ObjTalk84
Reference Manual

Andreas C. Lemke

Technical Report CU-CS-291-85

June 19, 1985
Dept. of Computer Science
University of Colorado, Boulder

1. Introduction

Objtalk84 is an object-oriented extension of UNIX 4.2bsd FranzLisp. It was developed by Christian Rathke at the University of Stuttgart, Germany.

The general control paradigm of ObjTalk is message passing. A message is sent to an object using the lisp macro `ask`.

1.1. Metanotation

For syntax descriptions we use the following metanotation:

`<a> ::= <b> <c>... <d>`

Angle brackets denote non terminal symbols. The symbol "..." denotes "any number of". `<a>` is defined as `<b>` followed by an arbitrary number of `<c>`s (including 0) followed by `<d>`.

`<b> ::= SLOT`

Words completely in upper case denote any lisp symbols.

`<c> ::= object`

Words not completely in upper case are terminal symbols. So are any special characters other than "`::=`" and "...".

2. Lisp Functions

Slot access functions are described in section 5.1.2, page 13.

`(ask <object> <message>)`

returns: the result of the execution of the method of `<object>` the filter of which matches `<message>`.

use: to send a message to an object.

note: `ask` evaluates only those arguments which are preceded by a comma. The only exception is the first argument (`<object>`) which is evaluated if it is a symbol. If it has no value but a file name on its autoload property then this file is loaded. For details see Appendix I.

`(ask self <message>)`
`,!(<message>)`

note: the second form is equivalent to the first.

`(ask (<object> <class>) <message>)`

note: `<object>` behaves as if it were of the class `<class>`; i.e. methods are searched for in `<class>`.

see also: the `viewed-as:` message in section 3.3.

`(class-of 'so_object)`

returns: the class of `so_object`. If `so_object` is a symbol then the class of an object which is the value of `so_object` or which is autoloaded is returned.

`(mapask 'l_msg 'l_objects)`

side-effect: the message `l_msg` is sent to each object in `l_objects`.

`(objectp 'g_x)`

returns: nonnil iff `g_x` is an object.

`(object-of 'so_object)`

returns: `so_object`, if it is an object. If `so_object` is a symbol, it's value is returned or it is tried to autoload an object (load it from a file the name of which is the value of the `autoload` property).

`(ofclassp 'o_object 'o_class)`

returns: nonnil iff `o_object` is an instance of `o_class` or one of its subclasses.

`(try-rules '(s_rulename... ))`

side-effect: the specified rules are activated. If `try-rules` is applied to nil all rules are activated.

3. Objects

Objects have two basic properties:

- A set of slots which can store values (any Lisp s-expressions).
- A set of methods which define the object's reaction when it receives a message.

In order to have easy access to objects you bind them to symbols (cf. the `make:` message, p. 10)

3.1. Definition of object

Common properties of all objects are defined in the class `object` which is superclass of all other classes. `object` is defined as follows:

```
(ask class renew: object
     (superc)
     (descr
       (pname
         (default (get_pname (concat "<some_" (get-slot (cadr self) pname) ">")))))
     (methods
       (edit:    => sysedit)
       (eval: ,?<expr>  => eval)
       (get:  ,*<msgs>  =>=> getmultiple)
       (init: ,*<msgs>  => sysiniti)
       (kill:    => syskill)
       (pp:      => syspp)
       (redefining-form:        => sysredefinei)
       (set:  ,*<msgs>  =>=> setmultiple)
       (show:    => sysshow)
       (super:   => syssuper)
       (viewed-as: ,?<class> ,*<msg>    => sysviewed)))
```

3.2. Slots of Objects

pname = st_name

description: contains a symbol or string which is used to print the object.

3.3. Methods of object

`object` has methods to interpret the following messages. Since `object` is superclass of any class, the messages are understood by any object. If you want to make any object understand a new message then add an appropriate method to `object`.

<message> ::= edit:

side-effect: calls the lisp editor on the object's redefining form. If you don't leave the editor regularly the object may be in an inconsistent state.

see also: `redefining-form:` message.

<message> ::= eval: <expr>

returns: the value of `<expr>` evaluated in the binding context of the object (e.g.[1] `self` will be bound to the object, see section 6.4).

use: normally you do not want to use this. Define a method which does what `<expr>` would do. A reason is that `<expr>` will not be compiled and you will therefore run into problems with unspecial variables.

<message> ::= get: SLOT...

returns: a list of the fillers of the slots.

[1] exempli gratia

<message> ::= init: <init-messages>
<init-messages> ::= (<message>)...

side-effect: <init-messages> are sent to the object.

note: this method will be called after creation of the object. Normally you do not send this message by yourself.

use: to initialize slots and to send initial messages. This method is often extended in subclasses.

see also: `make:` and `instantiate:` methods of `class`.

<message> ::= kill:

side-effect: the object is deleted.

note: this method is often extended to kill dependant objects or to remove links from other objects.

<message> ::= pp:

side-effect: the object's redefining form will be prettyprinted.

see also: `redefining-form:` message.

<message> ::= redefining-form:

returns: an s-expression which when evaluated redefines the object.

see also: `pp:` and `edit:` messages.

<message> ::= set: (<message>)...

returns: a list of the values of the messages.

side-effect: the messages are sent to the object.

use: to set multiple slots.

<message> ::= show:

returns: the filler of the `pname` slot.

use: this is used by the printmacro for objects. If you prefer a different printed representation, define your own `show:` method.

