A Quick Introduction to 4CAPS Programming

Scott P. Sanner
sanner@andrew.cmu.edu

Includes material adapted from the
4CAPS Manual by Sashank Varma

Revision 1 – 9/8/99

Overview

This document is intended as an introduction to the 4CAPS programming language for activation-based symbolic modeling. The 4CAPS programming language is a production system based language that organizes productions and working memory elements (wme's) into modules that are intended to model brain activation flow. Each module is assigned an activation capacity that allows one to model the effects that activation constraints have on cognitive processing.

In order to understand how models are built in 4CAPS, it is first important to understand how the underlying production system mechanism works. From that point, one can simply augment the basic production system architecture with activation-based working memory elements, activation-based production firing, and capacity constrained modules to arrive at the general 4CAPS architecture.

Although one does not need to know Common LISP to program in the 4CAPS language, Common LISP objects do form a fundamental basis for many of the constructs used in 4CAPS and thus are useful to know. For example, as will be discussed later, declarative memory chunks in 4CAPS are actually built on top of Common LISP object classes and thus one can define attribute member variables for these objects as well as class methods to perform operations on these objects. Consequently, it is to the 4CAPS programmer's advantage to know at least a little about Common LISP objects to program efficiently in the system. For those already familiar with Object-Oriented Programming paradigms, the syntax definitions and examples provided in this manual should provide enough information for one to become fairly proficient with the Common LISP side of 4CAPS.

Production Systems

There are many sources which argue for the utility of production systems for cognitive modeling (e.g. "Production Systems as Notation"- Newell & Simon, "Unified Theories of Cognition" – Newell, and "The Atomic Components of Thought" - Anderson). Without digressing into a philosophical discussion, it should suffice to say that there exists sufficient evidence to suggest that production systems represent a useful level of abstraction for representing procedural tasks as sets of recognition-action pairs.

Production systems generally deal with two types of memory: declarative and procedural. Whereas declarative memory stores chunks of knowledge (e.g. [star,above:plus]), procedural memory defines what actions should be taken when the state of declarative memory matches some specified condition (e.g. if ([star,above:plus]) then (say TRUE)).

Declarative memory can be broken down into two parts: classes and instances. Every instance of a working memory element (*wme*) must belong to a class. That is, one could have a wme class for a person and their general attributes (e.g. eye-color, hair-color, gender) and another wme class for objects with their general attributes (e.g. size, shape, color). However, a class alone does not specify the existence of any class instances and many different instances of a class could conceivably exist concurrently. Consequently, classes must have the ability to be instantiated with distinct values. Thus, in working memory, one could have one wme instance of class person (e.g. [person, eye-color: blue, hair-color: brown, gender:male]) and two instances of class object (e.g. [object, size:small, shape:round, color:blue] and [object, size:large, shape:square, color:black]). Thus, it is important to distinguish a class instance from a class type which is simply a template for all instances of that class.

Thus, working memory elements in the declarative memory store are instances of classes having the following conceptual form:

$$\textit{wme-class-type} := \{\textit{type}, \textit{relation}_1\!:\!\textit{value}_1, \textit{relation}_2\!:\!\textit{value}_2, \ldots, \textit{relation}_n\!:\!\textit{value}_n\}$$

Here, *type* and $\textit{relation}_n$ are distinct identifiers and $\textit{value}_n$ refers to other wme instances or Common LISP symbols (e.g. t,nil,'success,'failure). Keep in mind that this is a conceptual representation and the mapping from this representation to the 4CAPS syntax will be covered in a later section.

Procedural memory consists solely of productions. A production consists of two components, a left-hand side (LHS) that matches a specific condition and a right-hand side (RHS) that indicates the action to take when that condition is satisfied in declarative memory.

The condition (LHS) of a production must perform two tasks:

1) Define one or more type and instance name bindings that are to be matched by the production – these declarations are the *wme-specifications* for the production. A *wme-specification* simply states a *wme-class-type* and *instance-name* pair such that a variable matching *wme-class-type* will be bound to the identifier given by *instance-name*.
2) Define one or more constraints for the wme attributes that are required for the production condition – these constraints are *wme-tests* for the condition. The requirement for the production condition to be met is that each of the single *wme-tests* itself is met (thus, condition satisfaction is a conjunction of all *wme-tests*). A *wme-test* usually consists of a comparison using a Common LISP operator that allows the comparison of an attribute of one matched instance of a class with either an attribute of another class instance or a Common LISP symbol. There are other possibilities for tests (e.g. absence tests), but these are generally specific to individual production systems and will be covered in the 4CAPS syntax section.

