ObjTalk Primer

Christian Rathke, Andreas C. Lemke

Technical Report CU-CS-290-85

July 17, 1985

Translated Version by

Vera M. Patten, Christopher P. Morel
Dept. of Computer Science
University of Colorado, Boulder

Table of Contents

List of Figures

Preface

This is an introductory primer to the ***ObjTalk***[1] language.

ObjTalk is implemented in ***Lisp***. Since ***ObjTalk*** can be viewed as an object-oriented extension of ***Lisp***, knowledge of ***Lisp*** is helpful.

A description of the language ***ObjTalk*** can give a good impression of its features. However, to learn the language, immediate practice on a terminal is recommended. The examples given in this primer can easily be followed and small ambiguities can be clarified through testing by the user.

First, a short introduction to object-oriented programming will be given, followed by an overview of the available ***ObjTalk*** language constructs. This section is only intended to give the reader an idea of the constructs, and therefore, attention to the syntax is not yet required. In further sections, the objects available in the system and the messages that are understood by them will be examined more closely and examples will be given. The last chapter will give a complete summary of the ***ObjTalk*** syntax.

ObjTalk is being developed as part of a research plan of the Institute for Computer Science at the University of Stuttgart.

Since ***ObjTalk*** is not yet fully developed, we welcome comments of any kind.

1. Overview

ObjTalk is a frame-based, object-oriented language for the representation of knowledge. The structural organization of the knowledge is achieved through a hierarchy of frames, and computation is object-oriented, i.e. represented through the sending and receiving of messages between objects.

1.1 Objects

Objects are active, structured knowledge entities for describing a situation. An example of an object could be the representation of a room. A room consists of walls, doors and usually pictures. Furthermore, it is a three dimensional structure, consists largely of air, and people can congregate in it and find electrical outlets. All of these facts could be represented in a "room object".

The parts of the room object can also be described through objects: pictures, chairs and walls could be objects as well as people, life forms, situtations, verbal expressions, attitudes, or thoughts.

Objects are active in the sense that they can exchange messages with each other. As a result, a room object can be "asked" about its attributes. An object that represents a lamp can be told through a message to plug itself into an outlet in the room. Consequences of these actions can be propogated through objects (the lamp is on, the room is brighter, electricity is being used).

---

[1] ObjTalk is an acronym for **obj**ect and **talk**.

1.2 Classes and Instances

There exists two object types: *classes* and *instances*. Classes are objects that describe other objects. For example, all people can be described by a *person* class. In ***ObjTalk*** the definition of a *person* class can be:

```
(ask class new: person
        (descr (last-name)
               (first-name)
               (telephone-number)))[2]

==>[3] person
```

With this expression, the class `person` is defined. Through the use of the keyword `ask`, a message is formulated. The receiver of the message is the object `class`. This portion

```
(descr (last-name)
       (first-name)
       (telephone-number))
```

defines the slots of the class `person`. In ***ObjTalk***, the slots describe the characteristics of the class. The class `person`, in this example, is described with the slots `last-name`, `first-name`, and `telephone-number`.

From a class, objects can be generated in which the slots have predetermined values. Such objects are called `instances`. For example, by sending a message to the class `person`, the instances `john` and `mary` could be created with:

```
(ask person make: john with:
        (last-name = smith)
        (first-name = john)
        (telephone-number = 4423342))

==> john


(ask person make: mary with:
        (last-name = miller)
        (first-name = mary)
        (telephone-number = 4444434))

==> mary
```

We have so far created three objects: `person`, `john`, and `mary`. `person` is the class of all people; `john` and `mary` are instances of `person`. The slots that we defined in the class `person` were initialized when `john` and `mary` were created.

[2] You are encouraged to immediately type in these expressions.

[3] `==>` means "results in".

Every instance belongs to a certain class. In the class, the slots are defined; in the instance the slots are filled.

1.3 Sending Messages

Objects in ***ObjTalk*** can be manipulated by sending them messages. Let us look at the two examples from above:

```
(ask class new: person
        (descr (last-name)
               (first-name)
               (telephone-number)))

==> person


(ask person make: john with:
        (last-name = smith)
        (first-name = john)
        (telephone-number = 4423342))

==> john
```

We sent messages to the objects `class` and `person` and in the process created new objects.

The sending of messages is achieved through the use of the keyword **ask**. Following **ask** is the object to which we want to send messages. The rest is the message itself. With the **new** message to `class`, new classes are created.

In the second case, we sent the **make** message to the class **person**. All classes can understand the **make** message which creates instances of that class.