<message> ::= super:

returns: the object's class

<message> ::= viewed-as: <class> <message>

note: `<message>` is sent to the object but method searching begins at `<class>`.

use: to select a certain perspective (i.e. to direct method search) if the class of the object has multiple superclasses or extended methods.

4. Classes

Every object is instance of a class. The class defines the properties and behaviour of its instances. Classes are themselves objects and therefore instances of a class (usually of the class `class`).

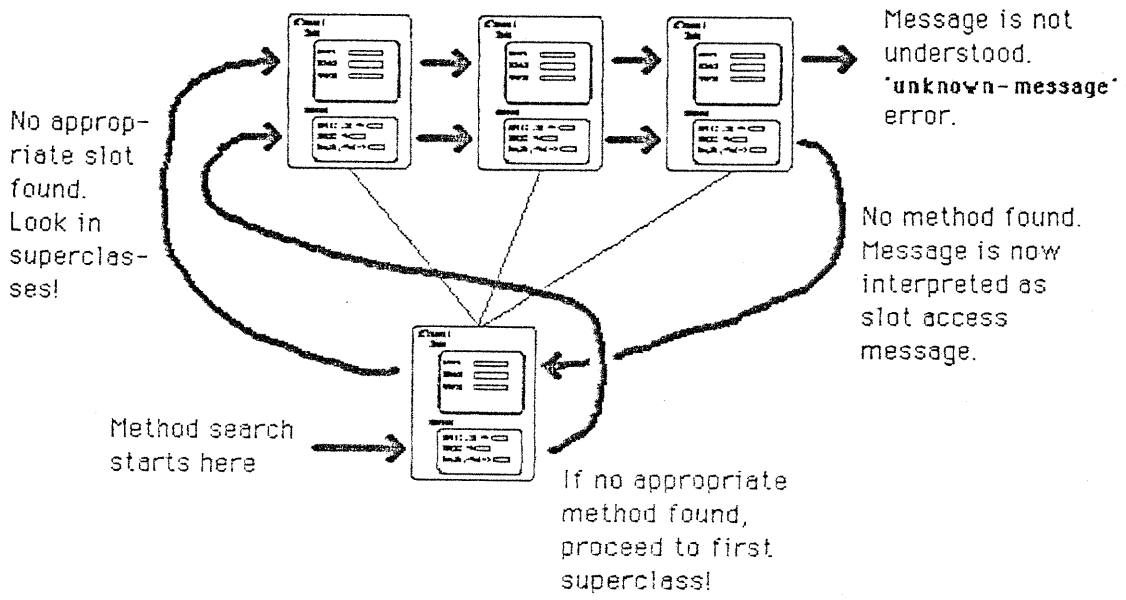

Figure 4-1: Method Search

4.1. Interpretation of Messages

The behaviour of an object is its reaction to received messages. The methods and slots of its class and its tree of superclasses define this reaction (see figure 4-1).

- The methods of the object's class are searched for a method with a filter matching the message (see section 6.1) in left to right order. If found, the body of the method is executed (see section 6.4).
- If no method is found, search proceeds in the class's superclasses again in left to right and depth first order.
- If no matching method is found at all, the message is interpreted as a slot access message (see section 5.1.1).

4.2. Overview of Predefined Classes

There are a set of predefined classes and objects which determine the structure of ObjTalk itself. They are explained in more detail below.

4.2.1. Sub- and Superclasses

Figure 4-2 describes the hierarchy of the predefined classes. For example, `Action` is a subclass of `object` and `Method` is the superclass of `Extended-Method`.

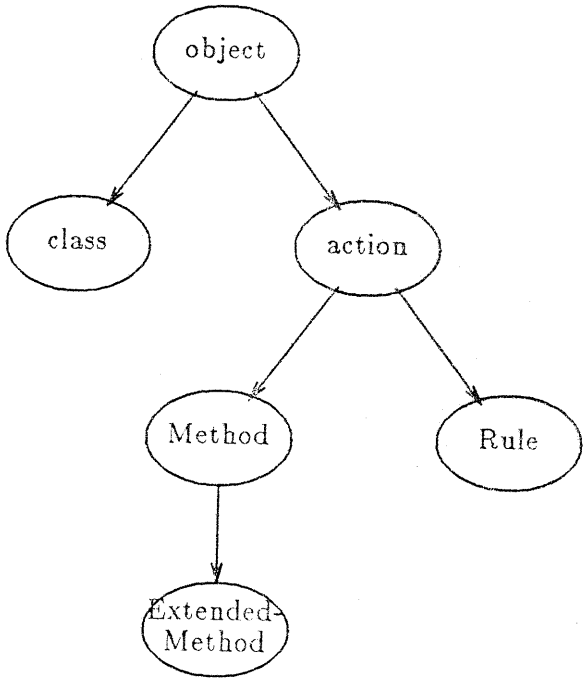

Figure 4-2: Sub- and Superclasses

4.2.2. Classes and Instances

Figure 4-3 describes the class - instance relationship of the predefined objects. For example, `Action` is an instance of `class` and `set-form-rule` is an instance of `Rule`.

4.3. class

Every class is an instance of the special class `class`. Since `class` can have instances it is itself a class and therefore an instance of itself.