Thus, the condition side of a production rule could be represented in the following conceptual format (where $^+$ indicates that one or more instances are required):

$$\begin{aligned} &\textit{lhs} && := \{\textit{wme-specification}^+, \textit{wme-test}^+\} \\ &\textit{wme-specification} && := \{\textit{wme-class-type},\ \textit{instance-name}\} \\ &\textit{wme-test} && := \{\textit{operator}, \textit{value}_1 | (\textit{attribute}, \textit{instance-name}_1)_1, \\ & && \quad \textit{value}_{21} | (\textit{attribute}_2, \textit{instance-name}_2)\} \end{aligned}$$

The action (RHS) of a production simply states the actions that should be carried out if the LHS of the production found a successful match. The RHS can consist of various operations on the declarative memory store using the wme's that matched the production condition for operations such as addition, deletion, modification, or activation spread (all termed *wme-operators*. Also general Common LISP expressions are also generally allowed on the RHS to deal with any issues such as printing information to the screen or modifying global variables to interact with some component of the system. Thus, the rhs could simply be represented in the following format:

rhs := {{*wme-operator, class-instance,…*} | {*Common-LISP-expression*}} $^+$

Therefore, the overall conceptual syntax for a production is simply the following:

production := {*production-name, lhs* → *rhs*}

This matches our intuitive notion that a production is simply a mapping from a condition (LHS) to an action (RHS).

So, up to this point we have covered the declarative and procedural aspects of production systems. To quickly review this indormation, the declarative store has a set of wme class templates associated with it and a set of instances of those class templates that form the working memory set. The procedural store contains a set of productions as defined above. The system matches instances of wme's to the LHS of each production and for each matching instance, it performs the action specified on the RHS of the production.

The final basic concept to be understood in general production systems is that of production cycles. The production system will generally start out at production cycle 1 where it has some initial set of wme's initialized and placed in working memory by the user. An initial set of productions will match the initial memory state and these productions will fire concurrently, modifying working memory according to the RHS of their productions. Thus, production cycle 2 is now reached where the memory is in a new state due to the modifications of the previously fired productions. These productions now fire, updating working memory and production cycle 3 is now reached with a different state than before. And so the cycle continues with each iteration of production matching (recognition) and production firing (action) constituting one production cycle. This is also sometimes referred to as the *recognition-action* cycle.

4CAPS Note: It is possible for many possible distinct matchings among the working-memory elements and the productions in the system. Whereas some production systems impose a serial order on the firing of productions using a conflict resolution scheme (e.g. OPS5, ACT-R), 4CAPS fires all matching productions since it is a concurrent production system intended to model processes occurring simultaneously in separate parts of the brain. Thus if a production's condition matched a person having blue eyes and multiple instances of a person existed in the database that had the attribute of blue eyes, the production would fire once for each separate matching. It is important to keep in mind the complications that can result from parallel production firing when constructing *models. One possible error that is indeterminate in 4CAPS is the result of concurrently modifying the same wme in the RHS actions of two different productions. This action is undefined and should be avoided.*

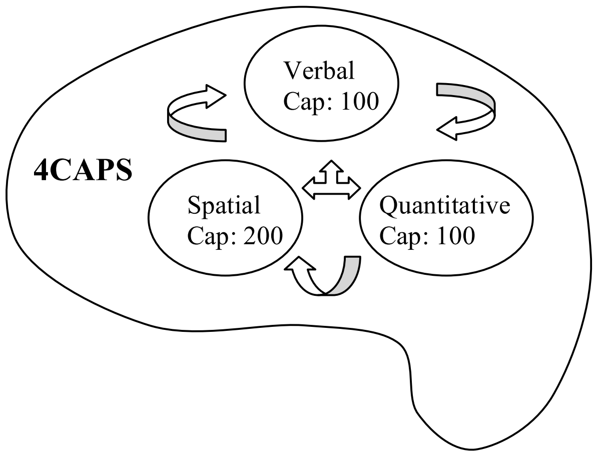
Activation Flow

Without augmentation, a production system is a purely symbolic system that matches wme's on the left-hand side of its productions and performs actions on the right-hand side. However, in terms of cognitive modeling, this limits one to a fairly small representational space. That is, with a purely symbolic production system, one could only model working memory elements as either bring present or not present and productions would fire solely based on these properties.

However, a slightly more useful model involving production systems in cognitive models would be one involving activation. That is, rather than having a simple binary representation for a wme (i.e. present or not present), wme's can be assigned a continuous activation value that changes over time as a result of activation spreading and inhibition in the model. Furthermore, productions can be set to fire only when the activation for a wme has exceeded some threshold therefore allowing conditional processing based on activation. And lastly, if one can separate the model into separate modules, then each module can be assigned an activation capacity and a set of productions and working memory elements where activation would flow within each module and also between modules. The activation capacity would serve to limit processing in the module when the activation demand by its current task exceeded what it could supply. This could lead to a variety of consequences including forgetting, confusion, and impairment of processing.

