0 Introduction

4caps is a new cognitive architecture in the same family as CAPS (Thibadeau, 1982) and 3CAPS (Haarmann, 1995; Varma, 1990; 1992).

The major difference between 4caps and its predecessors (and indeed all other production system languages) is the ability to define multiple modules, where each module functions as a complete production system. There are mechanisms for modules to communicate with each other, and it is through cooperation and coordination that modules interact to perform complex cognitive tasks. The new module facility enables 4caps to support model given recent findings in cognitive neuroimaging.

The manual is divided into four sections, describing modules, working memory elements (wmes), productions, and rmacrocycles, respectively. Each section contains two parts. The first is a tutorial introduction to the relevant commands, anchored in the development of a production system that performs addition. The second part is a reference that describes each command in greater detail.

There are also three appendices. Appendix A lists the full 4caps code for the additiona model developed throughout the tutorial parts of each section. Appendix B describes the system requirements necessary to run 4caps, as well as the process by which 4caps is loaded. Finally, Appendix C provides detail on the Common Lisp functions that underlie 4caps. These functions can be used by programmers to automate aspects of the development of 4caps models and the running of simulations.

Questions can also be directed to the author at sashank@vuse.vanderbilt.
edu.

1 Modules

A module is an encapsulated production system. 4caps provides commands to list all known modules, to add and delete modules, and to specify hierarchical relationships between modules.

1.1 Tutorial

When 4caps is first started, it creates a default module:

```
Welcome to Macintosh Common Lisp Version 3.2!
?
#<MODULE MOD-1>
?
```

The default module has the name MOD-1. In general, all modules created by 4caps without explict, user-supplied names are given names of the form MOD-*n*, where *n* is a natural number.

Recall from above that the tutorial sections of this manual will develop a production system model that performs addition. The sections of this manual mirror the organization of 4caps models. In particular, the first thing all models should do is create their component modules.

The addition model will consist of two modules, one where the actual calculation takes place and an executive module that sequences the steps of the calculation. We have no need for the default module, so the first thing to do is delete it:

```
? (del-mod mod-1)
?
```

The del-mod command takes one argument, the name of the module to be deleted. If the goal is to delete all pre-existing modules and you are not sure of their names (or do not want to execute multiple del-mod commands), then it is easier to use the del-mods command:

```
? (del-mods)
?
```

This deletes all known modules.

The next step is to define the two modules of the addition model:

```
? (add-mod exec)
#<MODULE EXEC>
? (add-mod arith)
#<MODULE ARITH>
?
```

The add-mod command takes a single argument, the name of the new module to be added.

To ensure that all desired modules have been created and all extraneous modules deleted, it is useful to have 4caps list all known modules. This is accomplished with the mods command:

```
? (mods)
MODULES:
  EXEC
  ARITH
?
```
Thus, the commands executed thus far have achieved the desired state.

There is one final bit of business to attend to before the modules of the addition model can be considered completely defined. 4caps provides the notion of the "current module". This is the module that, unless otherwise stated, 4caps commands are assumed to apply to. A module may be designated as current in the following manner:

```
? (set-cmod exec)
#<MODULE EXEC>
?
```
The current module can be queried in two ways. The direct method is to use the cmod command:

```
? (cmod)
#<MODULE EXEC>
?
```
However, the mods command indirectly provides the same information

```
? (mods)
MODULES:
 *EXEC
  ARITH
?
```
by placing an asterisk before the name of the current module, if one exists.

Because exec is the current module, all commands not explictly directed to a module will be directed to exec. For example, consider the set-cap and set-cap@ commands that will be introduced in section 4 for specifying the activation capacity of a module. We can set the capacity of the exec module using the set-cap command

```
? (set-cap 5.0)
?
```
This works because 4caps assumes that the module targeted by the set-cap command is the current module, which is exec. In contrast, setting the capacity of the arith module requires using the set-cap@ command and explictly naming the targetted module:

```
? (set-cap@ arith 5.0)
?
```
This asymmetry of convenience means that the module of a model that is most central, i.e., will be handling most of the commands, should be made the current module to avoid having to use the @ variant of commands and providing explict module names.

4caps also provides the notion of module embedding. When one module, called the subordinate, is embedded in another module, called the superordinate, it gains access to additional activation resources. In the case of the addition model, it makes sense to embed the arith module within the exec module:

```
? (super exec arith)
?
```

This means that if the processing and storage demands in arith module exceed the 5.0 units of avilable activation, the shortfall can be made up with activation from the exec module. This naturally brings up the question of what happens if the exec module itself has an activation shortfall; this question is answered in section 4.