The sending and receiving of messages is the defining characteristic of programming in ***ObjTalk***. In other words, "programming in ***ObjTalk***" means the formulation of messages. The sending of messages is indicated through the use of the keyword **ask**. Objects receive and interpret messages on the basis of their own specific attributes.

Next, we know that the objects `class` and `person` can understand messages for the production of new objects. Which messages, for example, `john` can understand depends completely on which class `john` belongs to. `john` is an instance of `person`. Therefore, `john` understands questions regarding the slots that were defined in the class **person**:

```
(ask john last-name)

 ==> smith

(ask john first-name)

 ==> john
```

Additionally, we can ask `john` to reassign his slots:

```
(ask john telephone-number = 4423333)

==> 4423333


(ask john telephone-number)

==> 4423333
```

The messages that can be understood by a given object depend entirely on which class the object belongs to. As shown above, an instance (`john`) can be asked about the values in its slots or can have the values of its slots reassigned through a message.

1.4 The Class Hierachy

We want to introduce the class `co-worker` that, in addition to all the slots contained in `person`, also has a project-slot in which the co-worker's research project membership can be stored. ***ObjTalk*** allows the previously existing class `person` to be used in the definition of `co-worker`:

```
(ask class new: co-worker
        (superc person)
        (descr (project-affiliation)))

==> co-worker
```

We will also define a new class `project`.

```
(ask class new: project
        (descr (boss)
               (employees)))

==> project
```

To generate an instance, we send the `make` message (as above) to a class:

```
(ask project make: ai)

==> ai[4]


(ask co-worker make: donald with:
        (last-name = anderson)
        (first-name = donald)
        (telephone-number = 4421234)
        (project-affiliation = ,ai[5]))

==> donald
```

What was accomplished with these statements? The object `class` was asked to generate a new class. The message, however, was extended to say:

> Make a new class `co-worker` with the slot `project-affiliation`; also, the class `person` should be a superclass of `co-worker`.

In this way, the class `co-worker` contains the slot `project-affiliation` and also inherits all the slots from `person`. Therefore, when we define `donald`, the slots of both classes can be filled.

To avoid redefining slots, they can be passed from one class to another. An analogy to the example would be the expression: "every `co-worker` is a `person`". In ***ObjTalk*** this means: "every instance of the class `co-worker`" inherits all the slots of the class `person`".

The inheriting mechanism plays a substantial role in ***ObjTalk***. All classes are embedded in an inheritance hierarchy. The root of this hierarchy (the class that passes its slots to all other classes) is named `object`. `object` is automatically established as a superclass when no other is specified (as in the definition of `person`).

With our examples, we have generated a class hierarchy (see Fig. 1-1).

---

[4]When you type in the previous expression, actually the newly created object is returned. Since objects are some complicated internal data structures, the **Lisp** system will only print their name. There is also a way to print out a more full fledged representation of an object.

[5]The comma is necessary here, because objects are implemented as some lisp data structures which are bound to symbols. E.g. the object that describes the project "ai" is the value of the **Lisp** symbol `ai`. So, in this example, `ai` has to be preceded by a comma to assign the object and not the symbol `ai` to the slot `project-affiliation`.

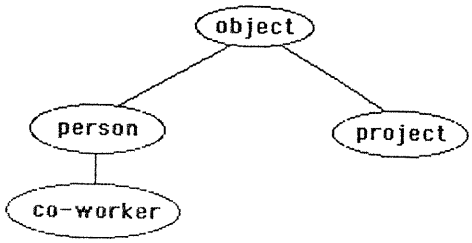

Figure 1-1: A Class Hierarchy

To complete the example, we will define a few more classes and instances:

```
(ask co-worker make: gerhard with:
        (last-name = fischer)
        (first-name = gerhard)
        (telephone-number = 4920000)
        (project-affiliation = ,ai))

==> gerhard


(ask ai boss = ,gerhard)

==> gerhard


(ask ai employees = (,donald ,gerhard))

==> (donald gerhard)


(ask donald project-affiliation)

==> ai


(ask ai boss)

==> gerhard
```

With the previously introduced object definitions, the class `project` and its instance `ai` were introduced, as well as a new instance of `co-worker` (`gerhard`). Furthermore, messages were used to ask `donald` for the value of his `project-affiliation` slot and to ask `ai` the value of its `boss` slot.

1.5 Methods

Objects can understand messages that inquire about the status of their slots. For example, we have seen above that the objects `person` and `class` can understand messages for the production of other objects (the `new` and `make` messages). Such messages, which are independent of slots, are understood by objects on the basis of methods.