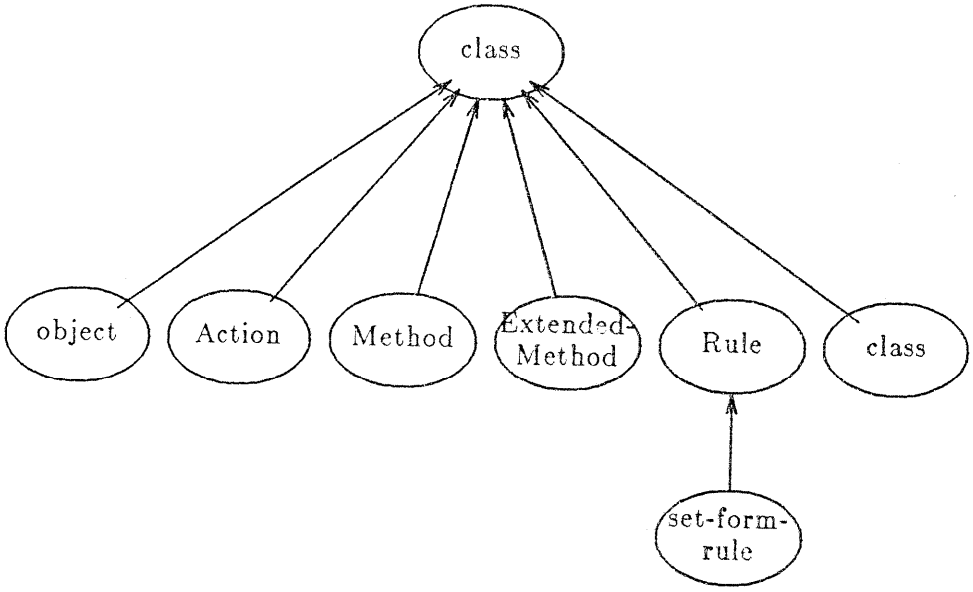

Figure 4-3: Classes and their Instances

4.3.1. Definition of class

The following could be a definiton of `class`:[2]

```
(ask class renew: class
     (methods
       (addmethod: ,?<name>:atom ,*<method-rest>        => sysaddmethodc)
       (addslot: ,?<newslot>:atom ,*<newdescr>  => sysaddslot)
       (compile:        => syscompile)
       (delmethod: ,?<oldslot>:atom     => sysdelmethodc)
       (delslot: ,?<oldslot>:atom       => sysdelslot)
       (describe: ,?<oldslot>:atom      => sys??c)
       (init: ,*<msgs>  =>=> sysinitc)
       (instantiate: ,*<msgs>   => sysinstwith)
       (kill:   =>=> syskillc)
       (make: ,?<obj>:atom      => sysmake)
       (make: ,?<obj>:atom with: ,*<msgs>       => sysmake)
       (new: ,?<obj>:atom ,*<alist>     => sysnew)
       (redefining-form:        => sysredefinec)
       (remake: ,?<obj> => sysremake)
       (remake: ,?<obj> with: ,*<msgs>  => sysremake)
       (renew: ,?<obj> ,*<alist>        => sysrenew)
       (replace: ,?<oldslot>:atom ,*<newdescr>  => sysrepl)
       (repmethod: ,?<name>:atom ,*<method-rest>        => sysrepmethodc)
       (repslot: ,?<newslot>:atom ,*<newdescr>  => sysrepslot)

       (slots:  => sysslots))
     (descr
```

[2] Since `class` cannot be created by sending a message to itself, class is created in a more basic way.

```
    (superc
      (default (list object))
      (if-set (instead
                (lambda (<superc>)
                  (mapcar (function object-of) <superc>)))
              (after
                (lambda (<superc>)
                  ,!(hierarchie = ,(mkhierarchie ,!superc))))))
    (descr
      (default nil)
      (if-set (instead mkdescrs))
      (if-added (instead mkdescr)))
    (methods
      (default nil)
      (if-set (instead mkmethods))
      (if-added (instead mkmethod))
      (if-changed
        (after
          (lambda (m)
            (in-all-subclasses-do ',!(all-methods = nil))))))
    (rules
      (default nil)
      (if-set (instead mkrules))
      (if-added
        (instead mkrule)
        (after
          (lambda (r)
            (in-all-subclasses-do '(establish-all-rules))))))
    (class-slots)
    (subclasses (default nil))
    (instances (default nil))
    (hierarchie)
    (proto)
    (all-descr (default nil))
    (all-methods (default nil))
    (all-rules (default nil))
    (init-list (default nil))
    (pattern (default 'make-form))
    (pname (default "<some_class>")))
  (superc object))
```

4.3.2. Slots of Classes

`superc = (<class> ...)`
description: contains a list of the class's superclasses.

`subclasses = (<subclass> ...)`
description: contains a list of the classes which have this class in their `superc` slot.

`instances = (<instance> ...)`
description: contains a list of the class's (immediate) instances. Instances of subclasses are not contained.

4.3.3. Methods of class

`class` provides methods to interpret the following messages. So, any class (being an instance of `class`) understands them.

<message> ::= addmethod: <method-specification>

side-effect: adds a new method to the left of the existing methods.

note: if the new method is an extended method, a preexisting method with the same key is not deleted and remains accessible. The method is accessible for already existing instances.

<message> ::= addslot: SLOT <slot-description>...

side-effect: adds a new slot.

note: the new slot will be propagated to existing instances. The slot will be unbound. It is an error for `SLOT` to already exist.

see-also: but see `default` and `if-needed` filler descriptions.

<message> ::= delmethod: KEY

side-effect: every method with key `KEY` is removed.

<message> ::= delslot: SLOT

side-effect: `SLOT` is removed.

note: Instances are not affected.

