3 Symbolic Aspects of Knowledge Representation

4CAPS is a hybrid architecture, encompassing both symbolic and connectionist processing styles. This chapter describes the symbolic aspects of 4CAPS knowledge representations; the connectionist aspects are covered in the next chapter.

The symbolic aspects of 4CAPS follow from its status as a production system interpreter. As such, knowledge is represented in two ways in 4CAPS. The transient bits of information that characterize the contents of working memory during processing are encoded as *declarative memory elements*. Long-term memory is viewed as an associative memory that incrementally transforms working memory over time. Knowledge in long-term memory is encoded as *productions*.

4CAPS differs from other production system interpreter-based architectures in two major ways.

First, it is *neoclassical*, to use Anderson's (1983) terminology, in not positing a third memory system. That is, the architecture is neutral on whether long-term declarative knowledge is encoded in a separate store or whether such knowledge is encoded procedurally. Techniques for representing long-term declarative knowledge in 4CAPS are discussed in a later chapter.

The second difference between 4CAPS and similar architectures is the power of its knowledge representations. Conventional production system interpreters provide relatively impoverished declarative and procedural representations. Declarative memory elements in 4CAPS are based on the Common Lisp object system, and can thus be used to build the kinds of abstractions that programmers familiar with object-oriented programming languages will want to build. Similarly, productions are based on Common Lisp methods/functions, and thus inherit great power and flexibility.

3.1 Declarative Memory

Working memory contains storage and processing aspects. The storage function of working memory is to maintain a representation of the current processing state. This state is composed of a set of declarative memory elements.

Each declarative memory element belongs to a particular declarative memory *class*. The class of a declarative memory element governs its internal structure and the operations which it admits. 4CAPS defines a single declarative memory class, `base-wme`. 4CAPS models can define additional declarative memory element classes. For example, it is common to define a `goal` class so that goals can be created in working memory.

Each declarative memory elements contains a number of named components, called *slots*, which can have a single *value*. Other symbolic architectures refer to slots as *attributes*, while connectionists prefer the terms *features* or *microfeatures*. Object-oriented programming languages sometimes use the term *instance variables* for the same concept.

The class of a declarative memory element dictates the slots it contains. For example, all declarative memory elements of class `base-wme` contain the slots `id` and `act`. The value of a slot can be nearly anything: a number, a symbol, a string, another declarative memory element, etc.

The printed representation of a declarative memory element includes its class and various slot/value pairs between parenthetical delimiters: `(`*class :slot1 value1 :slot2 value2 ...*`)`. For example, a declarative memory element of class `base-wme` might print as: `(base-wme :id 21 :act 1.0)`.

Declarative memory elements support certain operations. Minimally, the values of all slots may be accessed. The definition of additional operations on declarative memory elements is discussed in a subsequent chapter.

3.1.1 Declarative Memory Classes

3.1.1.1 Basic Class Definition

The **`defwmclass`** command is used to define a declarative memory class. For example:

```
(defwmclass addition-fact ()   ; the class of additive facts
      operand1                 ; the first addend
      operand2                 ; the second addend
      sum)                     ; the sum of the two addends
```

This command defines a new declarative class `addition-fact`. Following the name of the new class is a list of existing declarative memory classes from which it *inherits*. If class A inherits from class B, then elements of class A automatically contain all slots of class B in addition to those slots specified in class A's definition. In the example above, `addition-fact` appears to inherit from no other declarative classes. In fact, this is not true. All declarative memory classes automatically inherit from the `base-wme` class. That is, the class definition above can be written as:

```
(defwmclass addition-fact (base-wme) ...)
```

The classes from which a class inherits are called its *direct superclasses*, or alternatively, *parent classes*. Finally, a list of slots possessed by the new class are listed. Thus `addition-fact` contains five slots: `id` and `act` are inherited from its direct superclass `base-wme`, and `operand1`, `operand2`, and `sum` are unique to it.

(Note that the bits of text following the ";" at the end each line of the class definition are comments, text meant for consumption by human programmers. They document the purpose of each aspect of the definition. They are, of course, optional. It is a good idea to comment one's code to facilitate its comprehension by other programmers in the future (and the author as well).)

3.1.1.2 Inheritance

The general benefit of declarative memory classes and inheritance can be seen if we consider the task of writing a model of arithmetic performance. The model can depend on a storehouse of general arithmetic knowledge, which we can capture with the definition:

```
(defwmclass arithmetic-fact ()
      operand1
      operand2)
```

This class of arithmetic knowledge can then be *specialized* for the four primary arithmetic operators:

```
(defwmclass addition-fact (arithmetic-fact)
      sum)
(defwmclass subtraction-fact (arithmetic-fact)
      difference)
(defwmclass multiplication-fact (arithmetic-fact)
      product)
(defwmclass division-fact (arithmetic-fact)
      ratio)
```

This *refactored* scheme makes clear that `addition-fact` is a *subclass* of (or just *isa*) `arithmetic-fact`, and that `arithmetic-fact` is a subclass of `base-wme`.

