PARMENIDES: A Class-Based Frame System

Version 1.5

22 April 1991

Peter Shell
Jaime Carbonell

1. Introduction

PARMENIDES (PropagAting, Rulekit-Motivated ENgine for Instance DEScriptions) is a frame-based knowledge representation system. It is influenced by **SRL**, **Framekit**, **CommonLoops**, and the Spice lisp structure implementation. The slot and facet access functions have comparable speed to the Slisp slot access functions. It has some of the Framekit and SRL functionality, such as facets, demons, and user-defined relations. However, like Loops, it makes a distinction between classes and instances. This means that instances can only be an instance of one class, while classes may be subclasses of (have an *is-a* relation with) more than one class. Instances may not have any slots that their classes don't have. Classes describe a way to make instances, and instances only participate in the frame network through their classes. By storing much of the frame information at the class level, the amount of storage needed for instances is reduced. For now, classes can not also be instances, but this could be implemented in the future. Parmenides is implemented in Common Lisp, so it is assumed that the user is familiar with this dialect of Lisp.

The name is in recognition of Plato's *Parmenides* dialogue, in which he is confronted with the "third person argument". This is the notion that in order to describe and generate instances through the generative powers of classes, the classes in turn need a meta-class to generate them, and so on. This argument is used as an excuse not to implement meta-classes in this frame system.

Other features of **Parmenides** include:

- *Propagation* instead of *inheritance*. See the section on propagation for more details.
- *Optional facets*. Usually, slots contain facets, which can be thought of as annotated property lists of a slot. For example, one might want to store both a value and degree of certainty with some slots. However, if only one value and no facets need be stored, then the slot may be declared to not have facets, thus making access to and representation of that slot more efficient.
- *Extendable Frames*. Since frames are represented as adjustable arrays, it is possible to add slots and facets dynamically (i.e., after the class has been defined).
- *Class slots*. **Parmenides** supports class slots, which are ways to specify meta-information about the classes, and apply not to any particular instance or slot, but the class itself. For example, the *propagate* attribute is a class slot.
- *Pre-access* and *post-access* demons. **Parmenides** allows demons to called both before and after access to values. See the section on demons.
- *Multiple Language Messages*. Parmenides messages may be displayed in a number of languages. The currently supported languages are English and Spanish, although it is easy to add messages in new languages. See section 6 for details.
- *User-Defined Relations*. The *instance* and *is-a* relations are special hard-wired relations, but the user may define his/her own relations as frames. Inverse relations are also supported.
- *User frame files*. Files containing **def-frame** commands may be compiled when a set of frames is known to be stable. After that, the compiled frame file may be loaded without having to load in the source frame file, resulting in considerable loading and executing speed up.
- *A freelist of frames*. **Parmenides** uses a *freelist* in order to more efficiently manage memory. Whenever a frame is removed, it is put on the freelist, and when a new frame is needed, if there is one of the appropriate type already on the freelist, then that one is recycled. The function **remove-frame**, described below, releases frames to the freelist, which means that you shouldn't expect to be able to use the frame after calling them.

2. Propagation and Caching versus Inheritance

Inheritance is a process by which properties of frames are inferred from other related frames, typically through is-a or instance links. Many frame and schema languages perform inheritance at the time of retrieval. This saves space because when values are shared by a whole tree of frames, they only need to be stored in one place. For example, the fact that animals breathe air need only be stored in the animal class frame, and not in any subclasses, such as mammal or dog, since the breathes slot of a dog frame can be inherited from the animal frame. However, retrieving can be very slow if retrieving with inheritance is performed often. Also, if other programs use a frame network as a data base, then if a class is changed, it will not necessarily know the implications of this change unless it tests all of its data again.

For these reasons, an alternative to inheritance, *caching*, of selected slots and values, is allowed in **Parmenides**. Caching a value simply means storing that value directly in a frame. If a slot is designated a *cached* slot by a class, then the slot and its values will be stored in all subclasses of that class. Thus, instead of performing inheritance at access time, **Parmenides** is able to retrieve the data directly from the frame. For example, suppose the *animal* class is first defined with a slot *(breathes (value 'air))* and that the *breathes* slot is defined to be *cached*. If a *dog* frame is defined as *is-a animal*, the slot *(breathes (value 'air))* is immediately stored in the *dog* frame (unless of course if the dog frame were to provide its own specification of the *breathes* slot). Whenever an instance of dog is made, that instance also gets that slot as a default cached value.

Cached slots are designated by the *cache* class slot (see the description of the **def-frame** command for the details of designating cached slots). The tradeoff of cached slots is that it takes more space to cache all values. Thus, if a system never need know if Fido the dog breathes, then it shouldn't specify the breathes slot as cached. Standard inheritance is still performed if a cached value can't be found in a frame (see the Command Summary, below).