<message> ::= describe: SLOT

returns: a list of the form `(SLOT <slot-description>... )`

<message> ::= init: (<message>)...

side-effect: the messages are sent to the object.

note: this message will be automatically sent to an object after its creation.

<message> ::= instantiate: <init-messages>

returns: a new instance.

side-effect: after creation the message `"init: <init-messages>"` is sent to it. The `instances` slot of its class is updated to hold the new instance.

note: this is the basic way to create objects.

<message> ::= kill:

side-effect: the class, all instances and subclasses are killed.

<message> ::= make: INSTANCE with: <init-messages>
<message> ::= make: INSTANCE

note: this is equivalent to the `instantiate:` message. In addition the new instance is bound to the symbol `INSTANCE`.

<message> ::= new: NAME <class-description>...

returns: a new class.

side-effect: the new class is bound to NAME.

note: this is equivalent to the `make:` message but uses the syntax

```
(SLOT . <value>)
```

instead of:

```
(SLOT = <value>)
```

<message> ::= remake: INSTANCE with: <init-messages>
<message> ::= remake: INSTANCE

note: this is equivalent to the `make:` message, but if `INSTANCE` is bound to an existing instance the messages are sent to it and no new object is created.

use: to redefine an instance.

warning: this does not always do what you might expect: the object is not completely rebuilt. Instead the old object is modified.

<message> ::= renew: NAME <class-description>...

note: if `NAME` is bound to an existing class this class is redefined rather than created as a new class. Otherwise this is equivalent to the `new:` method.

warning: see `remake:` message.

<message> ::= replace: SLOT <filler-description>...

side-effect: the mentioned slot descriptions are substituted. If `<filler-description>` is a symbol (e.g. `default`), then this filler description is deleted. slot are deleted.

<message> ::= repmethod: <method-specification>

note: equivalent to the `addmethod:` message, but if a method with the same key already exists, it is removed first.

<message> ::= repslot: SLOT <filler-description>...

note: equivalent to the `addslot:` message but if `SLOT` already exists it is removed first.

<message> ::= slots:

returns: a list of all slot names of the class and its superclasses.

4.4. Class Descriptions

<class-description> ::= (superc SUPERCLASS...)
This description defines the superclasses of a class. A class inherits class descriptions from its superclasses in depth-first, left to right order. `object` is the default superclass.

<class-description> ::= (descr <slot-description>...)

This description defines the slots of instances of a class. Information can be stored into slots and read from slots by sending messages to an instance. See Chapter 5 for a detailed description of slots.

<class-description> ::= (methods (<method-specification>)...)
<method-specification> ::= <filter> <type> <body>

Methods define the reaction of instances of the class to messages sent to them. See Chapter 6 for a detailed description of methods.

<class-description> ::= (rules <rule-specification>...)
<rule-specification> ::= (RULENAME <premises> <affected-slots> <body>)

For each rule specification a rule is created as follows:

```
(ask Rule instantiate:
     (trigger-names: = <premises>)
     (output-names: = <affected-slots>)
     (body: = (<body>)))
```

See chapter 7.

5. Slots

Slots are used to store data in objects. Each slot may have at most one value called "filler".

5.1. Slot Access

Slots can be accessed either by sending messages to the object or by using certain basic access functions from within an object's method or rule bodies.

5.1.1. Slot Access Messages

The following behaviour of slots is only defined if there is no `if-accessed` filler description (see section III.1). Slot access messages can trigger demons (see section 5.2.2).

<message> ::= SLOT

returns: the filler of SLOT. If it doesn't have a filler, the message

```
SLOT needed:
```

is sent to the object and its result is returned instead of the filler of the slot.

<message> ::= SLOT = <value>

returns: <value>.

side-effect: SLOT is set to <value>.

<message> ::= SLOT forget:

returns: SLOT

side-effect: the filler of SLOT is removed.

<message> ::= SLOT add: <elem>

returns: <elem>

side-effect: <elem> is inserted in the list which is the filler of SLOT if it is not already there (checked with eq).

<message> ::= SLOT delete: <elem>
<message> ::= SLOT sub: <elem>

returns: <elem>

side-effect: removes <elem> from the list which is the filler of SLOT (uses eq).

<message> ::= SLOT describe:

returns: the slot description of SLOT.

see also: describe: method of class.

<message> ::= SLOT needed:

note: this message is sent automatically by ObjTalk if SLOT is read and it has no value.

returns: if the slot has a default filler description, the default is reevaluated, assigned to the slot and returned. If the slot has an if-needed or set-if-needed filler description, its value is returned. Otherwise a "slot not set:" error is signaled.

<message> ::= SLOT setp:

returns: nonnil iff SLOT has a filler.

<message> ::= SLOT <message>

note: <message> is forwarded to the object which is the filler of SLOT. If the filler of SLOT is no object, the break function is called and its result is used instead of the filler.

use: to send a message along a path in a network of objects.

5.1.2. Slot Access Functions

The following functions may be used from within objects (i.e. methods or rules).

`(setp SLOT)`
`(setp* 'SLOT)`
returns: non nil iff `SLOT` has a filler.

`,!SLOT`
`(slotvalue SLOT)`
`(sysget 'SLOT)`
returns: the filler of `SLOT`. If it has no filler, the message

```
SLOT needed:
```

is sent to the object and its result is returned instead of the filler of the slot.

see-also: Section II.1.

5.2. Slot Descriptions

`<slot-description> ::= (SLOT <filler-description>... )`
The following sections describe the types of possible filler descriptions.