A method is a procedure that is executed when an object receives a certain message. Methods are composed of two parts:

- a recognition portion, that is used in the understanding of the message
- an executable portion, that "computes" the "answer" to the message.

In this example, we will specify a method `supervisor`. The definition of `co-worker` would then have to be as follows:

```
(ask class new: co-worker
        (superc person)
        (descr (project-affiliation))
        (methods
          (supervisor => (ask ,(ask self project-affiliation) boss))))

==> co-worker
```

Everything that appears before the arrow (`=>`) comprises the recognition portion (or "filter") of the method. In the example, all instances of `co-worker` can understand the message `supervisor`. After the arrow, the executable portion (or the body) of the method appears. In this portion, the value of the slot `project-affiliation`, which has to be an instance of the class `project` in this example is asked about its slot `boss`. `self` always refers to the indicated instance (which in this case is `co-worker`, either `gerhard` or `donald`). A comma before an expression causes its evaluation. The two messages in the executable portion of the method are predefined for reading the slots `project-affiliation` or `boss`.

Since `donald` and `gerhard` are instances of `co-worker`, we can send them the message `supervisor`.

```
(ask donald supervisor)

 ==> gerhard

(ask gerhard supervisor)

 ==> gerhard
```

Methods can also have parameters as the following example for the definition of the second method of `co-worker` (`colleague`) demonstrates:

```
(ask class new: co-worker
        (superc person)
        (descr (project-affiliation))
        (methods
             (supervisor => (ask ,(ask self project-affiliation) boss))
             (colleague of ,?person ? =>
                    (and (memq person
                               (ask ,(ask self project-affiliation) employees))
                         (not (equal self person))))))

==> co-worker


(ask donald colleague of gerhard ?)

==> t

(ask gerhard colleague of donald ?)

==> t

(ask donald colleague of donald ?)

==> nil

(ask donald colleague of mary ?)

==> nil
```

The recognition portion of the method consists of four parts:

1. `colleague`
2. `of`
3. `,?person`
4. `?`

The parts 1, 2 and 4 are constant parts of the filter and must be literally the same in the corresponding portion of the message. `,?person` denotes a variable designated through the prefix "`,?`". `,?person` is a filter variable and one can substitute an arbitrary name in its place.

The filter variable is bound to a value in the executable portion of the method. In the example, we tested whether the portion bound to the variable (`person`) belongs to the `co-worker's` own project group and whether the variable is not equal to the referred to instance (of `co-worker`). The variable `self` is used as a temporary storage for the referred to instance. In the executable part of the method, many ***Lisp*** expressions can exist which will be evaluated in sequence. The recognition portion of a method can become quite complex. A special "pattern matcher" serves to compare the method filter with the arriving message.

1.6 The System Kernel

At system start-up, two predefined classes exist: `class` and `object`. These classes already have specified slots and methods. The slots and methods of the two objects determine the behavior of the entire system. The reason for this lies in the system architecture of ***ObjTalk*** which we will look at more closely now.

Previously, we split objects into the two categories: classes and instances. We said that classes will be embedded into a hierarchy, whereas instances belong to a specified class. These assertions have to be somewhat modified now: all objects are instances of a particular class. All classes are instances of the class `class`.

When we distinguish between classes and instances, then in reality we discern between objects which are instances of `class` and those which are not instances of `class`.

Since all classes are instances of the class `class`, they can themselves generate instances as well as have slots and methods. Classes are ordered in an inheritance hierarchy. For every class, we can define superclasses. Along the inheritance hierarchy, slots and methods of the superclasses are passed to their subclasses. This means that the later generated instances of a class have all the slots of the superclasses that lie in the inheritance path, and that they can react to all the messages that are defined as methods in the superclasses.

Now we turn to objects which are not instances of `class`. These objects also inherit their attributes from the classes (and superclasses) to which they belong. Often these objects and their classes are created by the user and comprise their special ***ObjTalk*** application. The inheritance mechanism is very powerful and can be used to create complicated, nested systems. However, the inheritance path of all the objects always ends with the designated class: `object`. This results in all objects inheriting the attributes that were defined in `object`.

What complicates the whole story is the fact that `object` is a superclass of `class`, and this is why all classes inherit attributes from `object`. As an instance, however, `object` behaves in accordance with the definitions of its class (also the class `class`). `class` has as a superclass `object` and therefore the behavior of `object` also defines itself in `object`.

The entire structure of ***ObjTalk*** is not easy to understand. Therefore, the inheritance hierarchy with the previously introduced classes and instances are illustrated here in figure 1-2. Also, see chapter 7 for a further description.

