-field(number between 0 and 3, name)

The lines in the datafile have a number of extra fields that you can put values in, in case your run multiple different conditions in the model, or want to put other information in the datafile. There are four columns for this, and you can set the value in the column with this command, e.g., `set-data-file-field(0,"control")`

random-string(optional prefix)

Generate a unique string starting with the optional prefix. If none is given, the prefix is "fact".

UNIT 2

Learning and Discovering Operators, Variable Binding, Far Transfer

Multiple parallel goals

We encounter new tasks every day, many of which we can carry out without instruction or training. The probable reason is that these new tasks are combinations of things that we already know how to do. Up to now, tasks have corresponded to a single goal. However, a task may consist of several goals, possibly parallel or with some imposed control structure. In this unit we will examine the option to have multiple parallel goals.

In the earlier examples, the goal was stored in slot `G1`. However, we can also store goals in subsequent slots `G2` and up. Each of these goals can be associated with its own set of operators. Goals in slots in the goal buffer spread activation to associated operators (the Sji for this association is `default-operator-assoc`). Operators for a particular goal are also associated with each other (with an Sji of `default-inter-operator-assoc`). This increases the likelihood of using an operator for the same goal as long as there are any applicable. Finally, in most models we don't want the same operator to fire repeatedly, so operators are negatively associated with themselves (by `default-operator-self-assoc`). Operators can put new goals in G slots (e.g., `count-goal->G2`), and can also remove them (by putting nil into them, e.g. `nil->G2`).

For implementation purposes it is important to declare which symbols are goals within the model, and which of these are already in the goal buffer. We are already familiar with the `initial-goals:` declaration, although up to now only with a single goal (which is placed in `G1`). However, if we put more goals in the list, they will placed in subsequent slots of the goal buffer. Other goals that might place in G slots later on have to be declared in a `goals:` declaration. For example, in the upcoming Stroop model, the declaration is:

```
initial-goals: (stroop)
goals: (attendcolor default-attend focuscolor)
```

The challenge of long-term learning is to organize skills, to define mechanisms to evaluate knowledge, and to have strategies to discover the best operators for a new task. Although this puzzle is far from solved, we offer some starting points in this Unit.

Variable binding in goals

In the examples we have looked at in Unit 1, any specific values such as "say" and "count-fact" have been part of the operator. This disadvantage of this approach is that this makes operators quite specific, and therefore less suitable for reuse. To make operators more general, we can also bind the specific values at the level of the goal, keeping operators more general.

For example, in the example of Count and Semantic in Unit 1, the conditions and actions of some of the operators were identical, but each still required a separate operator node because their specific values were different. We can generalize this by building a more generic operator, such as:

```
operator start {
     V1 <> nil
     WM1 = nil
     ==>
     V1 -> WM1
     fact-type -> RT1
     V1 -> RT2
     action -> AC1
     V1 -> AC2
}
```

For the count goal, we can the instance `fact-type` with `count-fact`, and `action` with `say`, while in the semantic task we instantiate it with `property` and `subvocalize`.

We can do this instantiation in the script, for example for count:

```
set-goal(“count",["fact-type","count-fact"],["action","say"],["final-action","say"],
["final-response","stop"])
```

and for semantic:

```
set-goal("semantic",["fact-type","property"],["action","sub-vocalize"],["final-
action","say"], ["final-response","yes"],["final-response-negative","no"])
```

In order for the implementation to know what constants require this extra indirection, we have to declare them in the task definition by adding the following line:

```
references: (fact-type action final-action final-response final-response-negative)
```

Evaluating operators

An operator is a chunk in declarative memory, so just like any other chunk it has a base-level activation, and strengths of association with other chunks. Just like any other chunk, these activations correspond to the odds that we need the operator in the current context.