5.2.1. Slot Initialization

5.2.1.1. The Procedure of Slot Initialization

1. If there are either

   `<filler-description> ::= (init <s-expression>)`

   or

   `<filler-description> ::= (init create) (class CLASS)`

   the slot is filled either with the value of `<s-expression>` or with an instance of `class`.

2. Then any `<init-messages>` are sent to the object.

3. Then if

   `<filler-description> ::= (modality required)`

   is provided and the slot is not set yet, an error message

   "slot `SLOT` is required"

   is printed.

4. Then if there is no `init` filler description and the slot has not been initialized by an init-message and either

```
<filler-description> ::= (default <s-expression>)
```

or

```
<filler-description> ::= (default create) (class CLASS)
```

is provided then the slot is filled either with the value of `<s-expression>` or with an instance of `CLASS`.

Slot initialization may be executed in parallel but for any one slot the above mentioned order is in general valid. An exception is when during initialization of one slot a demon reads the value of a not yet initialized slot. In this case, a default of the second slot will be evaluated prematurely.

5.2.1.2. Inheritance

Slots are inherited from the superclasses. If some superclass has a slot with the same name then its filler descriptions are inherited unless a subclass has a filler description of the same type (e.g. `default`). In the latter case, always the filler descriptions of the lowest subclass are valid. An inherited filler description can be disabled by specifying an empty filler description e.g.

```
<filler-description> ::= (default)
```

which means: no default.

5.2.2. Demons

5.2.2.1. Assignment Demons

The following filler descriptions are used to specify functions (demons) to be triggered if a slot gets a filler via the `"="` message:

```
<message> ::= SLOT = <s-expression>
```

5.2.2.1.1 if-set Filler Description

```
<filler-description> ::= (if-set <time-description>... )

<time-description> ::= (instead <demon>)
<demon> ::= <function-name>
<demon> ::= <function>
<demon> ::= <s-expr>...
```

the result of the demon being applied to the assigned s-expression will be the new filler. `instead` functions are inherited as follows: If `foo`, `sfoo`, `ssfoo` are the `instead` functions of the same slot of a class, its superclass and the superclass of its superclass, the new filler will be:

```
(ssfoo (sfoo (foo <s-expression>)))
```

If there are more than one `instead` functions at one level, the rightmost one will be applied first, the second rightmost will be applied to the result of the rightmost etc..

```
<time-description> ::= (before <demon>)
<time-description> ::= (after <demon>)
```

Before (after) the slot is assigned a value, the demon is applied to the new filler[3].

5.2.2.1.2 if-changed Filler Description

```
<filler-description> ::= (if-changed <time-description-1>... )
```

```
<time-description-1> ::= (before <demon>)
<time-description-1> ::= (after  <demon>)
```

This is similar to if-set demons. But these demons are only triggered if the slot has already a filler and `<s-expression>` is not `eq` to the old filler. `before`-demons are applied to the new filler, `after`-demons to the old filler.

5.2.2.1.3 trigger-rules Filler Description

```
<filler-description> ::= (trigger-rules RULENAME...)
```

Each rule is triggered in the specified sequence.

5.2.2.1.4 Invocation Sequence of Assignment Demons

When a slot is assigned a new value, the following demons (if present) are invoked in the following order. If there are more than one demon on the same level (in the same class) evaluation proceeds from left to right.

1. instead functions in leaf to root order of superclasses.
2. if-changed-before demons in leaf to root order.
3. if-set-before demons in leaf to root order.
4. new filler is assigned to slot.
5. if-set-after demons in root to leaf order.
6. rule demons in leaf to root order.
7. if-changed-after demons in root to leaf order.

5.2.2.2. Set Manipulation Demons

The following filler descriptions are used to specify functions (demons) to be triggered if a value is added to (deleted from) a set (represented as a list) which is the filler of a slot via the `add:` (`delete:`) message.

[3] after being transformed by `instead` functions.

```
<filler-description> ::= (if-added <time-description>... )
<filler-description> ::= (if-deleted <time-description-1>... )
```

Inheritance of set manipulation demons is analoguous to that of assignment demons.

5.2.2.3. if-forget Demons

```
<filler-description> ::= (if-forget <time-description-1>... )
```

This filler description is used to specify functions (demons) to be triggered if a filler is removed from a slot via the `forget:` message. The demon is triggered only if the slot actually had a filler before.

5.2.2.4. if-needed Demons

```
<filler-description> ::= (if-needed <s-expression>)
<filler-description> ::= (set-if-needed <s-expression>)
```

This filler description specifies an s-expression to be returned as the result of a slot read message

```
<message> ::= SLOT
```

if there is no filler and no default description. Note, the first form (`if-needed`) does not automatically assign the value of the expression to the slot. Therefore, In most cases you will prefer the `set-if-needed` form.

If several `(set-)if-needed` demons are specified then the first (in subclass to superclass and left to right order) is valid.

5.2.3. Filler Descriptions for Documentation Purposes

These descriptions are not interpreted by ObjTalk. The user may add other types of descriptions.

```
<filler-description> ::= (restrict PREDICATE... )
<filler-description> ::= (one-of (value <s-expression>)... )
<filler-description> ::= (all-of ???)
<filler-description> ::= (list-of ???)
```

5.2.4. Other Slot Descriptions

5.2.4.1. Constant Slots

```
<filler-description> ::= (quote <value>)
```

If this description is present the slot is a constant slot with the value `<value>`. Constant slots have always the same value for every instance.

6. Methods

Methods define the procedural behavior of objects. Methods consist of

- a filter which describes the set of messages which trigger the method and
- a body (sequence of s-expressions) which describes how the message is to be interpreted.