This structure gives the ***ObjTalk*** programmer a high degree of control over the ***ObjTalk*** system itself. By overwriting and supplementing the predefined methods in `class` and `object`, one can modify the system to fit one's own special applications.

2. Sending Messages

As mentioned before, the basis of ***ObjTalk*** programming is the sending and receiving of messages. Since we find ourselves in a ***Lisp*** environment, this action is introduced through a function call:

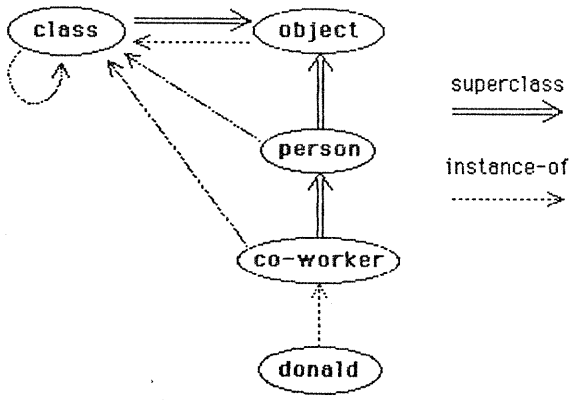

Figure 1-2: An Inheritance Hierarchy

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

`ask` is a ***Lisp*** function that does not evaluate its arguments unless the comma prefixes are used. These are analogous to the "`" (backquotes) in ***Lisp***.

The first argument of `ask` must be an ***ObjTalk*** object.[6] It is either a class or an instance. This object receives the message. The message consists of a symbol (the method name) and a sequence of elements of any length.

The object interprets messages with the methods that are known to it. The methods known to an object are those that were defined for the class of the object.

For a class, the following methods are defined:

- the methods of the class itself
- the methods that are defined for its superclasses.

Note the recursiveness of theses definitions. In this way, a class inherits the entire methods of all its superclasses, even those several levels away.

The interpretation of a message depends on the construction of the class hierarchy and on the available method filters. The classes are searched with a "depth first strategy". However, no class is examined until all of its subclasses were searched. In a class, the comparison of a message with the method filters decides which method is sufficient for the execution.

The body of the method is evaluated by the ***Lisp*** interpreter. The filter variable and the variables `self` and `sender` are bound in the method body. The result of this evaluation will be the value of the call to `ask`.

[6] It is also possible to use a symbol which is bound to an object. This is done throughout this primer.

3. Definition of a Class

In ***ObjTalk*** there are two pre-existing objects: `class` and `object`. These objects understand predefined messages of the system. `class` reacts to the `new` message with the creation of a class.

```
(ask class new: <class>
     (superc <superclass> ...)
     (descr <slot description> ...)
     (methods <method> ...)
     (corefs <coref> ...)
     (constraints <constraint> ...))
```

This message to `class` generates a class with the name `<class>`. At the same time, optional superclasses, slots, methods, coreferences, and constraints (see below) can be specified.

If these statements are missing, the `new` message has the simple form:

```
(ask class new: <class>)
```

```
(ask class new: life-form)

 ==> life-form
```

With this the class `life-form` is generated. It has no slots or methods. `object` automatically became its superclass.

3.1 The superc Aspect

```
(ask class new: <class>
     (superc <superclass> ...))
```

With the `superc` aspect, `<class>` is embedded into the inheritance hierarchy. The `superc` aspect defines the superclass(es) of `<class>`. If this aspect is missing, `object` is the superclass.

One can specify more than one superclass. This means that the newly defined slots and methods of a class can be inherited through more than one path.

```
(ask class new: person
     (superc life-form))

==> person

(ask class new: animal
     (superc life-form))

==> animal

(ask class new: fictitious-creature
     (superc person animal))

==> fictitious-creature
```

Through these messages to `class`, the classes `person`, `animal`, and `fictitious-creature` are produced. At the same time, an inheritance hierarchy is established along which the characteristics of the classes are passed (see Fig. 3-1).

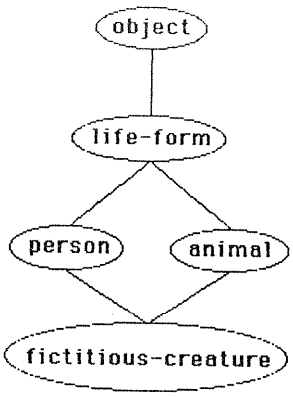

Figure 3-1: Another Inheritance hierarchy

3.2 The descr Aspect

```
(ask class new: <class>
       (descr <slot description> ...))
```

During class definition, the use of the `descr` aspect will define new slots of the class. In addition to the methods (see below), the slots are a portion of attributes that are passed to subclasses through the inheritance hierarchy.