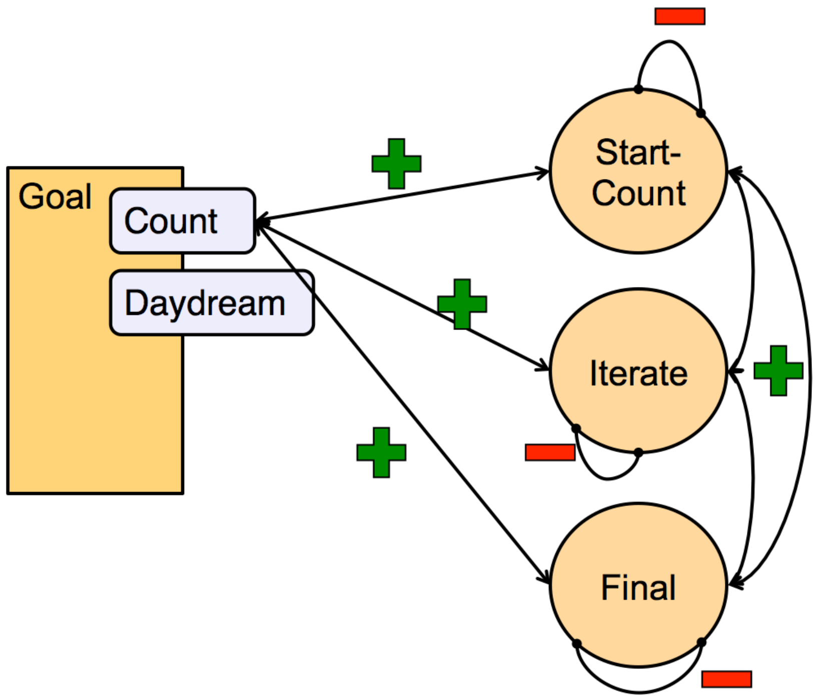
For operators, the base-level activation represents how useful the operator is regardless of the task. In the current implementation, it receives an extra reference if it leads to a successful completion of a goal. To set this up properly, we need to specify a number of things in a task. Here is the parameter declaration of `count-learn.prims`, a model we will use as a demonstration:

```
define task count-learn {
    initial-goals: (learn-count)
    default-activation: 1.0
    ol: t
    rt: -2.0
    lf: 0.2
    default-operator-self-assoc: 0.0
    goal-operator-learning: t
    reward: 10.0
    beta: 0.1
    references: (fact-type action final-action final-response final-response-negative)
}
```

The `goal-operator-learning` parameter switches on the learning mechanisms for operators. Each time an operator is successful in reaching a goal, it will receive an extra reference. Success is defined in terms of whether the goal-action is carried out. The reward parameter sets the maximum time to reach the goal (10 seconds in the example).

Associative strength will be used to represent the usefulness of an operator for a particular goal. The figure on the next page illustrates the overall picture of associations.

The key associations that need to be learned are the associations between goals (like count and daydream), and operators. The assumption is that these associations are initially set to 0, but increase if an operator is successful in accomplishing the goal involved.



Remember that there is no hard connection between operators and goals. So in any situation any operator may be retrieved for any set of goals, but typically only operators will be chosen with a positive association to that goal. However, if a goal is in a situation in which there are no associated operators, any matching operator can be tried. If that leads to success, the association between the operator and the goal will be strengthened using reinforcement learning. The equation for the update is:

$$\Delta S_{ji} = \beta(\text{payoff} - S_{ji})$$

in which

$$\text{payoff} = \text{maxS}_{ji} * (\text{reward} - \text{timeToReward})/\text{reward}$$

If the operator does not lead to a reward, it is penalized:

$$\text{payoff} = \text{maxS}_{ji} * (0 - \text{timeToReward})/\text{reward}$$

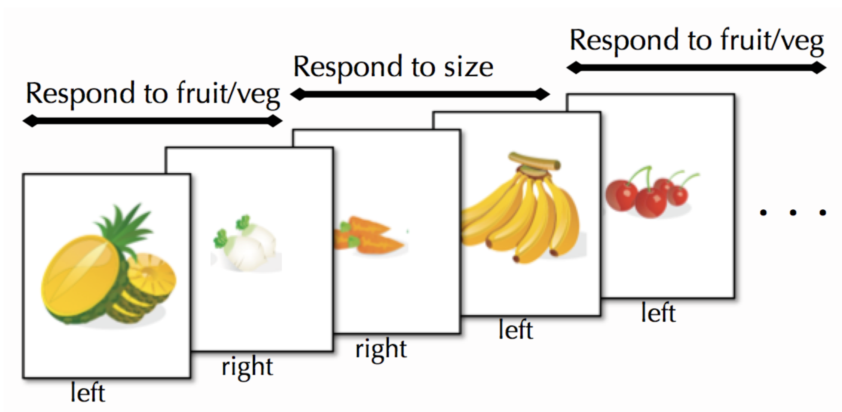
In these equations, reward, beta and MaxSji are parameters that are set by the model (see example above). MaxSji is the `default-operator-assoc` parameter.

If we run the count-learn model, nothing will happen, because there are no operators. But luckily, another model can supply all the necessary operators. So, also load `semantic-global.prims`, run the semantic task a couple of times so that it learns some productions, and now try to run count-learn. It will now run successfully, and will learn associations between the semantic operators and the count goal.