Under some circumstances, it may be necessary to change a class definition after it and its sub-classes have already been defined. To ensure that the correct values are always cached, in the event that a class is changed, the change is propagated to the sub-classes and instances. There is a distinction between *cached* and *local*. A cached value is one that is stored directly in a frame, but which could have been inherited from somewhere else. A local value is one which was originally stored in the frame (i.e., not inherited from somewhere else). **Parmenides** retains the distinction between cached and local slots by keeping a special *depth* facet on slots that participate in propagation[1]. The depth facet indicates from how far up in the frame hierarchy the value was inherited. When the depth is 0, the value is local. If a slot has a depth which is less than the depth of a slot which is trying to propagate a new value to it, then that value won't be propagated to it. The *local-p* function tells whether or not a certain slot has a local value.

Propagation is done according to the global system flag !!inheritance-type, so that when a cached value is retrieved, it will be the same as if the specified inheritance type was performed. !!Inheritance-type must be a keyword, and its value may be either *:dfs* (depth-first) or *:bfs* (breadth-first). It is initially set to *:dfs*. (Note that currently *:bfs* is not currently implemented).

3. Parmenides Commands

Parmenides commands are divided into the following categories: Class definition, instance creation, data retrieval, data storage, predicates, and miscellaneous commands.

[1]That is, slots which have facets and whose class have the *propagate* property.

3.1. Class Definition Commands

It is necessary to define a class before making frame instances, since classes are prototypes of instances.

def-frame <name> <cplist> &rest <slot-descriptions> Top level frame definition function. Defines a frame class of name <*name*> and makes a default instance of it with the given default values. Default values are evaluated at run time. Defines a make function for that class, defines slot access functions for each slot, and defines facet access functions for the first facet (typically *value*) in each slot in <*slot-descriptions*>. Optionally defines set and setf methods for slots and facets (see section 7.2 of **Common Lisp, the Language** by Guy Steele for an explanation of setf methods). The arguments will now be described.

3.1.1. slot-descriptions

<*slot-descriptions*> is a list of slot descriptions, which are of the form:

```
<slot-description> := <slot-name> <slot-contents>
<slot-contents> :=  <constant> | <facet-plist>
<facet-plist>   :=  '(' {<facet-name> <default-value>}+ ')'
<default-value> :=  (any lisp object)
<facet-name>    :=  <slot-name>
<slot-name>     :=  :<symbol>
<constant> is any atom, including NIL.
a <constant> in the facet field denotes a facet-less slot.
Note: constants are not evaluated, but default-values are.

Example of a faceted slot:
:height (:value 'tall :time-inferred 23 :certainty 1.0)

Example of a slot without facets:
:height NIL
```

A value facet with default value NIL is automatically inserted into slots that have facets but don't have a value facet.

Note that slot descriptions may be inherited from other classes, as described in the <*is-a*> description, below.

3.1.2. cplist

<*cplist*> is a class plist, which is the way to define class slots. It has the same syntax as <*slot-descriptions*>, and can also be inherited from super-classes. Certain class variables are recognized by the system. For efficiency, they are all facetless. These are:

- *:is-a* Default: NIL. An ordered list of class names which the given class is a sub-class of. Slot and other cplist descriptions are inherited from the superclasses. When there is more than one description of the same slot or cplist in the superclass list, the first such superclass in the *is-a*'s list is used. Local descriptions take precedence for the *is-a* relation. Because **Parmenides** performs propagation instead of inheritance, it needs to have all the information about the superclasses at class definition time. Thus, frame files should define the highest level frames first, working down.

- *:cache* The default is the union of what all the parents cache. A *cache* slot in a class means that that slot specification is inherited by any sub-classes of the class. The value of the *cache* class slot, if specified, should either be a list of the names of the slots to cache, or the symbol :*ALL*, in which case all slots in the current class will be cached. The list of slots to cache is added to what all the parents cache; thus, cached slots are themselves cached to all the descendants of the class which defined them. To override caching what the parents cache, explicitly specify NIL as the value of this c-slot.

- *:setable* Default: The slots which the parent classess explicitly declare to be setable. If the value of this slot is *T*, then a **set** function will be defined for each slot, analogous to the slot access functions. These set functions are named **set-**<*frame*><*slot*>, e.g. **set-dog-breathes**. Their first argument is the frame to modify, and the second argument is the new value to put in the slot. A set function will also be defined for the *value* facet for slots which have facets. If the value is *:setf* then, in addition to the set functions, a setf method for each slot will be defined. This allows users the full power of setf methods (at a slight increase in size of compiled frame files), such as being able to push onto slots. Finally, if the value of this slot is a non-nil list, then the first element of the list should be *T* or *:setf*, and the second element should be a list containing the slots for which a *set* method is to be defined. If the parents explicitly declare slots as settable, then they are also settable by the children; to override this inheritance, declare the value of *:setable* to be NIL.