The slot description consists of the name of the slot and a desired number of filler descriptions. Filler descriptions denote the later values of a slot during class instantiation.

```
<slot description> ::= (<slot> <filler description> ...)
```

3.2.1 Default

With the help of filler descriptions, slots can be given default values:

```
(default <s-expression>)
```

If during class instantiation the slot is not given a value then the slot takes on the value <s-expression>.

```
(ask class new: person
        (superc life-form)
        (descr (sex (default 'female))
               (children (default (ask self spouse children)))
               (spouse)))

==> person
```

If in the creation of an instance of the class `person` no value is given for `sex` or `children`, then female is the default value and the value of the children slot becomes that of the spouse's.

3.2.2 Coref

```
(coref <path> ...)
```

The value of a slot as specified above is always equal to the value of the other slots described by the paths. That is, if the value of this slot changes, then the value of the other slots change also and vice versa. These other slots are indicated by path descriptions:

```
<path> := (<slot> ...)
```

A path is a list of slots. It begins with a slot of `<class>` and ends with the slot whose value should be equal to the described slot. In the case that two slots of `<class>` itself should be equal, then the path has a length of one.

```
(ask class new: person
        (descr (spouse (class person))
               (children (coref (spouse children)))))

==> person

(ask person make: jim) ==> jim
(ask person make: jack) ==> jack
(ask person make: susan) ==> susan
(ask person make: mary) ==> mary
(ask person make: john) ==> john

(ask mary spouse = ,john) ==> john

(ask john spouse = ,mary) ==> mary

(ask mary children = (,jim ,jack ,susan)) ==> (jim jack susan)

(ask john children) ==> (jim jack susan)
```

In the example, it is stated that married people presumably have the same children and therefore, the `children` slots of the two people should be equal.

3.2.3 If-changed

```
(if-changed (before <function>)
            (instead <function>)
            (after <function>))
```

Before and after a value is assigned to a slot, unary functions (ones with one argument) can be called. The value with which the slot will be filled can later be modified. The functions have a parameter which the new slot value is bound to.

```
(ask class new: person
     (descr
       (birthday
         (if-changed
           (instead
             (lambda (date)
               (cond ((objectp date) date)
                     (t (generate-date date)))))
           (after (lambda (date) (set-age date (today))))))
       (age
         (if-changed
           (after (lambda (year) (set-birthyear year (today))))))))
```

In the above example, the connected slots `age` and `birthday` are kept consistant with each other through functions that are activated after the slots are filled. Before the birthday of the person is filled, the given birthdate is checked to determine if it is an object. If not, a date is generated and placed into the slot instead.

As one can see, not all if-changed options have to be specified.

3.2.4 If-forget

```
(if-forget (before <function>)
           (after <function>))
```

Before or after a slot value is deleted, the functions are executed.

3.2.5 If-needed

```
(if-needed <function>)
```

In case a slot is accessed which has not received a value yet, then the result of `<function>` is returned. The slot name is supplied as the parameter of `<function>`.

3.3 The methods Aspect

```
(ask class new: <class>
     (methods <method> ...))
```

Objects are provided with methods so that they can react to messages. We have already seen that for every slot the system automatically prepares a slot method so that we can communicate with the slot; i.e., we can read or fill the slot (see also the section on "Slot Access Functions").

Methods determine the reaction of objects to messages. Therefore, they play an important role in the communication of an ***ObjTalk*** system. The ***ObjTalk*** system kernel itself depends on the definitions of such methods. At this point, we are describing the `new` method that is understood by the object `class`. The method aspect itself is a portion of this method.

```
<method> ::= (<method name> <filter element> ... => <method body>)

     <filter element> ::= <constant> | <element variable> |
                          <segment variable> | <class variable>

               <element variable> ::= ,?<symbol> | ,?<symbol>:<predicate>

               <segment variable> ::= ,*<symbol> | ,?<symbol>:<predicate>

               <class variable> ::= ,#<symbol>:<class>

<method body> ::= <s-expression> ...
                  ::= <function>
```

Next, we want to provide the class `person` with some methods that we will examine more closely later.

```
(ask class new: person
        (superc life-form)
        (descr (age)
               (height)
               (birthday)
               (mother))
        (methods
           (how old => (ask self age))
           (what is your ,?slot => (ask self ,slot))
           (what is your ,*slots =>
                 (mapcar '(lambda (s) (ask self ,s))
                         slots)))
           (is ,#x:person your ,?slot =>
                 (equal (ask self ,slot)
                        x)))))

==> person
```

Methods consist of two parts: the filter, whose first element is the method name, and the method body. The method name can be an arbitrary ***Lisp*** symbol. Through this name the method is accessed. The entire filter is used to decide if a message can be recognized.