Just semantic-global and count-learn can hardly go wrong. Load in a couple of other models, so that the right operators have some competition from wrong operators.

Far Transfer

Karbach and Kray have shown in an experiment that training on task-switching transfers to several other control tasks, among which the Stroop task. You can load in both models, and check the overlap. The key aspect to modeling transfer here is not so much that there is a large overlap between the two tasks (which there isn't), but that task switching, at least this particular version, trains a particular skill that is useful for picking the better strategy for Stroop.



In the particular version of task switching, subjects have to keep track of what the task is themselves: there is no external cue. The stimuli consist of either pictures of fruit or vegetables, which can be either small or large. For the first two stimuli, subjects have to respond to fruit/vegetable, the next two large/small, the next two fruit/vegetable, etc. The structure of this version of task switching forces a strategy the prepares for the upcoming stimulus (other versions of task switching also allow a more reactive policy, so training on them may not always have the desired result).

In the model this means that after a response has been made on an item, it is not sufficient to wait for the next item: it has to prepare for what to do with that next item. But it should only prepare once.

To prepare, the model will set up a second goal that it will retract when it is done preparing. Setting up the second goal is done with the following operator, which activates when the model is looking at the fixation cross:

```
operator prepare-for-stimulus {
```

```
        V1 = fixation
        G2 = nil
        ==>
        prepare -> G2
    }
```

The “prepare” that is place in `G2` is in fact an indirection: the particular goal that it points to is specified in the main goal:

```
    set-goal("taskswitching",["prepare", "prepare-next"])
```

So, in order to prepare, the “prepare-next” goal is placed in `G2`. This goal now has to decide what the task will be. While waiting during the first fixation cross, the model sets the task to `foodtask` (in `WM1`), the count to `one` (`WM2`), puts nil in `G2` to indicate that it is prepared for the task, and that waits for the stimulus.

```
define goal prepare-next {
    operator set-first-task {
        V1 = fixation
        WM1 = nil
        ==>
        foodtask -> WM1
        one -> WM2
        nil -> G2
        wait -> AC1
    }
```

`wait` is a special action in PRIMs: it will let the model wait until the next perceptual action happens. Although this is convenient we have to keep in mind that in reality our thinking never stops, and people probably fill any mental slack time with other thoughts.

After the task has been carried out (you can check the model yourself to see whether you can figure out how it works), the model needs to check what it has to do next. It therefore uses the `prepare-for-stimulus` operator again that reinstates the `prepare-next` goal. The following operator, which is also part of the `prepare-next` goal, starts determining the next task:

```
    operator determine-next-task-retrieve-count {
        V1 = fixation
        RT1 = nil
        WM1 <> nil
        ==>
        count-fact->RT1
        WM2->RT2
    }
```

Whereas the task switching model has no choice whether to prepare, the Stroop model does have a choice. The idea is that the model can just wait for a stimulus to appear, or that it can prepare by being ready to just focus on the color and ignore the word. If the model just attends the stimulus, both the word and the color will be put in slots in the input buffer, but if the focus is on color-only, only the color will be represented. In the case of a conflict trial and a regular attend action, spreading activation will both increase and decrease activation of the response, while in a congruent trial spreading activation only increases activation. But if the focus is just on color, the difference disappears.

Preparation in the Stroop model is done by the following operator:

```
    operator prepare-for-stimulus {
        V1 = fixation
        G2 = nil
        ==>
        prepare -> G2
    }
```

This the same operator as in the task-switching model, but now "prepare" is instantiated with "focuscolor":

```
set-goal("stroop",["prepare","focuscolor"])
```

While watching the fixation cross, the model sets up a second goal to focus on just the color when the stimulus appears. This goal has just a single operator, `attendjustcolor`, which carries out the `attendcolor` action that will put the color of the ink in `V2`.

```
define goal focuscolor {
    operator attendjustcolor {
        V1 = stim
        V2 = nil
        ==>
        attendcolor -> AC1
    }
}
```

The preparation strategy competes with the more default `just-wait` strategy:

```
    operator just-wait(activation=1.5) {
    "Just wait for the stimulus"
        V1 = fixation
        ==>
        wait -> AC1
    }


    operator attend(activation=1.5) {
        V1 = stim
        V2 = nil
        ==>
        attend -> AC1
    }
```

The `attend` action will attend both the color of the ink (which will appear in `V2`) and the identity of the word (which appears in `V3`). If both attributes are attended, the identity of the word will interfere with the color of the ink, but if only the color of the ink is attended, interference will be absent.