- *:getters* Default: all slots. Normally, retrieval functions are written for each slot. This class slot, whose value must be a list of slot names, :*ALL* or NIL, may be used to force **Parmenides** to write retrieval functions for only specified slots and their facets. It is treated the same way the *:cache* facet is treated, i.e., the keyword :*ALL* is recognized, and it is inherited from super-classes. This makes **Parmenides** more efficient when there are many frames with many slots, and a retrieval function is not needed for most slots.
- If you use FRulekit, the *:getters* slot should not be altered since it should normally be :*ALL, which is the default.
- *:propagate* Default: T. If true, then Def-frame will enable the class to participate in the propagation process when a class is changed. Since the propagation process requires that the slot that is being changed has at least a value and depth facet, **def-frame** will automatically add these facets in faceted slots of frames that participate in propagation. If propagation is not going to be needed, then turning propagation off will result in saving space. If a frame is propagatable, then all of its parents and descendants must also be propagatable; otherwise an error will occur. Note that slots may be cached even when propagation is turned off, since propagation only takes effect when slots are modified, and the *cache* attribute takes effect when frame classes are defined.
- *:if-needed* Default: NIL. Demons may be specified both as class slots and as slot facets. See section 5, below, for a description of this and the next two demon class slots.
- *:pre-if-set* Default: NIL.
- *:post-if-set* Default: NIL.

In addition to the *is-a* relation, user-defined relations may be class slot names. For example, if the user defines the *part-of* relation and the *table* frame, then in the class slot plist, *part-of table* would be allowed. Since class slots are meant to modify the definition of the class, user-defined relations should be put into class slots when they affect which slots the class would contain. See the section on relations, below, for more details.

3.1.3. Discussion

Access functions take a frame instance as argument. Every slot has an associated slot access function, which has the name <*framename*>-<*slotname*>. For example, the function **person-height** would return the height of a person instance. Regardless of whether the slot has facets, the slot access function will return that slot. If the slot has no facets, then the value will be the value stored by the user; otherwise it will be a slot data structure containing all facets fillers. If the *:setable* class facet is true, then **Parmenides** would define the function **set-person-height**, which takes a frame instance and value as arguments, and stores the given value in the *:height* slot of the specified person instance. If *:setable* is set to *:setf*, then in addition, the person *height* slot is setf-able, e.g., one may perform: *(push 10 (person-height person1))*.

For slots with facets, a slot access function for the first facet of that slot will be defined in a way similar to slot functions for frames with no facets, except facet names will be separated from slots by a "." rather than a dash (to distinguish slots from facets). For example, if the *size* slot for the house frame had a *value* facet, then the value of size of a house would be retrieved through the function *house-size.value*. If the *:setable* class facet is true, then a set function will also be defined for the house/size/value facet. If it is set to *:setf* then that facet will also be setf-able; e.g., it would be possible to modify the *value* of the *size* slot of a house instance by saying: *(setf (house-size.value house1) <new-size-value>)*.

It is important to note that there are two different ways to refer to frames in **Parmenides**. Classes are always referred to by name, as in Framekit, and are defined by **def-frame**. *Instances*, on the other hand, should be thought of as abstract data structures which are created by the make functions (below), and which the access functions operate on. However, the make functions allow one to name an instance. This gives the user two different ways to refer to the same instance.

3.1.4. Examples of Def-frame

```
(def-frame animal (cache :*ALL*)
  :consumes yes
  :reproduces (:value 'sexually))
```

This defines a class named *animal*, which has no superclasses and has *consumes* and *reproduces* slots. The functions *animal-consumes*, *animal-reproduces*, and *animal-reproduces.value* would be defined. Since the *consumes* slot has no facets, there is no need for an *animal-consumes.value* function. All of its slots are cached into its descendants.

The function **make-animal** would be defined, and it would take the keywords *:consumes* and *:reproduces* as arguments. To make an instance of animal with reproduces value NO, we would issue the following command:
(setq a1 (make-animal 'a1 :reproduces '(:value NO)))
This makes an animal which doesn't reproduce, calls it a1, and binds it to the lisp variable a1. Note that since we didn't specify the *consumes* slot, it receives as default 'yes.

Note the placement of colons ":" to designate keywords. Slot and facet names are stored internally as keywords so that the dependence on which package the user is in is minimized. However, **def-frame** and the slot and facet retrieval and storage functions take either keywords or non-keywords as slot and facet arguments; they will coerce slot and facet names to keywords if they are not already. Thus, it is slightly more efficient to use keywords as arguments, but not necessary. Frame names are not stored as keywords.

Every good document needs a transporter frame, so we'll define it as:

```
(def-frame transporter (cache (function))
  :function ptrans
  :user (:value 'rider))
```

This defines a frame class *transporter*, with slots *function* and *user*. The *function* slot has no facets - its default value is *ptrans*.