At this point, a hypothetical example would probably help illuminate how activation constrained modules and a production system can be used to model brain activation flow and in the following case, individual differences as well.

Imagine that one built a 4CAPS system having a spatial module, a verbal module, and a quantitative module. Each module would have its own productions and working memory elements (wme's). The spatial module productions could take care of tasks such as 3D rotation, the verbal module could take care of tasks such as sentence parsing and understanding, and the quantitative module could deal with tasks involving numeric computation. Each module would be assigned an activation limit that constrains the total activation that the wme's in the module can collectively consume (this limit is indicated in the diagram to the right). Once the total demand for activation within a module exceeds its limit, activation is scaled back to



remain within the activation constraints. So, for example, if a module demands 300 units of activation, but its capacity constraint is set at 200, then the actual activation supply to each wme in the module will be scaled back by 2/3.

If the subject had low verbal capabilities (as modeled by a low verbal module activation constraint) then the subject may display difficulties in processing the written text. That is, as they parse each sentence, they produce wme's that semantically represent what they are reading. If these working memory elements make large activation demands, then at some point, the activation in the module will exceed the supply. At this point, the activation of various elements would be scaled back to fit within the activation constraints. Thus, if productions had certain thresholds required for matching and processing wme's, the activation constraints would limit processing by preventing certain wme's from exceeding the target threshold activation required to fire a production. By changing the activation constraints on the module, one could model individual differences in verbal processing. Likewise, by using activation constraints and wme activation thresholds in other modules, one can simulate individual differences in the quantitative and spatial modules as well (or any other module in the system at that).

Even more interesting though, modules are not limited to internal activation spreading. Productions in modules can interact with wme's from any other modules. Any module can spread activation or inhibition to a wme, the cumulative effect of the activation supply (positive or negative) from all modules determines that wme's actual activation. So, for instance, both the verbal module and the spatial module could spread activation to a wme representing an instruction. The total activation that is given to the wme is the sum of the activations as drawn from both modules. However, the activation of a wme from a module's point of view is only the activation that it is supplying to that wme.

The main operation responsible for all of the activation and inhibition spreading is known as *spewing*. Spewing is a process by which activation from one wme in the production system is spread to another wme. Consequently, if the factor for spewing activation is positive, one wme in the system will spread activation to the other and if the factor is negative, one wme will inhibit the activation of another. As mentioned earlier, each module in 4CAPS has its own activation supply and thus the activation provided to a wme is by default supplied by the module of the production that is performing the spewing operation. However, spewing can also draw activation from multiple modules simultaneously. For a more detailed explanation of spewing and the classification of different types of spewing, the 3CAPS manual provides a good introduction to this. However, it should be obvious that spewing can mimic many activation flow processes such as element-to-element activation and inhibition, fan-in (many wme's spewing to one element), fan-out (one wme spewing to many elements) and any combination of the above processes.

There are two main components to analyzing the activation spread in 4CAPS. The first method involves only activation local to a module and is used when testing the left-hand side (LHS) activation of a working memory element (wme) in a production. The second method involves the total activation for the wme from all modules and is used in the right-hand side (RHS) action of spewing. Two examples that exhibit this interaction follow:

Example 1: Spewing Activation

1) *Exec* module with a capacity of 6.0
2) *Arith* module with a capacity of 5.0

The following sample run exhibits the result of spewing from *t* (always at a default activation value of 1.0) to *goal-1* in the first cycle and then spewing from *goal-1* to *goal-2* in the second cycle.

Cycle 1:
Spew *t* → *goal-1* (5.0 from *Exec* module) (5.0 from *Arith* module)

wme	*Exec* Module	*Exec* Act Scaling Factor	*Arith* Module	*Arith* Act Scaling Factor	Total Act
goal-1	5.0	1.0	5.0	1.0	10.0

Cycle 2:
Spew *goal-1* → *goal-2* (0.1 from *Exec* module) (1.0 from *Arith* module)

wme	*Exec* Module	*Exec* Act Scaling Factor	*Arith* Module	*Arith* Act Scaling Factor	Total Act
goal-1	5.0	1.0	5.0	.33	6.67
goal-2	1.0	1.0	10.0	.33	4.33

Example 2: Inhibiting Activation and LHS Testing

Assume the same modules that were used in the previous example. The following sample run exhibits the result of spewing from *t* (always at a default activation value of 1.0) to *goal-1* in the first cycle and then spewing from *goal-1* to *goal-2* in the second cycle (using inhibition in the *Exec* module and activation in the *Arith* module).