6.1. Method Filters

The filter of a method describes the set of messages which can invoke it.

`<filter> ::= KEY <filter-element>...`
A message matches a filter if it begins with KEY and each filter element matches.

```
<filter-element> ::= ATOM
<filter-element> ::= (*any* ATOM... )
```

matches the atom ATOM or any element of the list.

```
<filter-element> ::= ,?ELEMENTVARIABLE
<filter-element> ::= ,?ELEMENTVARIABLE:<pred>:<pred>...
```

matches any single s-expression if all of the predicates hold for it.

```
<filter-element> ::= ,*SEGMENTVARIABLE
<filter-element> ::= ,*SEGMENTVARIABLE:<pred>:<pred>...
```

matches any sequence of s-expressions (even the empty sequence) if all of the predicates hold for it.

```
<filter-element> ::= ,#CLASSVARIABLE:<class>
```

matches a single s-expression if it is an instance of class `<class>`.

6.2. Types of Methods

Methods can be of different types.

```
<type> ::= =>
```

Recursive methods. The method is of class Method. A message sent from within its body that matches its own filter invokes it recursively.

<type> ::= =>=>

The method is of class Extended-Method. Method search for messages with the same key sent from within its body[4] ignores this method. This kind of method is used to *extend* the behavior of existing, inherited methods by defining a sort of shell around them.

<filter> ::= (KEY <method-type-name>) <filter-element>...

Users can define their own method types (as subclasses of Method). When a class is created (e.g. using the new: message) for each method-specification an instance of its method class is created as follows:

```
(ask <method-type-name> instantiate:
     (pname = KEY)
     (filter: = (<filter>))
     (body: = (<body>)))
```

6.3. Method Classes

Methods are instances of the following classes:

```
(ask class renew: Action
     (rules (set-form !,! . set-form-rule))
     (descr (body: (trigger-rules set-form))
            (type:)
            (predicates: (trigger-rules set-form))
            (form:)))

(ask class renew: Method
     (superc Action)
     (descr (filter: (if-set (after Method:set-predicates)))
            (type: '=>))
     (methods (cond-form:       => Method:cond-form:1)
              (show:            => Method:show:1)
              (make-form:       => Method:mkform:)))

(ask class renew: Extended-Method
     (superc Method)
     (descr
       (type: '=>=>)
       (unique-name (default (gen-unique-name)))
       (predicates:
         (if-set
           (instead
             (lambda (p)
               (cons `(not (memq ',,!unique-name prohibit-calls)) p)))))
       (form:
         (if-set
           (instead
             (lambda (f)
               `(((lambda (prohibit-calls) ,@f)
                  (cons ',,!unique-name prohibit-calls)))))))))
```

[4] dynamic scope.

6.4. Method Bodies

`<body> ::= <s-expression>...`
`<body> ::= FUNCTIONNAME`

The body of a method is executed if the object receives a message which matches the filter (i.e. it interprets the message). If the body is a sequence of s-expressions then it is executed with the filter variables lambda bound to their actual values in the message. If the body is a function name then the function is applied to the values of the filter variables. In addition the following variables are available:

`self`	is bound to the object receiving the message
`sender`	is bound to the object sending the message. If the message is sent from the toplevel then `object` is the sender.
`msg`	is bound to the whole message.

7. Rules

Rules are similar to methods but they can only be invoked from within their objects. There are two ways to invoke a rule:

- if a slot is set which has a `trigger-rules` slot description.
- explicitly using the function `try-rules`. See page 2.

Rules consist of:

- a set of premises for its execution,
- a list of the slots which are affected by the body,
- a body.

7.1. Premises

`<premises> ::= (<premise>... )`
The premises are in the `trigger-names:` slot of the rule.

`<premise> ::= SLOT`

The premise is satisfied if the slot is set (has a filler).

`<premise> ::= <s-expression>`

The premise is satisfied if the expression evaluates to non nil.

7.2. Affected Slots

`<affected-slots> ::= (SLOT... )`
The list of affected slots is in the `output-names:` slot of the rule. This list has only documentary purposes.

7.3. Rule Body

`<body> ::= <s-expression>...`
The variable `rule` is lambda-bound to the object representing the rule. The body is in the `body:` slot of the rule.

7.4. Definition of Rule

```
(ask class renew: Rule
     (superc Action)
     (descr
       (type: 'Rule)
       (trigger-names:
         (default nil)
         (if-set
           (after
             (lambda (tn) ,!(predicates: = ,(mkpreds tn))))))
       (output-names: (default nil)))
     (methods (make-form: => Rule:mkform:)))
```

8. Constraints

Constraints[5] are used to define abstract relations like *sum*, *maximum*, etc.. It is then possible to declare such a constraint to hold for some slots of a specific object or alternatively of all instances of a class in general.

Once declared, constraints are automatically maintained by the ObjTalk system. Values which can be inferred are computed and assigned to their slots; contradictions are detected and presented to the user for resolution.

8.1. Definition of Constraints

Constraints are instances of the class `constraint` which is a subclass of `class`.

```
(ask class renew: constraint
     (superc class)
     (descr
       (patterns (default nil))
       ...)
     ...)
```

[5]The features described in this chapter are made available by loading the file `constr` in the ObjTalk directory.