Now we can define a more complex frame, horse, in terms of these two frames:

```
(def-frame horse (is-a (animal transporter))
  :consumes (:value 'oats :when 'daily))
```

The horse class will have the slots: *consumes, reproduces,* and *function*. Note that while the horse class obtains the *reproduces* and *function* slots from the animal and transporter classes, the *consumes* slot is locally defined as *(:value 'oats :when 'daily)* instead. Also, note that the *user* slot is not inherited from transporter since it is not in the *cache* list of transporter.

The slot and facet-access functions are fast because they are pre-compiled. However, they can only be used if the programmer knows the class, slot and facet names at code-writing (or compile) time; otherwise, the more general functions such as **get-facet** and **set-facet** (described below) should be used.

Note that in order to retrieve a value using inheritance or demons, one can't use the simple slot-access functions defined by Def-frame; rather, one should use **get-facet** or **get-facet-demons.** Perhaps demon-checking should be compiled into the slot-access functions; however, this would take much extra space.

def-frame* <name> <cplist> <slots> The non-macro form of **def-frame** (i.e., evaluates all of its arguments). Note that <*slots*> is not a rest argument in this function.

3.2. Commands that Make Instances

In the following functions, where the argument <*frame*> occurs, either the name of a frame or a frame instance may be given.

make-frame <classname> <instance-name> . <slot-values> This is a general function for making instances of the given <*classname*>. <*instance-name*> is the name of the instance (optionally nil for no name), since **Parmenides** functions may take frame names as arguments. <*slot-values*> has the same syntax as <*slot-contents*> of **def-frame** except the slot names are keywords. **Make-frame** returns an abstract data structure which represents an instance of the frame, and provides the default slot values that are described by the frame class. For slots without facets, any lisp object may be given; for slots with facets, a facet-plist should be given as the value. All arguments are evaluated. For example, to make an instance of an *animal* defined by the previous **def-frame** example, we could write:

```
(make-frame ’animal ’a1
        :reproduces ’(:value usually-not))
```

This returns a new instance of animal, called a1, with a *reproduces* slot value of *usually-not*. Note that since we didn't specify the *consumes* slot, it gets the default value of yes. Note also that for each class <classname> defined, a make function called **make-<classname>** is defined; for example **make-animal** would be defined when the animal class is defined.

make-frame0 <classname> <instance-name> <slot-values> Same syntax and definition as **make-frame**, except <*slot-values*> is a list instead of a rest argument.

3.3. Data Retrieval Commands

get-facet <frame> <slotname> <facetname> This function returns multiple values. The first value returned is the filler found in the specified facet of the slot of the given frame. If the given slot is not cached in the frame, then it tries to inherit the value from the parents. It searches the parents in a left-to-right, depth-first fashion. If either the slot or the facet cannot be found, it returns NIL as the second value (it also returns NIL for the first value in this case); if it is found then it returns T as the second value.

It works on both frame classes and instances. When a class is given, *get-facet* returns the default value for that facet. <frame> may be either the name of a frame or a frame data structure.

get-value <frame> <slotname> Equivalent to: **get-facet <frame> <slotname> :value**. For example, *(get-value 'horse :consumes)* would return values 'oats and T.

get-immediate-facet <frame> <slotname> <facetname> The same as **get-facet** except only retrieves data from the given frame (i.e., doesn't perform inheritance). Thus it is faster than **get-facet**. Also returns two values. The second value is nil if the given slot is not local in the given frame.

get-immediate-value <frame> <slotname> Equivalent to: **get-immediate-facet <frame> <slotname> :value**.

get-value-demons <frame> <slotname> Equivalent to: **get-facet-demons <frame> <slotname> :value**. See section 5 for an explanation of demons.

get-slot <frame> <slotname> Returns the slot <slotname> for frame <frame>, attempting inheritance if the slot is not found locally. Like **get-facet**, it returns two values. The first value returned is the value found in the specified slot; the second value returned is T iff the specified slot could be found. The difference between **get-slot** and **get-value** is that **get-value** gets the value facet of the slot (therefore assuming that the slot has facets), whereas **get-slot** is normally used for slots without facets, since it returns whatever data structure is stored at the slot level (not the facet level).

get-immediate-slot <frame> <slotname> Like **get-slot** except doesn't try inheritance. Thus it is faster than **get-slot**.

get-generic-value <frame> <slotname> A cross between **get-facet** and **get-slot**. If <slotname> does not have facets in <frame>, then this is equivalent to **get-slot**. If <slotname> has facets, then it is equivalent to **get-value** (it return the value facet of the slot).

get-cslot <frame> <slotname> Returns the value of the class slot <slotname> for frame <frame>. Sample class slot names are *:propagate* and *:setable*.

get-facet-demons <frame> <slotname> <facetname> Like **get-facet**, except if it gets a NIL cached value, tries running first, the local if-needed demons, then the if-needed demons attached to the frame. Doesn't try inheritance.