Cycle 1:
Spew *t* → *goal-1* (5.0 from *Exec* module) (5.0 from *Arith* module)

wme	*Exec* Module	*Exec* Act Scaling Factor	*Arith* Module	*Arith* Act Scaling Factor	Total Act
goal-1	5.0	1.0	5.0	1.0	10.0

Cycle 2:
Spew *goal-1* → *goal-2* (-0.1 from *Exec* module) (1.0 from *Arith* module)

wme	*Exec* Module	*Exec* Act Scaling Factor	*Arith* Module	*Arith* Act Scaling Factor	Total Act

goal-1	5.0	1.0	5.0	.33	6.67
goal-2	-1.0	1.0	10.0	.33	2.33

When matching a wme, if a threshold is not specified a default of 0.0 is assumed. Consequently, any subsequent production trying to access *goal-2* from the *Exec* module would not be able to access it (because it has negative activation in the *Exec* module), however a production from the *Arith* module would be able to access it (because it has positive activation in the *Arith* module). Therefore spewing activation on the RHS makes use of the total activation, but LHS matching makes use of the activation local to the module.

Example 3: Cumulative Activation

[Example not made yet.]

4CAPS Syntax

A formal definition of the 4CAPS grammar is included at the end of this document. Following are simple examples intended to help one quickly assimilate the general syntactical structure of 4CAPS models.

Modules:

Initially, there is one default module built into the system. To delete this module, use the *del-mods* command:

```
(del-mods)
```

To add new modules to the system and set their capacity, use the commands *add-mod* and *set-cap*. Since one module in the system must be the default module, use the command *set-cmod* for this purpose. In the following example a module named *exec* is added as the default module and its activation capacity is set to 5.0:

```
(add-mod exec)
(set-cmod exec)
(set-cap 5.0)
```

Any commands that do not reference a module (as in *set-cap*) automatically perform the operation on the current default module. In order to force commands to apply to non-default modules, use the @ extension on the command and specify the intended target module as the first argument to the function. Thus, to add an arithmetic model and set its activation capacity, the following commands could be used:

```
(add-mod arith)
(set-cap@ arith 5.0)
```

Working Memory Elements:

WME classes (types) are based on Common LISP objects and can be defined using the *defwmclass* command. In the following example, a class of type *digit* is declared and given two member variables, value which has default value of 0, and op-num which does not have a default specified resulting in a default value of nil.

```
(defwmclass digit ()
  (value 0)
  op-num)
```

[WME class methods can also be added using the *defmethod* command. Details to be added later, these are not required for 4CAPS functionality.]

There are three ways to modify the contents of the declarative memory store. The command *add* simply adds instances of a class to the declarative memory store and can be used outside of productions. The command *del* deletes an instance of a working memory element and should generally be called from the RHS of a production. The command *spew* allows the spread of activation from one memory element to another and is the main command involved in spreading activation/inhibition in 4CAPS. The following examples demonstrate the use of these commands - note that the first example can exist standalone or in a production, but the last two examples should only be called from the RHS of a production:

```
(add (digit :value 1 :op-num 4) 1.0); adds digit wme with act. = 1.0

(del goal-1)                        ; deletes a wme matched as goal-1

(spew goal-1 (digit :value 0 :op-num 4) 1.5) ; spew 1.5 times the act.
                                    ; of goal-1 to the new wme digit;
                                    ; note that this is creating and
                                    ; spewing activation to a new wme

(spew t goal-1 2.0)                 ; spew 2.0 times the act. of t
                                    ; (always at act 1.0) to goal-1

(spew goal-1 goal-2 3.0)            ; spew -3.0 times the act. Of
                                    ; goal-1 to goal-2 (inhibiting)
```

Productions:

As mentioned in an earlier section, a production consists of a production name, left-hand-side (LHS) and right-hand-side (RHS). The LHS defines the conditions for which the RHS will fire. The LHS consists of two main components, the wme specifications for describing which wme types how many of each can match and the tests which define the constraints on the matching. It is important to note that there are two types of tests, positive tests and negative tests. Positive tests are probably the most used and define what constraints on the wme's in the wme specification must be met. The negative test actually has its own wme specification and defines which constraints should not be met by the matched wme's. That is, a positive test would allow any wme to match which had the property of being red, but the negative test would prevent any wme matching if it did have the property of being red. Therefore, the negative test can be used as an absence test.