3.1.1.3 Default Slot Values

When a declarative memory element is created, it will contain all of the slots specified in the class definition and the definitions of the class's superclasses. The initial values of these slots will be `NIL`, which is to say that they will have no value by default. It is possible to specify non-`NIL` default values for slots. This is done by including the slot in a list whose second value is the desired default value. Extending the example above:

```
(defwmclass arithmetic-fact ()
      operand1
      operand2
      (operator '?))
```

`Arithmetic-fact` contains a new slot, `operator`, which has a default value of `'?` (instead of `NIL`).

3.1.1.4 Overriding the Default Slot Values of Inherited Slots

Because `addition-fact` inherits from `arithmetic-fact`, and because the default value of the `operator` slot in `arithmetic-fact` is `'?`, the same will be true for `addition-fact`. But this is not desirable. We would like the ability to *override* the default values of slots inherited from superclasses. This can be done using the following syntax:

```
(defwmclass addition-fact (arithmetic-fact)
      sum
      :default-initargs :operator '+)
```

This definition states that while `addition-fact` inherits from `arithmetic-fact`, its default value for the inherited operator slot is `'+`, not `'?`. Following this template, we can redefine the remaining subclasses of `arithmetic-fact`:

```
(defwmclass subtraction-fact (arithmetic-fact)
      difference
      :default-initargs :operator '-)
(defwmclass multiplication-fact (arithmetic-fact)
      product
      :default-initargs :operator '*)
(defwmclass division-fact (arithmetic-fact)
      ratio
```

```
        :default-initargs :operator '/)
```

3.1.1.5 A Top-Level Command on Declarative Memory Classes

Only one command, **defwmclass**, is used to define declarative memory element classes. Its formal syntax is:

(defwmclass *class* **(***{superclass}**)**
{slot | **(***slot default-value***)***}**
*{***:default-initargs** *{inherited-slot overriding-default-value}*}***)**

The name of the new class is *class*. Its direct superclasses are the zero or more *superclass* names listed. All new classes automatically inherit from `base-wme`. Following the list of superclasses are the slot specifications for the slots unique to class; there may be zero or more of these. A slot specification is either a *slot* or a list of two items, a *slot* and a *default-value*. If `:default-initargs` appears anywhere in the list of slot specifications, then it is followed by an even number of default overrides. A default override consists of an *inherited-slot* followed by an *overriding-default-value*.

3.1.2 Declarative Memory Elements

Recall that this chapter covers the symbolic aspects of 4CAPS. From this perspective, declarative memory elements can be discretely *added* and *deleted* from working memory. This hard distinction will be softened in the next chapter, which covers the connectionist properties of 4CAPS. There, the gradual excitation and inhibition of declarative memory elements will be described.

3.1.2.1 Adding Declarative Memory Elements

Once a declarative memory class has been defined, it can be used to create declarative memory elements. This is done via the **add** command. For example:

```
(add (addition-fact :operand1 5  ; 5+7=12
            :operand2 7
            :sum 12)
     1.0)
```

The **add** command takes one or two arguments.

The first argument must be a *template* for a declarative memory element. Templates resemble the format in which 4CAPS prints declarative memory elements. They are delimited by opening and closing parentheses. The first element of the template is the class of the declarative element being created. In the example above, this is an `addition-fact`. We know from the definition of this declarative element class that it contains six slots: `id` and `act` inherited from `base-wme`; `operand1`, `operand2`, and `operator` inherited from `arithmetic-fact`; and `sum`, which it possesses uniquely. 4CAPS automatically defines the `id` value; the `act` value is handled separately, as described below. The definition of `addition-fact` automatically provides the desired `operator` value of `'+`. All that remains is to define the two operands and sum expressed by this declarative memory element. This is done following the class. For each slot and value to be defined, the slot is written with a preceding colon. This is called a

keyword in Common Lisp and thus 4CAPS. It signifies that the slot is not a value, but names an argument of some sort, which follows immediately. The desired value of the slot follows the keyword. Thus, the above command creates an `addition-fact` with user-defined values of `5` for the `operand1` slot, `7` for the `operand2` slot, and `12` for the `sum` slot.

The second argument to **`add`** is optional. It specifies the desired value of the `act` slot of the newly-created declarative memory element. If no second argument is given, it defaults to `1.0`. Additional properties of element activations are described in subsequent chapters.

3.1.2.2 Listing Declarative Memory Elements

The set of declarative memory elements defines the contents of working memory. These can be inspected using the **`wm`** command. The following is a sample interaction with 4CAPS that follows the addition of the `addition-fact` shown above with the addition of a second declarative memory element and a listing of working memory:

```
> (add (subtraction-fact
            :operand1 5
            :operand2 7
            :difference -2)
    1.0)

> (wm)
=====================================================================
all modules
---------------------------------------------------------------------
1.00: (addition-fact :id 1 :operand1 5 :operand2 7 :operator +
            :sum 12)
1.00: (subtraction-fact :id 2 :operand1 5 :operand2 7
            :operator - :difference -2)
=====================================================================
>
```

The **wm** command prints each declarative memory element in working memory in template form. That is, between parenthetical delimiters is first the class and then each slot in keyword form followed by its value.