The filter consists of an arbitrary number of constants and variables. The individual filter elements are compared with the message. The type of the filter elements distinguishes the kinds of comparison procedures. As a side effect, the variables are bound to appropriate portions of the message so that one can access these bindings in the message body. A comparison of the message with the filter is successful when constants coincide and a binding between variables is possible.

Constants must match in the message and the filter.

```
filter: how old
example: (ask donald how old)

No other message can be recognized through this filter.
```

An element variable binds exactly one element of the message to the variable. Element variables are denoted with the prefix ",?" .

```
filter: what is your ,?slot
example: (ask donald what is your age)
binding: slot ==> age

example: (ask donald what is your (age height birthday))
binding: slot ==> (age height birthday)

not recognized messages would be:
                          your age
                          what is your age and height
```

A segment variable binds an arbitrary number of elements (including none) to the variable. It is designated with a ",*" .

```
filter: what is your ,*slots
example: (ask donald what is your)
binding: slots ==> nil

example: (ask donald what is your age)
binding: slots ==> (age)

example: (ask donald what is your age height birthday)
binding: slots ==> (age height birthday)

example: (ask donald what is your (age height birthday))
binding: slots ==> ((age height birthday))

not recognized messages: what's your height
                          what are your siblings
```

A class variable binds exactly one element of the message to the variable. This element has to be an instance of the class `<class>`. A class variable is introduced with ",#" .

```
Filter: is ,#x:person your ,?slot
Example: is john your father
Binding: x ==> john, slot ==> father
          (in case john is an instance of person)
```

The method body can consist of a series of s-expressions which are evaluated one after another like in a `progn`, or it can be a function. In the method body, all variables of the filter are bound. In addition, the variables `self` (bound to the referred object) and `sender` (bound to the sender of the message) are bound.

In the method body, the referred to objects can be accessed through their slots with the command `,!<slot>`.

```
(methods (who is your father => ,!father))
```

Instead of the arrow (=>), a double-arrow (=>=>) can be placed between the method filter and the method body to indicate an "extended method". In the body of such a method, a recursive call will not cause the application of the same method. In this way, existing methods (in the example, the slot access method) can be extended. In this case, a superclass of the method will be searched instead for the method.

```
(ask class new: number
     (descr (value)))

==> number


(ask class new: positive-number
     (superc number)
     (descr (value))
     (methods
         (value =>=> (concat '+ (ask self value)))))

==> positive-number


(ask positive-number make: pi with: (value = 3.141))

==> pi


(ask pi value)

==> +3.141
```

In "simple" methods (those that use the => symbol), the message `(ask self value)` would have led to a recursive behavior without a termination point.

3.4 The corefs Aspect

```
(ask class new: <class>
     (corefs <coref> ...))
```

With the help of the `corefs` aspect, portions of objects that can only be reached through the slot path of `<class>`, can be connected with one another. The slots located at the end of the path always have the same value.

```
<coref> ::= (<slot1> ... == <slot2> ...)
```

An example should clarify this:

The class `line` should have two slots which store the begin and end points. The class `triangle` is defined as

a class with three slots that has values that are instances of the class `line`. For a `triangle`, the begin and end points of its corresponding "coref" lines must be equal.

```
(ask class new: point
     (descr  (x-coord)
             (y-coord)))

==> point


(ask class new: line
     (descr  (begin-point)
             (end-point)))

==> line


(ask class new: triangle
     (descr  (line1)
             (line2)
             (line3)
     (corefs (line1 begin-point == line3 end-point)
             (line1 end-point == line2 begin-point)
             (line2 end-point == line3 begin-point)))

==> triangle
```

If one of the corner points of a triangle is modified through the shifting of a line point, then the corresponding point on the adjacent line is changed.

```
(ask triangle make: triangle1 with:
     (line1 = ,(ask line instantiate:
              (begin-point = ,(ask point make: p1 with:
                                   (x-coord = 0)
                                   (y-coord = 0)))))
     (line2 = ,(ask line instantiate:
              (begin-point = ,(ask point make: p2 with:
                                   (x-coord = 10)
                                   (y-coord = 0)))))
     (line3 = ,(ask line instantiate:
              (begin-point = ,(ask point make: p3 with:
                                   (x-coord = 5)
                                   (y-coord = 10))))))

==> triangle1


(ask triangle1 line1 begin-point)

==> p1


(ask triangle1 line3 end-point)

==> p1
```

```
(ask triangle1 line1 begin-point = ,(ask point make: p4 with:
                                        (x-coord = 1)
                                        (y-coord =0)))

==> p4


(ask triangle1 line3 end-point)

==> p4
```

3.5 The constraint Aspect

```
(ask class new: <class>
     (constraints <constraint> ...))
```

With the help of constraints, relationships among slots that are automatically updated by the system can be defined.