The following example production is intended to determine when a second digit for an addition problem has been recognized and to build a new goal for performing the arithmetic and finishing the problem. The production's name is *finish-problem* and it matches one wme of type *goal* at activation level greater than 2.5 and binds it to the name *old-g*. (Note that the binding for the type is on the left and the type itself is the second argument, the threshold activation is the third argument and defaults to 0.0 if not specified.) The positive tests specify that the goal that is currently in memory is to get the second digit and that the field of this goal indicates that is has been completed. The negative (*no*) test for this production matches another wme of type *goal* (binding it to ~g) and ensures that no goal exists whose purpose is to perform an addition. If these cases are met, then the RHS of the production spews activation from the current goal *old-g* to a new goal whose purpose is to perform addition. This new goal receives 1.0*(activation of *old-g*) activation contribution from the default module and it receives 0.5*(activation of *old-g*) activation contribution from the arith module.

```
(p finish-problem ((old-g goal 2.5))
  (eq (is old-g) 'get-second-digit)
  (eq (done old-g) t)
  (no ((~g goal))
      (eq (is ~g) 'do-addition))
  -->
  (spew old-g (goal :is 'do-addition) 1.0 (in arith 0.5))
)
```

More examples are exhibited in a full production model for performing arithmetic in a later section of this manual.

4CAPS Run-time Environment

To run 4CAPS, start up the Allegro Common LISP Enterprise IDE application (this will be one of the menu options when Allegro Common LISP 5.0 has been installed). In the *Debug Window* that appears, type the following line:

```
(in-package "CL-USER")
```

The response to this input should be:

```
#<The COMMON-LISP-USER package>
```

Now, from the *File* menu, choose the *Load* option. Select *4caps9.lsp* as the file to load (this is the LISP source for the 4CAPS system). The system should respond with the following message in the *Debug Window* (and no errors should occur):

```
; Loading Z:\shared\4CAPS\4caps9.lsp
```

At this point, 4CAPS is loaded and all you need to do is evaluate and run your models. To load a model into 4CAPS, open it in the Allegro *Editor* window. Once it is ready to run, you must compile the model, so highlight the entire window (or press *ctrl-a*). To compile this selection, press the button resembling **(=>)** on the toolbar. No errors should occur during compilation, if they do, check the syntax of the model. Occasionally warnings occur during compilation, generally the model will run fine in spite of these.

To run the model, go back to the *Debug Window*. In this window, you can type one of the two following *run* commands. The first command runs the production system until no more productions fire and the second command single steps the system, one production cycle at a time (actually you can step any number of production cycles by entering a different number for this command).

```
(run)   ; Run the model to completion
(run 1) ; Single step the model
```

To test out your installation, try running a small model (perhaps one of the example models below). To view the contents of the working memory and the activation values of these wme's at the end of the simulation or while single stepping the model, use one of the following commands:

```
(wm)                ; show the working memory of the default module
(wm@ module-name)   ; where module-name can be any module
```

Finally, for purposes of turning production tracing on and off, the commands *trace-mods* and *untrace-mods* can be used:

```
(trace-mods)   ; Turns on production tracing
(untrace-mods) ; Turns off production tracing
```

[Loading additional modules (e.g. the problem-solver)]
[Monitoring output and activation flow]
[Tracing modules for debugging]

4CAPS Advanced Concepts

[Using class hierarchies for wme's and matching]
[Using embedded module hierarchies]

4CAPS Examples

[More complex problem-solving module example with object inheritance]
[Sample 4CAPS interface code for reading/writing file-based data]

4CAPS Arithmetic Example

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File:   Arithmetic.lsp
;; Author: Sashank Varma
;;
;; Description: A simple 4CAPS arithmetic example using
;; two modules, an executive module for control and an
;; arithmetic module for performing computation.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(del-mods)

(add-mod exec)
(set-cmod exec)
(set-cap 5.0)

(add-mod arith)
(set-cap@ arith 5.0)

(trace-mods)

(defwmclass goal ()
  is
  done)

(defwmclass digit ()
  value
  op-num)

(defwmclass answer ()
  value)

(defwmclass start ()
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Productions in the executive module.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p start-problem ((s start))
   (no ((~g goal))
       (eq (is ~g) 'get-first-digit))
-->
   (spew s (goal :is 'get-first-digit) 1.0
                                       (in arith 0.5))
   (del s)
)

(p continue-problem ((old-g goal))
  (eq (is old-g) 'get-first-digit)
  (eq (done old-g) t)
  (no ((~g goal))
    (eq (is ~g) 'get-second-digit))
  -->
  (spew old-g (goal :is 'get-second-digit) 1.0 (in arith 0.5))
)