In general, super commands can be used to build a hierarchy of embedded modules. It is not possible for a module to have more than one superordinate module at the current time because the algorithms for sharing and constraining activation across modules do not generalize to this case.

Just as the mods command marks the current module, it also displays superordinate and subordinate information for each module:

```
? (mods)
MODULES:
  ARITH:  exec < leaf
 *EXEC:  root < arith
?
```

The output indicates that exec is the superordinate module of arith, and that arith has no subordinate modules, i.e., it is a leaf in the hierarchy of embedded modules. In contrast, exec is a root of the module hierarchy, i.e., has no superordinate module, although arith is its subordinate.

1.2 Reference

(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 superordinate module and subordinate modules, if they exist.

(add-mod [*module-name*])

> *module-name* ::= symbol
>
> Creates a module with name *module-name* (or a default name of the form MOD-*n* if no name is provided). Note that the new 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.

(del-mods)

Removes all modules from 4caps; there will be no current module afterwards.

(cmod)

Returns the current module.

(set-cmod *module-name*)

module-name ::= symbol

Sets the current module to the module with name *module-name*.

(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.

1.3 Notes

In general, many commands take two forms, one implictly applying to the current module and the other, marked by an @ suffix to the command name, taking an explicit module name as the first argument; the command is then applied to the module of that name.

Note the deliberate symmetry between the names of the commands for adding and deleting modules (add-mod and del-mod) and the names of the commands for adding and deleting wmes (add and del). This consistency should serve a mnemonic function.

2 Working Memory

Declarative knowledge in 4caps is represented in working memory elements.

2.1 Tutorial

In traditional production system languages, wmes are typed, attribute-value lists. For example,

(person ^name frank ^gender male ^occupation cabbie)

is an ops5-style wme of type person whose name, gender, and occupation attributes have the values frank, male, and cabbie, respectively. The attributes a wme possesses are delimited by its type because, for example, it makes little sense to stipulate a value for the gender attribute in the case of a wme of type car.

4caps extends the traditional conception of wmes by granting them full status as objects in the sense of object-oriented programming. This necessitates a slight change in terminology. Every wme is now said to be of a certain class. Each class has two properties. First, it inherits from one or more superclasses. In particular, it inherits all of their attributesm or slots. The second property of a class is the set of unique attributes it includes above and beyond those inherited from its superclasses.

2.1.1 Defining Wme Classes

To make this concrete, consider the following two wmes, written in 4caps notation:

(mammal :name noam-chimpsky :gender male)
(person :name frank :gender male :occupation cabbie)

It makes sense to assume that mammal is a superclass of person, i.e., wmes of class person include all of the attributes of class mammal as well as any person-specific attributes, such as occuptation.

This leads naturally to the question of how classes of wme are defined. This is done with the defwmclass command:

```
? (defwmclass mammal () name gender)
MAMMAL
? (defwmclass person (mammal) occupation)
PERSON
?
```

Consider the second definition first. It states that the class person is a subclass of (i.e., has as its superclass) mammal. Above and beyond the attributes which inherits from mammal (i.e., name and gender), person includes the additional attribute occupation. The first definition simply defines mammal as a class with attributes name and gender.

Note that mammal has no superclass. Or does it? It turns out that all classes include as a superclass the class base-wme. This class supplies two attributes that all wmes possess (because all wmes inherit from base-wme), namely activation and id. The activation of a

wme is treated specially by 4caps, in ways which are described later. The id attribute has a different value in each wme. It is useful for simulating pointer referencing.

2.1.2 Adding Wmes

Once a wme class has been declared using the defwmclass command, wmes of that class may be created using the add, add@, and spew commands. These commands have been defined above, and will be covered in greater depth in the next section. The following series of commands provides some feel for their effects:

```
? (add (person :name 'frank :gender 'male :occupation 'cabbie))
(PERSON :OCCUPATION CABBIE :GENDER MALE :NAME FRANK :ACT
0.00 :ID 1)
? (run 1)
? (wm)
============================================================
MOD-1
------------------------------------------------------------
  1.00:  (PERSON :OCCUPATION CABBIE :GENDER MALE :NAME FRANK
:ID 1)
============================================================
?
```

The add command has been used to add a wme of class person. This wme is added to the current module MOD-1; a different destination could have been specified if the add@ command was used instead.

The add command specifies the class of the wme (person) and a set of attribute-value pairs. Notice that each of the values has a quote mark before it. This is because in Common Lisp, the language on which 4caps is written, a symbol is normally interpreted as a variable and it is evaluated to produce a value unless the symbol is preceded by a quote, which indicates that the symbol is not a variable, but a literal value. In the example above, the value of the name attribute is not the value of the variable frank, but the literal symbol frank, so it is quoted.