To check that a slot is local, use **local-p**.

local-p <frame> <slotname> Returns T iff the slot named <*slotname*> is local in the given frame (i.e., if the *:depth* facet of the slot is equal to 0). Being local means that it must have gotten its value from a **set-facet**, **setf**, **make-frame** or a set function if it's an instance, or from a **set-facet**, **setf** or **def-frame** if it's a class. If the slot has no facets then it is defined to be local, although if it doesn't have a depth facet, then it is defined to not be local. To get a feel for the depth facet and localness, try defining some classes, subclasses and instances, then check the depth of various slots in them.

pa-class-of <frame> Given a frame, returns the name of the class that the frame is an instance of.

isas <class> Given the class, returns an ordered list of all the classnames that <*class*> is-a, including superclasses of superclasses. For example, *(isas 'horse)* would return (animal transporter). If animal was is-a living-thing, then *(isas 'horse)* would return (animal transporter living-thing).

immediate-isas <class> Like **isas**, but only returns the immediate ancestors (i.e., the value of the *is-a* c-slot). For example, if animal was is-a living-thing, then *(immediate-isas 'horse)* would return (animal transporter).

inverse-isas <class> Returns a list of class names which are the immediate is-a of the given <*class*>. For example, both (inverse-isas 'animal) and (inverse-isas 'transporter) would return the singleton (horse). Note that inverse-isas only returns immediate inverse-isas, in contrast to the way isas returns *all* isas. This is because **inverse-isas** is used for marker propagation, where it is desired to have only the immediate ancestors of the given class. However, it is often useful to know if a given class is is-a another class, even if indirectly. This function returns NIL if the argument is not a class, because only classes have inverse-isas.

instances-of <class> Returns a list of the instances of the given frame class, including the instance representing the class.

set-instances-of <class> <instances> Sets the instances of the given class. This function should be used with care.

instance-names-of <class> Returns a list of the names of the instances of the given frame class. Doesn't include any instances which don't have names.

With-All-Subinstances-Of <class> <lambda> *Example: (with-all-subinstances-of 'relation #'(lambda (instance) (pp-frame instance)))*

This macro can be used to perform some function on instances of a class, as well as the instances of all the classes under the given class in the hierarchy. The given <lambda> must be a lambda which takes one argument, the instance. This macro runs that lambda on every instance of <class> and the instances of every class which is-a <class>, including the instance representing classes. To filter out instances representing classes, use **classp**, below.

get-slot-names <frame> Returns a list containing the names of all the local (cached) slots of the given frame. Note that slot names are generally referred to as keywords, and are always stored internally as keywords.

do-facets (<facet-name> <facet-val> <slot> <frame>) &rest expressions This macro executes the given expressions for each facet in the given <slot> and <frame>. The name of the facet (which will always be a keyword) is bound to <facet-name> and the value is bound to <facet-val>. For example:

```
(do-facets (name val :consumes 'horse)
  (format T "~S: ~S~%" name val))
```

would print the name and value of each facet in the *:consumes* slot of the given *horse*.

Note that this function only works for faceted slots; also it is much more efficient than calling **get-facet** on each facet.

3.4. Data Storage Commands

set-facet <frame> <slotname> <facetname> <newvalue> Sets the facet of the slot of the given frame (class or instance) to the given <*newvalue*>. If it's a class, then the value will be propagated down to all sub-classes and instances. Of course set-facet will not work (and will cause an error) if one tries to set a facet of a slot which doesn't have facets.

set-value <frame> <slotname> <newvalue> Equivalent to: **(set-facet <frame> <slotname> :value <newvalue>)**.

set-slot <frame> <slotname> <newvalue> Sets the value of the slot of the given frame (class or instance) to the given <*newvalue*>. Does not propagate to the instances or sub-classes or fire demons, since there are no facets on the slot to tell Parmenides whether it is a local value or what the demons are.

set-cslot <class> <cslotname> <newvalue> Sets the value of the class slot of the given frame class to the given <*newvalue*>. Note that unlike **set-slot**, this function only works with classes. Does not propagate to the instances or sub-classes.

set-facet-demons <frame> <slotname> <facetname> <newvalue> Same as set-facet, except also fires any pre-if-set and post-if-set demons it finds attached to the slot or frame class.

set-value-demons <frame> <slotname> <newvalue> Equivalent to: **(set-facet-demons <frame> <slotname> :value <newvalue>)**.

add-to-facet <frame> <slotname> <facetname> <filler> Adds the given filler to the list of values stored in the position in <*frame*> specified by <*slotname*> and <*facetname*>.

add-to-value <frame> <slotname> <filler> Adds the given filler to the list of values stored in the *value* facet of <*slotname*> in <*frame*>.

add-to-slot <frame> <slotname> <filler> Adds the given filler to the list of values under <*slotname*> in <*frame*>.

add-to-cslot <class> <cslotname> <filler> Adds the given filler to the list of values under class-slot <*cslotname*> in class <*class*>. Note that the first argument must be a frame class.

add-to-facet-demons <frame> <slotname> <facetname> <filler> Adds the given filler to the list of values stored in the position in <*frame*> specified by <*slotname*> and <*facetname*>. Since **add-to-facet-demons** uses **set-facet-demons**, it runs the same demons that would be run by **set-facet-demons**.