(p finish-problem ((old-g goal))
  (eq (is old-g) 'get-second-digit)
  (eq (done old-g) t)
  (no ((~g goal))
      (eq (is ~g) 'do-addition))
  -->
  (spew old-g (goal :is 'do-addition) 1.0 (in arith 0.5))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; Productions in the executive module.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p@ arith input-first-digit ((g goal))
  (eq (is g) 'get-first-digit)
  [not (done g)]
  (no ((~d digit 1.0))
      (= (op-num ~d) 1))
  -->
  (multiple-value-bind (val str) (perceive-digit 1)
     (spew g (digit :value val
                    :op-num 1)
       str))
)

(p@ arith input-second-digit ((g goal))
  (eq (is g) 'get-second-digit)
  [not (done g)]
  (no ((~d digit 1.0))
      (= (op-num ~d) 2))
  -->
  (multiple-value-bind (val str) (perceive-digit 1)
     (spew g (digit :value val
                    :op-num 2)
       str))
)

(p@ arith done-inputting-the-first-digit ((g goal) (d digit 1.0))
  (eq (is g) 'get-first-digit)
  [not (done g)]
  (= (op-num d) 1)
  -->
  (modify g :done t)
)

(p@ arith done-inputting-the-second-digit ((g goal) (d digit 1.0))
  (eq (is g) 'get-second-digit)
  [not (done g)]
  (= (op-num d) 2)
  (no ((~a answer)))
  -->
  (modify g :done t)
)

(p@ arith add-digits ((g goal) (d1 digit) (d2 digit))
  (eq (is g) 'do-addition)
  [not (done g)]
  (= (op-num d1) 1)
  (= (op-num d2) 2)
  (no ((~a answer)))
  -->
  (spew g (answer :value (+ (value d1) (value d2))) (in arith 1.0) (in
exec 1.0))
)

(p@ arith done-doing-addition ((g goal) (a answer 1.0))
  (eq (is g) 'do-addition)
  [not (done g)]
  -->
  (modify g :done t)
)

;; perceive digits.
(setq *digits* '(3 5))