As an example, the description of a package in which the gross, net, and packaging weights can be described through a summation relationship:

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

==> package
```

In order to establish this relationship in the above specified form, the constraint `sum` must first be defined. For this purpose, the class `constraint` exists which is in the position to generate new constraints.

One creates a new constraint through sending the `new` message to the class `constraint`. This message has the following form:

```
(ask constraint new: <constraint>
             (superc <superclass> ...)
             (descr <slot description> ...)
             (methods <method> ...)
             (corefs <coref> ...)
             (constraints <constraint> ...)
             (rules <rule> ...)
             (patterns <pattern> ...))
```

As one can easily infer, this form of the `new` message is similar to the one of `class`. Except for two additional aspects: `rules` and `patterns`, the remaining aspects have the same meaning as previously defined.

3.5.1 The rules Aspect

The function behavior of a constraint depends significantly on the rules that are used to establish the relationship described through the constraint. These rules are specified in the `rules` aspect. A rule consists of four parts: a name, two slot lists, and the rule body:

```
<rule> ::= (<rule-name> <entry-slots> <output-slots> <rule-body>)
```

```
(ask constraint new: sum
  (descr (add-1)
         (add-2)
         (sum))

  (rules
       (r1 (add-1 add-2)
           (sum)
           (ask self sum = ,(+ ,!add-1 ,!add-2)))
       (r2 (add-1 sum)
           (add-2)
           (ask self add-2 = ,(- ,!sum ,!add-1)))
       (r3 (add-2 sum)
           (add-1)
           (ask self add-1 = ,(- ,!sum ,!add-2)))))

==> sum
```

With this, the functionality of the constraint `sum` is completely described. The first slot list after the `<rule name>` describes the slots that when modified will activate that rule. In our example, the rule r1 "triggers" when one of the two operands is filled with a new value.

The second slot list specifies the slots that will be modified by the rule. In the rule r1, this is the sum-slot `sum`.

The `<rule body>` consists, like a method, of a series of s-expressions that are evaluated in sequence.

With the above defined constraint `sum`, instances can now be generated and can be connected to instances of other classes where this constraint should hold.

In our specific case, this could look as follows:

```
(let ((my-sum (ask sum instantiate:)))
   (ask package make: present with: (net = 100) (wrap = 5))
   (ask present net == ,my-sum add-1)
   (ask present wrap == ,my-sum add-2)
   (ask present gross == ,my-sum sum)
   'present)

==> present


(ask present gross)

==> 105


(ask present wrap = 10)

==> 10


(ask present gross)

==> 110
```

Through the binding of the package slots with the slots of the sum constraint, we have achieved the effect that the relationships specified in sum will also apply to an instance of the class package. This is a form of "constraint application".

The above depicted procedure is however still too complicated. It is not desirable to write a large piece of code each time a package is instantiated. Furthermore, the relationship "gross = net + wrap" might preferably be specified during the definition of the class package.

For this we have the patterns aspect during the definition of a constraint and the constraints aspect which is possible to use during each class definition.

3.5.2 The patterns Aspect

The patterns aspect describes the relationship specified through a constraint in one or more expressions. When the definition of sum is extended to include this aspect, the definition looks like this:

```
(ask constraint new: sum
   (descr (add-1)
          (add-2)
          (total))
   (rules
     (r1  (add-1 add-2)
          (total)
          (ask self total = ,(+ ,!add-1 ,!add-2)))
     (r2  (add-1 total)
          (add-2)
          (ask self add-2 = ,(- ,!total ,!add-1)))
     (r3  (add-2 total)
          (add-1)
          (ask self add-1 = ,(- ,!total ,!add-2))))
   (patterns (,*total = ,*add-1 + ,*add-2)
             (,*add-1 = ,*total - ,*add-2)))


==> sum
```

After utilizing the `patterns` feature as above, one can now use this relationship under the `constraints` aspect when defining the class `package`:

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


==> package
```

The `constraints` aspect consists of several lists. The first element of every list is the name of a constraint (in the example: `sum`). The second element is an expression of slots and constants that have to correspond to one of the patterns of the constraint.