3.5. Predicates

framep <frame> Returns T iff <frame> is a valid frame class or instance.

frame-instance-p <frame> Returns T iff <frame> is an instance of a frame.

classp <frame> Returns T iff <frame> is a frame class. Note that the default instance representing classes are considered both classes and instances.

slotp <frame> <slot> Returns T iff <slot> is a slot in <frame>.

facetedp <class> <slot-name> Returns T if the given slot in the given class is faceted, NIL otherwise.

isa-p <classname1> <classname2> Returns T iff *<classname1>* is a sub-class of *<classname2>*, i.e., *<classname1>* is under *<classname2>* in the is-a hierarchy, or if *<classname1>* is the same as *<classname2>*.

isa-instance <frame> <classname> Returns T iff the given frame is an instance of the class named *<classname>*, or any of its superclasses. **isa-instance** could be defined as:

```
(member <classname> (isas (pa-class-of <frame>)))
```

isa-or-instance <frame> <classname> A combination of **isa-p** and **isa-instance**. Returns T iff *<frame>* is a subclass of *<classname>* or is an instance of *<classname>* or any of its superclasses.

3.6. Miscellaneous Commands

frame <framename> Given the name of a frame, this returns the actual frame. Frame classes are generally referred to by name, whereas instances can be referred to by either name or actual frame. When the name of a frame class is given to **frame**, the default frame instance is returned.

pp-frame <frame> &key :stream :all-slots Pretty-prints the given *<frame>*, which may be either an instance or a class. The keyword *:stream* defaults to the standard output (e.g., the terminal). *:all-slots* has a default value of T, which means that all slots are printed (but see **save-frame**).

save-frame <frame> &key :stream :all-slots Writes the given *<frame>*, in Parmenides-readable form, to the specified stream, or to the standard output if no stream argument is supplied. *:all-slots* has a default value of NIL which means that only those slots which differ from the corresponding parent's slots are printed.

copy-frame <frame> Returns a copy of the given frame. The data structures which are the values of the facets are not copied. Note that this function views <frame> as a frame instance, and doesn't copy any of the class information. This is because it isn't possible for two different classes to have the same instances, since instances are instances of only one class.

remove-frame <frame> Removes the given frame (instance or class). If *frame* is an instance, then it will be un-linked from its class (i.e., the result of **instances-of <class-of-frame>** will not include *<frame>*). It *frame* is a class, then the class will be removed from the list of inverse-isas which each of its parents contain, and each of its children classes will no longer have it as their parent. All other data structures pertaining to that class will be deleted, so that it will no longer be considered a frame or a class.

define-facet-getter <classname> <slots> <facets>
Parmenides usually defines access functions for only the *value* facet. To cause **Parmenides** to define access functions for facets other than *value*, use this macro. For example, executing: *(define-facet-getter house :age :depth)* allows one to retrieve the *depth* of the *age* of a house quickly by typing: *(house-age.depth <house>)*. <slots> and <facets> may be either lists or atoms.

define-facet-setter <classname> <slots> <facets>
Analagous to **define-facet-getter**. Defines a facet-setting function of the form: *(set-<class>-<slot>.<facet> frame value)*.

define-facet-accessors <classname> <slots> <facets>
Defines both a facet-getting and a facet-setting function for the specified facet(s).

add-slot <classname> <slot-name> <slot-contents> &key cache
example: (add-slot 'horse :rider '(value 'person)) Augments the given class by adding the slot <slot-name> with description *<slot-contents>* to the class. Only works for classes, since instances should

never have slots that their classes don't have. Propagates the change to all instances of the class. If *cache* is true, then adds the <*slot-name*> to the list of slots to cache, and propagates the added slot to all subclasses of <*classname*>.

add-cslot <classname> <slot-name> <slot-contents>

Example: (add-cslot 'horse :rider 'person) Augments the class slot of the given class by adding the slot <slot-name> with description <*slot-contents*> to the class's class-slot data structure. Only works for classes. Doesn't propagates the change to instances of the class.

4. Relations

Parmenides allows users to define their own relations, in order to allow links between frame instances and classes other than the standard *is-a* and *instance* links. Relations are defined as classes themselves, through **def-frame**. User-defined relations allow one to specify which slots to inherit through the relation, and how to combine values inherited from other frames. Relations may also have inverses, in which case certain bookkeeping is automatically performed by the system.