(defun perceive-digit (op-num)
  (values (nth (1- op-num) *digits*)
          (random 1.0)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Running a simulation…
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; set up working memory.
(add (start))
```

4CAPS Specification

(All material here is included from the original 4CAPS manual by Sashank Varma)

Modules

Introduction:

4caps supports the existence and interaction of multiple production systems. Each production system is called a module. Each module contains its own productions, wmes, capacity, and parameter settings (e.g., is tracing on?).

When 4caps is first started, a single default module will exist. It is customary to immediately delete this and define the modules that compose the model at hand. One module may be designated the current module. It will then field all commands that are not explictly directed at a module.

Modules may be embedded within one another. A subordinate model has access to the activation resources of its superordinate. This can be useful if the subordinate module experiences an activation shortfall but is able to borrow (at least some of) the missing resources from its superordinate. Currently, modules may be embedded to form a hierarchy; other topologies, such as DAGs and general graphs, are not supported.

Top-Level Commands:

(cmod)

Returns the current module.

(mods)

Lists the names of all modules. The current module's name is preceded by an asterisk. Each module name is followed by the names of its subordinate and superordinate modules, if any exist.

(trace-mods)

Arranges for the productions in all modules to print a trace of their activity.

(untrace-mods)

The complement of trace-mods; inhibits production tracing in all modules.

(set-cmod module-name)

module-name ::= symbol

Sets the current module to the module with name module-name. Note that a module name is a symbol.

(add-mod module-name)

module-name ::= symbol

Creates a module with name module-name. Note that this module is not installed as the current module; this must be done explicitly with the set-cmod command.

(del-mod module-name)

module-name ::= symbol

Removes the module with name module-name. If this module was the current module, there will be no current module afterwards.

(super super-mod-name {sub-mod-name}+)

super-mod-name ::= symbol
super-mod-name ::= symbol

Installs the module of name super-mod-name as the superordinate of one or more modules, each with name sub-mod-name. Because the shared capacity algorithms work only for the case of a hierarchy of modules (and not, for example, for a DAG or arbitrary graph of modules), this command will signal and error if one of the subordinate modules already has a superordinate.

(set-cap capacity)
(set-cap@ module-name capacity)

module-name ::= symbol
capacity ::= number

These commands set the level of activation resources in a module. set-cap sets this level in the current module, whereas set-cap@ operates on the module with name module-name. Module name must be a symbol. capacity is a number, a positive integer, rational, or decimal.

(set-trace boolean)
(set-trace@ module-name boolean)

boolean ::= t | nil

These commands are fine-grain relatives of trace-mods and untrace-mods, applying to a single module instead of all modules. set-trace applies to the current module, whereas set-trace@ affects the module with name module-name. The boolean value determines whether productions in this module will leave traces when firing or not.

(reset)
(reset@ module-name)

module-name ::= symbol

reset operates on the current module and reset@ on the module with name module-name. The cycle count, working memory, and any pending changes to working memory of the module are erased. All other module information is preserved, in particular its productions. This is a handy command when one wants to run the same module (e.g., a parser) several times on different working memories (e.g., sentences) without laboriously recompiling the productions before each run.

(wm)
(wm@ module-name)

module-name ::= symbol

These commands print the wmes of a module, the current one in the case of wm and the module with name module-name for wm@.

(matches)
(matches@ module-name)

module-name ::= symbol

Print the left-hand sides of all production instantiations that will fire on the next cycle when it is run. As with many of the commands above, matches applies to the current module and matches@ to the module with name module-name.

(add wme-pattern [activation])
(add@ module-name wme-pattern [activation])

wme-pattern ::= (class-name {:attribute value}*)
attribute ::= attribute of wmclass class-name, preceded by a ':'
value ::= any lisp expression
activation ::= number
module-name ::= symbol

add and add@ are both top-level commands and rhs actions. Each creates a wme from the supplied wme-pattern with an activation of activation. If activation is omitted, this value defaults to 1.0 units. add creates the wme in the current module and add@ in the module names by module-name.

A wme pattern is a list whose first element is the class-name of the wme to be created. Following the class-name are zero or more attribute-value pairs. Each attribute-value pair is an attribute, written with a preceding ':', followed by a value, which can be any lisp expression. These pairs serve to initialize the attributes of the newly-created wme, overriding default attribute values.

(spew t wme-pattern {amount | (module-name amount)}+)

wme-pattern ::= (class-name {:attribute value}*)
attribute ::= attribute of wmclass class-name, preceded by a ':'
value ::= any lisp expression
amount ::= number

spew is also a rhs action that can be used as a top-level command. It designates one or more wmes as targets and directs activation to each target in possibly multiple modules.

It is easiest to understand the simplest case for spew first before discussing complications. Using the same procedure as for add and add@, the wme-pattern is used to create a wme in the current module. The wme is assigned the amount of activation specified by amount. In this, the simplest case, spew is equaivalent to add. However, there are two special case which differentiate the commands.

If the wme created by wme-pattern is consistent with other wmes in the module (i.e., has the same attribute values), then the wme is not created. Instead, the activations of each consistent wme is incremented by activation. In this case, wme-pattern functions like a pattern or template for matching existing wmes, not as a mold for creating a new wme.

Another complexity arises when instead of a single amount, multiple amounts are specified. Each is either a numerical amount, in which case it is drawn from the current module, or a list where the first element is a module name and the second is an amount. In the latter case, the amount is drawn from the module with name module-name.

It is possible to write a spew command that combines both dimensions of complexity, directing activation to several wmes matching wme-pattern, where the activation directed to each wme is culled from several modules.

Notes:

In general, many commands take two forms, with one applying to the current module and the other, marked with an '@', applying to the module of the supplied name.

Note the deliberate symmetry in the names of the commands for creating and removing modules (add-mod and del-mod) and the commands for creating and removing wmes (add and del).

Top-level invocations of add, add@, and spew do not have an immediate effect. Rather, wmes are not created and activations are not adjusted until the beginning of the next cycle; it is as if top-level add, add@, and spew commands are buffered in an agenda. This means that if one executes an add command immediately followed by a wm command, the newly-created wme will not appear in the listing of working memory. However, if one cycle is run and the wm command is then executed, it will appear.

The (run) command is discussed in its own section below.

Working Memory Classes

Introduction:

Wmes in 4caps are object-oriented. They are instances of wmclasses. A wmclass A can inherit from another wmclass B, in which case wmclass A contains all of the attributes of wmclass B in addition to its own, unique attributes.

Top-Level Commands:

(defwmclass class-name (super-class*) {attribute | (attribute default-value)}*)

class-name ::= symbol
super-class ::= symbol
attribute ::= symbol
default-value ::= any lisp expression (e.g., a symbol, a number, a function call)

defwmclass is the main top-level command for defining a new wmclass.

class-name is the name of the new wmclass. It is a symbol.

The new wmclass inherits from each listed super-class. Each super-class is a symbol. All new wmclasses implictly and automatically inherit from the superclass 'base-wme'. This provides their activation and id attributes, described below. It is permissible for a new wmclass to inherit from no other wmclasses besides (the implictly listed) base-wme; in this case, no super-classes would be listed.

The remainder of the defwmclass command are the attributes of the new wmclass. An attribute is specified by either listing its name, which must be symbolic, or by a list whose first element is the attribute name and whose second element is the default value for that attribute. If unspecified, attributes default to a value of nil during creation.

A non-obvious behavior of defwmclass is the following: Consider a new wmclass C1 that inherits from a super-class C2, and C2 contains an attribute A with a default value of 'V1. Through inheritance, C1 will also contain attribute A with the same default value. This inherited default value can be overriden to 'V2 by including (A 'V2) in the call to defwmclass. In this way, inherited attributes may be defaulted and inherited defaults may be overriden.

The Activation and ID Attributes:

As noted above, every wmclass inherits from base-wme, which provides the activation and id attributes.

In general, you should not directly mess with the value of the activation attribute of a wme. Rather, the interfaces to this value, including the act accessor and the spew, add, and del rhs functions should be used instead. Both of these mechanisms are described below.

The id serves as a unique name for a wme. Such names are useful when embedding one wme as an attribute-value of another wme. The technique of wme-embedding is discussed below.

Productions

Introduction:

Productions are the formalism for encoding procedural knowledge. Just as wmes of 4caps are more object-oriented then in 3caps or ther production system language, the same is true of 4caps productions.

Top-Level Commands:

(p production-name (postive-wme-spec*) lhs --> rhs)
(p@ module-name production-name ({postive-wme-spec}*) lhs --> rhs)

module-name ::= symbol