The add command echos the wme to the screen. Notice that two extra attributes have been added, activation and id. These are just the attributes inherited from person's superclass base-wme. The initial value of activation is 0.00. This is misleading. The wme will have an activation of 1.0, the default, as soon as the run command is used to begin a macrocycle. That this is true can be confirmed by examing the activation of the wme after one such cycle has been run, which is revealed by using the wm command to list the contents of working memory. The id attribute is given an initial value different from the id value of all wmes created thus far. Thus, it is a unique "name" for the wme which other wmes may use as a pointer to point to it.

The add@ and spew commands are used similarly to create new wmes of specified classes.

2.1.3 The Wme Class Hierarchy

It is important to take seriously the hierarchy (actually, directed-acyclic graph) of wme classes, rooted in the "cosmic superclass" base-wme. Any wme of class mammal is, of course, of class mammal. A surprising addendum to this tautology is that any wme of a subclass of person is also of class mammal. For example, the wme of class person created above is also considered to be of class mammal.

One can confirm this by using the Common Lisp predicate subtypep, which returns T if the first class is a subclass of the second class:

```
? (nth-value 0 (subtypep 'person 'mammal))
T
? (nth-value 0 (subtypep 'mammal 'person))
NIL
?
```

In other words, person is a subclass of mammal (i.e., all persons are mammals) but not vice versa (i.e., all mammals are not persons). Which is of course true.

2.2 Reference

2.2.1 Defining Wme Classes

(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 wme class.

class-name is the name of the new wme class.

The new wme class inherits from each listed super-class. All new wme classes implictly and automatically inherit from the superclass base-wme. This provides their activation and id attributes, described below. It is permissible for a new wme class 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 wme class. 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.

2.2.2 Adding New Wmes

(add *wme-pattern* [*amount*])
(add@ *module-name wme-pattern* [*amount*])

> *wme-pattern* ::= (*class-name* {:*attribute value*}*)
> *class-name* ::= symbol
> *attribute* ::= attribute of wmclass *class-name*
> *value* ::= any lisp expression
> *amount* ::= 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 *amount*. If *amount* 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*}*)
> *class-name* ::= symbol
> *attribute* ::= attribute of wmclass *class-name*
> *value* ::= any lisp expression
> *amount* ::= number
> *module-name* ::= symbol
>
> 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 equivalent 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 activation of each consistent wme is incremented by *amount*. 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 a numerical *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.

2.3 Notes

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.

Top-level invocations of add, add@, and spew do not have an immediate effect. Wmes are not created and activations are not adjusted until the beginning of the next macrocycle; it is as if top-level add, add@, and spew commands are buffered in a module-specific 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 macrocycle is run and the wm command is then executed, it will appear.

A non-obvious behavior of defwmclass is the following: Consider a new wme class 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.

wm-default-act, *wm-front-act*, *wm-summ-embedds*, *wm-elide-nils*

3 Productions

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.

3.1 Tutorial

3.2 Reference

(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

3.3 Notes

compiling, *p-name-collision-policy*, *default-ce-thresh*

4 Macrocycles

4.1 Tutorial

4.1.? Module Tracing

Another property of modules that can be set is their tracing status. Each module has a flag that indicates whether a detailed trace will be printed when its productions fire. We can turn on tracing in both modules with the following series of commands:

```
? (set-trace@ mod-1 t)
? (set-trace t)
?
```

Note that the final command turned tracing on in the current module, SPATIAL. We can ensure that these commands had the desired effect:

```
? (trace?@ mod-1)
T
? (trace?)
T
?
```

We can use the same commands to turn tracing off in MOD-1 and to verify this change of state:

```
? (set-trace@ mod-1 nil)
? (trace?@ mod-1)
NIL
```

It is common to either trace all modules, during say model development, or to trace none, during say the running of simulations to fit data. Instead of changing the tracing status of each module individually, 4caps provides two commands that affect tracing in all modules:

```
? (untrace-mods)
? (trace?)
NIL
? (trace-mods)
? (trace?)
T
?
```

4.2 Reference

4.2.? Module Capacity

(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*. *capacity* is a positive number.

4.2.? Module Tracing

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

module-name ::= symbol
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.

4.2.? Other Commands

(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*.

(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.

4.3 Notes

to do:
remove the need for brackets by introducing a no-decomp mode.
the impl- convention
a note on regular expression notations
need to consoildate some of the module manipulation commands in a mod-mod command
standardize the names trace-mods, untrace-mods, set-trace, and set-trace@.
the run command
the (act) accessor
wme embedding
the wme-specs of productions should default like let variables if no class, or no class and surrounding parentheses, are present. also need to permit arbitrary threshold tests.