4.1. Relation Slots Recognized by the System

- *slots-inherited* Has a value facet since it can be a list of atoms. *Slots-inherited* should be a list of <slot-name> that are inherited from the parent frame. An extended syntax also allows the list (<slot-name> <combination-type>) to be used in the place of a <slot-name> in the list. This allows individual slots to have combination types which aren't the same as the *combination-type* for the relation class. The keyword :*ALL* is also recognized for the value of this slot.

- *combination-type* Facet-less since its value is an atom. The *combination-type* slot defines how values are combined with a frame from the parent frame. Currently its value can be either *:first*, *:append* or *:nconc*. *:First* means that the most local frame's value has precedence; *:append* and *:nconc* mean to append or nconc the values from the related frame to get the result.

- *has-inverses* T or NIL (facetless). If true, then the relation is defined to have inverses. Inverse relations are supported in a variety of ways in **Parmenides**. When a class which is a relation with inverses is defined, the class corresponding to its inverse is also defined. Also, when an instance-making function or a data-storage function such as **set-slot** or **set-facet** stores a value on a slot <*relation-slot*> which is a relation with inverses, the value <*range*> is assumed to be a frame. If it is, then <*range*> should have a slot <*inverse-relation-slot*> whose name is the same as the inverse relation corresponding to <*relation-slot*>. The <*range*> frame will automatically have the original frame added to the list of values stored in its <*inverse-relation-slot*>. When a class is defined to have a relation slot, and the value of that slot is another class (<range>), then the inverse relation slot corresponding to that slot will be added to the <range> frame, and the value will be the original frame. The examples below will make this more concrete.

- *inverse-names* Symbol (facetless). The name to be given to the inverse relation if the relation has inverses. Ignored if the relation does not have inverses. This is the name used to make the inverse relation class when a relation class is defined, and is also the name of the slot which is updated when a frame is stored in a relation slot. If not supplied, then the inverse name is constructed by concatenating "inverse-" to the beginning of the relation name.

4.2. Examples of Relations

For example, suppose the *part-of* relation were defined as follows:

```
(def-frame part-of (:is-a relation :propagate nil)
  :combination-type :FIRST
  :slots-inherited (:value '(location made-from))
  :has-inverses T
  :inverse-name contains)
```

Since *part-of* is defined to have inverses in this example, the system would automatically define the *contains* relation (if it hadn't already been defined) as:

```
(def-frame contains (:is-a relation :propagate nil)
  :has-inverses T
  :inverse-name part-of)
```

In order to define a frame which is *part-of* another, use *part-of* in the class slot plist. For example, to make *leg* a part-of *table*, first define the table frame, then type:

```
(def-frame leg (part-of table)
  :function (:value 'support))
```

Since the *part-of* relation was defined to inherit the slots *location* and *made-from*, the *leg* frame will receive these slots from *table*. Also, since *part-of* has inverses, and that inverse is named *contains*, the *table* frame will have *leg* added to its *contains* class slot. (If it didn't already have a contains class slot, then that slot will be added).

Note that the class(es) that a frame is a relation to need not be a list if it is only in relation to one frame.

Instances may also have relations with other instances or classes. Since slots can not be added to instances, instances may inherit only values from other frames. The relation is specified as an instance slot when defining the class. For example, in order to make different instances of legs part-of different instances of tables, the leg frame would be defined thusly:

```
(def-frame leg (part-of table)
  :part-of table
  :function (:value 'sit-on))
```

This would cause **Parmenides** to add the *contains* instance slot to the *table* frame if it didn't already exist. *Leg* would be added to the list of values of the *contains* slot of *table*.

To inherit values from table frames, when a leg instance is made, a table is given as the value of the *part-of* slot:

```
(make-leg 'leg1
    :part-of 'table1)
```

leg1 would be added to the list of values in the *contains* slot of *table1*.

Currently the *is-a* relation is special, so if the user defines the is-a relation, it will not affect inheritance through *is-a* specified as a class slot.

5. Demons

Instead of having only one type of demon associated with access functions, **Parmenides** provides two different types. Sometimes it is useful to perform some action before a slot is set (for example, to delete the old slot value from some data base), and then perform another action afterwards. Similarly, sometimes it is useful to perform some action both before a retrieve request (e.g., to compute the value), and after the request (e.g., to update a data base).

Demons (corresponding to methods in object-based languages) are single s-expressions which are evaluated when certain facets are accessed (set or retrieved). Since demons can be stored as both class slots and slot facets, they can be associated with either a class or a slot, or both. When a demon is evaluated, the following lisp variables are locally set (although *specially* bound - see p. 157 of **Common Lisp, The Language**, by Guy L. Steele, Jr.):

- framename -- the name of the frame that is being accessed
- slotname -- the name of the slot being accessed
- facetname -- the name of the facet being accessed
- frame -- the actual frame data structure (not the name) being accessed
- snum -- slot number of the slot
- facetnum -- facet number
- newval -- value being stored (only for demons which set values)

The following demons types are recognized by **Parmenides**.