The `(activation=1.5)` addition to more default operators means they have a higher base-level activation than the `prepare` operator, so they would normally win the competition most of the time. However, the `prepare-for-stimulus` operator also appears in the task-switching model. Training on task-switching will therefore increase the activation of that operator, making it more likely that it will be chosen after switching to Stroop.

Try running the Stroop model, and try running it after first running the Task-switching model for a number of trials (± 100 should do it). Observe the different choice of operators.

Running models in batch mode

The PRIMs interface is fine to observe your model, and see some qualitative evidence for learning and transfer, but if you want to fit actual data you need to be able to run the model hundreds of times, and average over the results.

To support this, PRIMs has an option to run a simple external script. The script has the extension .bprims, and contains simple commands to run one or more models multiple times. The script (in `taskswitchstrooptransfer.bprims`) is as follows:

```
repeat 10
reset
run stroop stroopcontrol 50
reset
run taskswitching taskswitching 300
run stroop strooptransfer 50
```

The first line always starts with "repeat" and a number, indicating how often the rest of the script has to be run. A "reset" puts the models back into their starting state. The run command has three arguments. The first is the name of the task. Make sure that the name matches both the filename with them model (e.g., `stroop.prims`), and the name of the task within that file (so the task has to be called `stroop`, and not `my-stroop` or anything else). The second argument is a label that is placed in the output file. This is useful for later processing of the output: in this example you want to distinguish running stroop without prior experience (`stroopcontrol`) from running stroop after taskswitching (`strooptransfer`). The last number is the number of times that you run the model.

To run the script, choose "Run batch…" in the "Run" menu in PRIMs, enter the bprims file name, then give an output file name, and wait until the simulation finishes. You can then analyze the output file (which is just a big table in a text file) with any program you like (we typically use R for this).

You can try this out by running this by hand and observing the choice of strategy (and the resulting latencies), or by running the provided batch script (`taskswitchstrooptransfer.bprims`). A simple R-script is provided to extract the results out of the data file. `TSStroop.R` is an example R-file that analyses the output.

Assignment

In this assignment you have to build a model of multi-column multiplication based on a model of multi-column addition. The multi-column addition model is given in the file `multi-column-addition.prims`, and a starting point for the assignment is in `multi-column-multiplication.prims`. To keep things simple, we ignore the perceptual aspect of the task, and have a script that insures that the model looks at the right place at the right time.

The addition model solves the following problem:

```
551
263 +
---
```

The assumption of the model is that visual focus is first on the rightmost column (so 1 is in `V1` and 3 is in `V2`). The script assumes that you carry out a `write` action to write down the answer under the column (what you write will end up in `V3`), and that you follow that by an `attend-next-column` action. The script then moves attention to the next column with 5 and 6 in it. Once the model has attended all columns, the attend-next-column action results in `done` in `V1`. The script assumes that the model ends with the action `SAY DONE`.

The procedure for multiplication is a bit more complex, because there are multiple ways in which you can do it. The script assumes that you do it in the way I was taught it in elementary school (feel free to modify it):

```
  15                    15
  24                    24
 --- x                 --- x
                        60
                       30
                       --- +
                       360
```

The procedure is as follows: you read the first column (5 x 4), and write down the rightmost digit in the answer (0). Then shift focus in the first row to the next left digit, but with the same digit in the second row (so 1 x 4). After the first row is done, now shift attention in the second row to the next digit, and start a new row in the answer (so now we looking at 5 x 2). After we are done with this part of the procedure, we end up with a multi-column addition, but we fortunately already know how to do that.

The script takes care of focussing on the right digits, and makes sure write actions put digits in the right place. So in a sense we are cheating, because we take away what is perhaps the most complex aspect of multi-column multiplication. What remains is still interesting though.

The given script assumes the following sequence of perception and actions:

At the start, it focusses on the right-most column (5 x 4). It expects a `write` action to write down the answer to that. An `attend-next-column` takes it to the appropriate next column (1 x 4). Once it is finished with the first row (but not the second), it will put `column-done` in `V1`. It then expects you do an `attend-next-column-second-row` action to move shift attention in the second row. Once it is also through the second row, it will put `all-columns-done` in `V1`. This signals the start of the addition part of the solution, and this works just like in the addition model.

In your model you will see three levels of transfer: the reuse of a complete goal, because multi-column-addition is part of multi-column-multiplication, the reuse of complete operators (some things you do in multiplication are the same as with addition, except that you use a different type of fact), and transfer of the type in which sequences of PRIMs overlap (the type we have already seen in Unit 1).