The main difference between a regular class and a constraint is that the rules of a constraint are triggered automatically whenever a slot is affected.

```
<class-description> ::= (patterns <pattern-specification>... )
<pattern-specification> ::= (<pattern-element>... )
<pattern-element> ::= ,*SLOT
<pattern-element> ::= <atom>
```

Any of the specified patterns may be used in a class description to specify the slots of the class which correspond to the slots of the constraint (see Section 8.4).

ObjTalk supplies for instance the following constraint for the *sum* relation:

```
(ask constraint renew: adder
     (descr (a:) (b:) (c:))
     (rules
       (adder-rule-1 (b: c:) (a:)
         (and (numberp ,!b:) (numberp ,!c:)
              (ask self a: = ,(- ,!c: ,!b:))))
       (adder-rule-2 (a: c:) (b:)
         (and (numberp ,!a:) (numberp ,!c:)
              (ask self b: = ,(- ,!c: ,!a:))))
       (adder-rule-3 (a: b:) (c:)
         (and (numberp ,!a:) (numberp ,!b:)
              (ask self c: = ,(+ ,!a: ,!b:)))))
     (patterns
       (,*c: = ,*a: + ,*b:)
       (,*a: = ,*c: - ,*b:)))
```

8.2. Coreferences

Slots can be declared *coreferent*.

```
<message> ::= SLOT1 == <object> SLOT2
```

side-effect: the values of `SLOT1` and of `SLOT2` of `<object>` are declared to be always the same.

With the following class description, coreferences can be declared for all instances of a class.

```
<class-description> ::= (corefs <coref>... )
<coref> ::= (<path> == <path>)
<path> ::= SLOT...
```

The slots at the end of the two paths have always the same filler. If either of them is changed the other will also be changed.

8.3. Constraint Application to a Single Object

In order to declare a constraint for a specific object it is necessary to make an instance of the constraint and to make its slots *coreferent* to slots of the object using the `"=="` slot message (see Section 8.2).

Example:

```
(ask class new: package
     (descr (gross:) (net:) (wrap:)))

(let ((adder1 (ask adder instantiate:)))
  (ask package make: package1)

  (ask package1 net: == ,adder1 a:)
  (ask package1 wrap: == ,adder1 b:)
  (ask package1 gross: == ,adder1 c:))
```

8.4. Constraint Descriptions of Classes

An additional class description is used to declare constraints to hold for any instance of the class.

```
<class-description> ::= (constraints <constraint-clause>... )
<constraint-clause> ::= (CONSTRAINT (<path-or-constant>... ))
<path-or-constant> ::= SLOT...
<path-or-constant> ::= <atom>
```

The `<path-or-constant>` list should match one of the constraint's pattern specifications and thereby specify which slots (or slot paths) correspond to which slots of the constraint.

Example:

```
(ask class new: package
     (descr (gross:) (net:) (wrap:))
     (constraints (adder (gross: = net: + wrap:))))
```

8.5. Behaviour of Constraints

If the global variable `ctrace:var` (default value: `t`) is nonnil then constraints report their actions (e.g. inferences of slot values). If a slot, the value of which is inferred from other slots by means of a constraint, is assigned a different value then this situation is considered a contradiction. In this case the user is asked to retract one of the slot values which were used for the inference.

Example:

```
-> (ask package1 net: = 100)
awakening <some_adder> because its a: got the value 100
100
-> (ask package1 wrap: = 20)
awakening <some_adder> because its b: got the value 20
adder-rule-3 computed 120 for c: from (a: b:)
awakening <some_adder> because its c: got the value 120
20
-> (ask package1 gross: = 130)
contradiction when merging 120 of (package1 gross:) and 130
these are the premises that seem to be at fault:
|1.| (package1 net:) = 100
|2.| (package1 wrap:) = 20

choose one of these to retract and RETURN it.
Break choose culprit
<1>: (return 1)
retracting the premise (package1 net:)
removing 100 from (package1 net:)
```

```
removing 120 from ("<some_adder>" c:)
              because of ("<some_adder>" a:) == (package1 net:)
awakening <some_adder> because its c: lost its value
awakening <some_adder> because its a: lost its value
awakening <some_adder> because its c: got the value 130
adder-rule-1 computed 110 for a: from (b: c:)
awakening <some_adder> because its a: got the value 110
130
->
```

9. Error Handling

ObjTalk calls the following standard error handlers. They can be redefined by the user.

`(no-object-error 'g_form)`

called-when: a message is sent to a non object.

side-effect: prints an error message and enters a break loop.

returns: the value returned from the break loop (if it returns!).

note: ObjTalk continues with the returned value instead of `g_form`.

`(slot-not-set-error 's_slot)`

called-when: `s_slot` is read and has not been assigned a filler yet.

side-effect: prints an error message and enters a break loop. The returned value is assigned to the unbound slot.

returns: the value returned from the break loop (if it returns!).

note: ObjTalk continues with the returned value.

`(unknown-slot 's_slot)`

called-when: there is no slot with this name.

side-effect: prints an error message and enters a break loop.

returns: the value returned from the break loop (if it returns!).

note: ObjTalk continues with the returned slot name instead of `s_slot`.

`(unknown-message 'l_msg)`

called-when: there is no method with the required key.

side-effect: prints an error message and enters a break loop.

returns: the value returned from the break loop (if it returns!).

note: ObjTalk continues with the returned value instead of `l_msg`.

`(unknown-pattern 'o_self 'o_sender 'l_msg)`

called-when: there is a method with the required key, but no method with all of its filter elements matching the message.

side-effect: prints an error message and enters a break loop.

returns: the value returned from the break loop (if it returns!).

note: ObjTalk continues with the returned value instead of `l_msg`.