```
production-name ::= symbol
positive-wme-spec ::= (wme-variable class-name [threshold])
wme-variable ::= symbol
class-name ::= symbol
threshold ::= number
lhs ::= test*
test ::= positive-test | negative-test
positive-test ::= lisp-predicate | [lisp-predicate*]
lisp-predicate ::= lisp expression (that returns nil or non-nil)
negative-test ::= (no (negative-wme-spec*) positive-test*)
negative-wme-spec ::= (wme-variable class-name)
rhs ::= action*
action ::= add-action | del-action | mod-action | spew-action | lisp-expression
add-action ::= ({add | add@ module-name}1 wme-pattern weight)
wme-pattern ::= (class-name {:attribute value}*)
attribute ::= attribute of wmclass class-name, preceded by a ':'
value ::= lisp expression
weight ::= number
del-action ::= ({del | del@ module-name}1 wme-variable)
mod-action ::= (mod wme-variable :attribute value)
spew-action ::= (spew source target {weight-expr}+)
source ::= t | wme-variable
target ::= wme-pattern | wme-variable
weight-expr ::= weight | (module-name weight)
```

p adds a production in the current module, p@ in the module with name module-name.

The production's name will be production-name. Note that if an attempt is made to add a production to a module with the same name as an existing production name, there are several policies for resolving the clash. The default policy is to augment the name of the new production with astericks until a novel name is produced. Other policies include signaling an error and permitting the redundancy.

Following the name is a list of specifications for positive wmes. Positive wmes are those wmes which must be present in working memory for the production to match. They contrast with negative wmes, described below, which must not be present in working memory. Each specification consists of two, sometimes three elements arranged in a list. The first element is the variable name by which the positive wme is known in the production. The second element is the wmclass of the wme. For a wme to match the specification, it must be of the specified wmclass or a subclass, e.g., a 'dog' wme will match a specification that calls for a wme of class 'mammal' if the dog wmclass inherits from the mammal wmclass. If there is a third element, it is a threshold specifying the minimum activation level a wme must have to match the specfication.

The remaining two aspects of a production are its lhs and rhs, which are separated by the '-->' marker.

The lhs consists of a series of condition elements (ces). Each ce is possibly nested. We wll distinguish between top-level ces and embedded ces. The lhs is considered as large conjunctive test of its top-level ces. Thus, a production will match a set of (positive) wmes if the wmes satisfy the (positive) specifications at the top of the productions (i.e., are of the right classes and have above-threshold activations) and if the wmes result in every top-level ce of the lhs yielding the value t, or rather, non-nil.

Top-level ces come in two flavors, positive and negative.

Positive ces are arbitrary lisp predicates, i.e., expressions that yield either nil or non-nil. There is one trap one must be aware of with positive ces. Consider the positive ce (not (animate n1)), where n1 is a wme variable of wmclass noun. One would expect this positive ce to be true if n1 is inanimate, for this would make (animate n1) nil and (not nil) is t.

4caps will not make this interpretation. To understand why, note that when adding a production to a module, 4caps decomposes it into as many primitive tests as possible, and when it contains the same primitive tests as an existing production, the two "share" the evaluation of these tests. This is done for efficiency purposes. The impact of this sharing is that (not (animate n1)) is split into two tests, temp=(animate n1) and (not temp), where 'temp' is conceptually a temporary variable. If an inanimate noun is added to the module, then temp=(animate n1)=nil. Whenever 4caps ecounters a negative test result, it assumes that no downstream tests that depend on it will match either. Thus, when ???