- **if-needed**. When **get-facet-demons** is used to retrieve a facet, the following algorithm is used:
    1. Try to get an immediate value.
    2. If there is no immediate value (i.e., if the value stored there is NIL), then try to invoke the if-needed demon stored on that slot.
    3. If there is no demon on that slot, then try calling a demon stored on the class.
- **pre-if-set**. When **set-facet-demons** is used to set the value of a facet, before the frame is changed, any pre-if-set demons stored on either the slot or the class will be evaluated.
- **post-if-set**. When **set-facet-demons** is used to set the value of a facet, after the frame is changed, any post-if-set demons stored on either the slot or the class will be evaluated.

In summary, when the **set-facet-demons** command is issued, the following sequence occurs:

1. Execute any **pre-if-set** demons associated with the slot.
2. Execute any **pre-if-set** demons associated with the class.
3. Set the facet to the new value.
4. Execute any **post-if-set** demons associated with the slot.
5. Execute any **post-if-set** demons associated with the class.

6. Multiple Language Messages

Parmenides can produce its messages in a number of languages. Although it currently supports only English and Spanish, it is easy to add new languages through message files. When Parmenides is loaded, it loads a message file for the appropriate language. When installing Parmenides, the definition of the Parmenides directory must be set - change the variable *PA-PATHNAME*, near the top of parmenides.lisp. The language can also be set by changing the variable *LANGUAGE*. The current language can be changed by calling **define-language** with the name of the appropriate language. "eng" defines English, and "esp" defines Spanish.

7. Compiling User Frame Files

As mentioned previously, user frame files (i.e., files containing Def-frame) may be compiled when a set of frames is known to be stable. After that, the compiled frame file may be loaded in without having to load in the source frame file. (compiled file extensions for spice lisp are .fasl; for the Symbolics they are .bin). This results in considerable loading and executing speed up, since in the act of compiling the files, the Def-frame command is executed and all the access and make functions are defined. However, the Parmenides program should be loaded in before the frame files are either compiled or loaded, in order to get the definitions of the general Parmenides functions (such as get-facet).

8. Future enhancements

- Frame classes should be able to be declared to not have instances if they are only used for concepts. For example, one probably wouldn't make any instances of the animal class if one had sub-classes such as mammal and amphibian. This would save the time and storage space of creating a default instance and defining maker and access functions.
- Frames should be able to be both classes and instances. For example, the person class is an instance of the class class.
- In order to make class slots more complete, they should also be allowed on slots. If a slot had a meta-value, this would only be stored in the slots of the class, and not the instances.
- Ability to delete facets and slots.

9. Acquiring, Loading, Compiling and Using Parmenides

Parmenides is free to affiliates and members of the CMU community. It is available for a nominal fee to all others. To request a copy, send mail to pshell@ml.ri.cmu.edu or write to:

Peter Shell
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The Parmenides source file is /usr/pshell/parmenides/parmenides.lisp on the ML vax or Wiener RT, and works under any reasonable dialect of Common Lisp. Parmenides should be compiled before used. Parmenides must be loaded before it is compiled. In order to load the multi-lingual message files, the variable *PA-PATHNAME*, near the top of parmenides.lisp, must be defined. This variable defines the directory in which the Parmenides files are found. To run Parmenides under CMU Common lisp on an RT, load /../wiener/usr/pshell/parmenides/parmenides.fasl. Parmenides is put into a package called 'Parmenides' with nicknames 'Parm' and 'Pa'. After loading Parmenides, one should do: *(use-package 'parmenides)*.

This document is /../ml/usr/pshell/parmenides/parmenides.{mss, doc, ps}. This directory also contains a test file called prtest.lisp. After compiling Parmenides, load this file to ensure that Parmenides works okay on your system. If there are any questions or comments, please send mail to PShell@ml.ri.cmu.edu.

Table of Contents

1. Introduction 1
2. Propagation and Caching versus Inheritance 2
3. Parmenides Commands 2
    3.1. Class Definition Commands 3
        3.1.1. slot-descriptions 3
        3.1.2. cplist 3
        3.1.3. Discussion 4
        3.1.4. Examples of Def-frame 5
    3.2. Commands that Make Instances 6
    3.3. Data Retrieval Commands 6
    3.4. Data Storage Commands 8
    3.5. Predicates 8
    3.6. Miscellaneous Commands 9
4. Relations 10
    4.1. Relation Slots Recognized by the System 10
    4.2. Examples of Relations 11
5. Demons 11
6. Multiple Language Messages 12
7. Compiling User Frame Files 13
8. Future enhancements 13
9. Acquiring, Loading, Compiling and Using Parmenides 13