10. Compiling ObjTalk

Liszt compiles method and rule bodies. Also demons which are lambda expressions are compiled. The rest of the object cannot be compiled.

Put the following line at the beginning of your file:

```
(includef (concat objtalk:dir 'objincl))
```

Appendix I. The Backquote Macro

The backquote macro is an extended version of the standard Franz Lisp version. The `ask` macro evaluates its arguments like the backquote macro.

I.1. Types of Evaluation

Only expressions preceeded by a comma are evaluated:

`<x>`	insert `<x>` (do not evaluate).
`,<x>`	insert the value of `<x>`.
`,.<x>`	splice in the value of `<x>` (using nconc).
`,@<x>`	splice in a copy of the value of `<x>` (using append).

I.2. Nested asks and Backquotes

`ask`s and backquotes can be nested within each other:

```
'(ask ,x ,@',msg)
```

This means: `msg` is evaluated at the first evaluation time and a copy of its value is spliced in.

```
-> (setq msg '(move: 5 17))
(move: 5 17)

-> '(ask ,x ,@',msg)
(ask ,x move: 5 17)
```

Figure I-1 describes the evaluation scheme in nested `ask`s and backquotes. An `x` means that the expression is evaluated at that time and inserted. An `@` means that the expression is evaluated at that time and a

copy of the value is spliced in. The use of ,. for splicing without copying is analogous.

```
|- third last evaluation
  |- second last evaluation
    |- last evaluation

-------------------
    x       ,a
  x         ,',a
  x x       ,,a
x           ,,'',a
x   x       ,,',a
x x         ,',,a
x x x       ,,,a
    0       ,0a
  0         ,0',a
  0 x       ,,0a
  0 0       ,0,0a
  x 0       ,0,a
```

Figure I-1: Use of the Backquote Macro

Appendix II. Efficiency Considerations

II.1. Reading Slots

There are two ways to read a slot:

`,!Slot` and `,!(Slot)`

Both are functionally equivalent, but the first is more efficient since it does not use the message passing mechanism.

II.2. Fast Slot Access Functions

These functions access slots without using the message passing mechanism and therefore are faster. They do not trigger any demons.

`(get-slot '<object> SLOT)`

returns: the filler of `SLOT` of `<object>`. If it doesn't have a filler, the message

`SLOT needed:`

is sent to the object and its result is returned instead of the filler of the slot.

`(<- SLOT '<s-expression>)`
`(set-slot '<object> SLOT '<s-expression>)`

returns: `<s-expression>`.

side-effect: `<s-expression>` is filled into `SLOT`.

`(set-slot-unbound 'SLOT)`
side-effect: the filler of `SLOT` is removed.

Appendix III. Experimental Features

III.1. if-accessed Description

`<filler-description> ::= (if-accessed FUNCTION)`
The user may alter the behavior of slots using the `if-accessed` filler description. If this description is provided all other descriptions are not interpreted but are passed to `FUNCTION`. Whenever a message is sent to `SLOT`, `FUNCTION` is called as follows:

```
(FUNCTION 'SLOT '<slot-description> '<rest-message>)
```

`<rest-message>` is the message without the slot name.

III.2. Inverse Slot Relations

```
<filler-description> ::= (<inverse-relation> SLOT)
<inverse-relation> ::= 1:1
<inverse-relation> ::= 1:n
<inverse-relation> ::= n:1
<inverse-relation> ::= n:n
```

These slot descriptions are used to declare inverse (bidirectional) links between objects. Declarations have to be present in both participating objects.

Example:

If the slot `FATHER` has the description

```
(FATHER (1:n SONS))
```

and the slot `SONS` has the corresponding description

```
(SONS (n:1 FATHER))
```

then the slot `SONS` of object `x` will always contain a list of all the objects for which `x` is a filler of slot `FATHER`.

III.3. ObjTalk Files

The facilities described in this section are intended to serve the file handling for objects.[6] Networks of objects can be saved on files to be restored later.

[6] The features described in this section are made available by loading the file `file` in the ObjTalk directory.

```
(ask class renew: file
   (descr (objects: ...) (objects-not-to-save: ...)
   (methods (save: ,*objs => ...)
            (close: => ...)))
```

The class `file` describes an ObjTalk file. The `save:` message is used to associate objects to a file (the objects are stored in the `objects:` slot).

The `close:` message causes a file object to store its network of objects on a file with the name equal to its `pname`. The `close:` method stores the following objects:

- objects contained in the `objects:` slot,
- objects which are referenced by objects to be stored except:
  - they are bound to a variable and the user decides (upon request) not to store them or
  - they are contained in the `objects-not-to-save:` slot.

An ObjTalk file is loaded like any other lisp file (e.g. using the `load` function).

```
(ask class new: ...
   (methods
      (just-loaded: =>=> ... ,!(just-loaded:) ...)
      ...)
   ...)
```

The `just-loaded:` method can be extended to accomplish special initialization actions when the object is reloaded from an ObjTalk file.

```
after-load-actions = (<s-expr>... )
```

This variable holds a set of s-expressions which are evaluated after loading an ObjTalk file.

```
(world ['o_class])
```

returns: a list of `o_class`, its subclasses and their instances. Without parameter, it returns a list of all currently existing objects.

Appendix IV. Pitfalls

In the current implementation names of slots are lambda-bound within method and rule bodies. So, do not use variables with the same names as your slots.

In general it is a good strategy to use different names for slots, objects, filter variables etc..

Index

Table of Contents

List of Figures