There are three things to notice about the sample output above. First, the value of the act slot is fronted for each declarative memory element. This vagary of activation values is covered in the next chapter. Second, 4CAPS did in fact automatically assign values to the `id` slots and defaulted the values of the `operator` slots as mandated by the class definitions. The final thing to notice is that the header specifies that this is the working memory of "all modules." Models comprised of multiple modules are covered in a subsequent chapter.

The **`wm`** optionally takes one or more arguments. These are classes by which the listing of working memory should be filtered. For example:

```
> (wm addition-fact)
=====================================================================
```

```
all modules filtered by addition-fact
--------------------------------------------------------------
1.00: (addition-fact :id 1 :operand1 5 :operand2 7
            :operator + :sum 12)
==============================================================
> (wm subtraction-fact)
==============================================================
all modules filtered by subtraction-fact
--------------------------------------------------------------
1.00: (subtraction-fact :id 2 :operand1 5 :operand2 7 :operator -
            :difference -2)
==============================================================
>
```

Providing filter classes allows the inspection of just those declarative memory elements of interest, which is useful when the contents of working memory swell. Note that the header of each listing lists the classes by which working memory was filtered. When no filter classes are provided, it as if `base-wme` is used; because all declarative memory elements inherit from `base-wme`, they are considered members of this class, and are thus printed.

It is left as an exercise to the reader to guess (and verify!) the results of typing the `(wm arithmetic-fact)` and `(wm multiplication-fact)` commands at this point.

3.1.2.3 Deleting Declarative Memory Elements

There is a facility for deleting declarative memory elements from working memory: the **`del`** command. The **`del`** command takes a single argument, the declarative memory element to be deleted. The problem arises as to how one "grabs hold" of a declarative memory element in working memory to "pass" to the **`del`** command for deletion. This task is performed by the **`get-wme`** command. For example:

```
> (get-wme 1)
(addition-fact :id 1 :act 1.00 :operand1 5 :operand2 7
      :operator + :sum 12)
> (get-wme 2)
(subtraction-fact :id 2 :act 1.00 :operand1 5 :operand2 7
      :operator - :difference -2)
>
```

The **`get-wme`** command takes a single argument, a number. It returns the declarative memory element with the same numeric value for its `id` slot, or `NIL` if no such element exists. This is the first example of the `id` slot's usefulness as a reference or alias for declarative memory elements.

The following **`del`** command makes use of the **`get-wme`** command to delete the `subtraction-fact` present in working memory:

```
> (del (get-wme 2))

> (wm)
==============================================================
all modules
--------------------------------------------------------------
```

```
1.00: (addition-fact :id 1 :operand1 5 :operand2 7
      :operator + :sum 12)
==========================================================
>
```

3.1.2.4 Top-Level Commands for Declarative Memory Elements

The **add** command creates a new declarative memory element in working memory. Its formal syntax is:

(**add** (*class {*:*slot value}**) *[activation-level]*)

The class of the new declarative memory element is class. Following the class are an even number of items (possibly zero). For each pair, the first is a *slot* expressed in keyword form, i.e., with a preceding colon. The second is the *value* to be initially assigned to *slot*. The *class* and *slot-values* are delimited by parentheses; this entire form serves as a template for the to-be-created declarative memory element. Following the template is optionally an *activation-level*, number indicating the desired initial activation level of the declarative memory element.

The **wm** command lists a subset of the declarative memory elements that comprise working memory. Its formal syntax is:

(**wm** *{class}**)

This command takes zero or more *class*es as arguments. If none are supplied, it is as if a single argument, `base-wme`, was specified. Every declarative memory element in working memory is considered. If the class of the element is one of the supplied classes *or a subclass of one of the supplied classes*, then the element is printed. Thus, if no explicit class argument is given and the implicit `base-wme` is used, then the entire contents of working memory will be printed because each declarative memory element is a subclass of this class. When a declarative memory element is printed by the **wm** command, its activation is fronted (i.e., printed first).

The **get-wme** command returns the declarative memory element with the same value for its `id` slot as the single argument given to the command. Its formal syntax is:

(**get-wme** *id-number*)

If no matching declarative memory element exists, `NIL` is returned. This command is useful for retrieving declarative memory elements needed as arguments for other commands.

The **del** command deletes from working memory the declarative memory element supplied as its sole argument. Its formal syntax is:

(**del** *declarative-memory-element*)

It is an error to supply anything but a declarative memory element as an argument to this command. It is often useful to embed a call to **get-wme** in **del** commands to "get hold of" the *declarative-memory-element* to be deleted.