During instantiation of the class `package`, the above procedure is executed. That is, an instance of a constraint is created and corresponding slots are bound to each other so that "constraint maintenance" is achieved.

Here is an additional example:

```
(ask class new: earnings
     (descr (salary)
            (base-pay)
            (bonuses)
            (taxes)
            (net-pay))
     (constraints
        (sum (salary = base-pay + bonuses))
        (sum (net-pay = salary - taxes))))

==> earnings
```

Note that in this example of the `constraint` aspect, the same constraint (`sum`) is used with two different patterns.

4. Classes and their Methods

By sending the `new` message to the object `class`, classes are generated. These classes can now themselves understand messages, especially those messages that generate instances from them.

The messages that are understood by the classes are defined in `class`.

4.1 The make Message

```
(ask <class> make: <instance> with:
     (<slot> = <value>)
     ...)
```

With the `make` message new instances are created and provided with a name through which they can be addressed. At the same time, the different slots are filled with their (default) values. This assignment of values is not required.

```
(ask person make: donald with:
     (birthday = ,(ask date instantiate: (year = 1921)
                                         (month = december)
                                         (day = 22)))
     (father = ,frank)
     (mother = ,marie))

==> donald

(ask fictitious-creature make: donald-duck with: (father = ,disney))

==> donald-duck
```

4.2 The instantiate Message

```
(ask <class> instantiate:
    (<slot> = value)
    ...)
```

Through this message, instances without names are created. This message only makes sense for temporary objects, or when the created object is used as a value in the slot of another object (so that it can be referenced).

```
(ask person make: frank with:
     (birthday = ,(ask date instantiate:
                    (year = 1920)
                    (month = february)
                    (day = 29))))

==> frank
```

4.3 The addslot Message

```
(ask <class> addslot: <slot> <filler-description> ...)
```

The `addslot` message joins another slot to a class.

```
(ask person addslot: age (default 0))

==> age
```

4.4 The describe Message

With the `describe` message, the filler descriptions of a slot can be returned.

```
(ask <class> describe: <slot>)
```

```
(ask person describe: age)

==> (age (default 0))
```

4.5 The replace Message

```
(ask <class> replace: <slot> <filler-description>)
(ask <class> replace: <slot> <key>)

<keyword> ::= default coref
```

With the `replace` message, existing filler-descriptions can be replaced or deleted. To delete a filler-description, just give the keyword of the filler description.

```
(ask person replace: age default)

==> age
```

4.6 The delslot Message

```
(ask <class> delslot: <slot>)
```

`<slot>` is deleted.

```
(ask person delslot: age)

==> age
```

4.7 The addmethod Message

```
(ask <class> addmethod: <method-name> <filter> => <body>)
```

Finally, methods are added to a `<class>`. From this point on, all instances of `<class>` understand the new message.

```
(ask person addmethod: parents => (list (ask self mother)
                                        (ask self father)))

==> parents
```

4.8 The delmethod Message

```
(ask <class> delmethod: <method-name>)
```

The method with the name `<method-name>` is removed from the list of methods in `<class>`.

```
(ask person delmethod: parents)

==> parents
```

5. Instances and their Methods

Instances are produced by sending a `make` or `instantiate` message to classes. These instances understand all methods defined in their class. However, they can also react to messages that are defined in the superclasses of their class, since these methods are passed in the inheritance hierarchy. We have said that the class `object` is the root of the inheritance hierarchy. This means that the methods in `object` are understood by every instance.

The methods defined in `object` will now be described.

5.1 The init Message

```
(ask <instance> init:)
```

During instantiation of every instance, this message is sent to the instance. The corresponding method insures that DEFAULTS, etc., are assigned. The user has the ability to extend the `init` method in order to create an initializing method.

As an example, we want to extend the `init` method so that during every definition of an object a counter is incremented.

```
(ask object addmethod: init: =>=> (incr object-counter)
                                  (ask self init:))

==> init:
```

5.2 The viewed-as Message

```
(ask <instance> viewied-as: <class> <message>)
```

This message selects a specific perspective through which the referred to instance should be viewed. It is only meaningful when using multiple superclasses.

```
(ask class new: vehicle
     (methods (task => 'transportation)))

==> vehicle

(ask class new: toy
     (methods (task => 'play)))

==> toy

(ask class new: auto
     (superc vehicle toy))

==> auto

(ask auto make: fancy-car)

==> fancy-car

(ask fancy-car viewed-as: toy task)

==> play

(ask fancy-car viewed-as: vehicle task)

==> transportation
```

5.3 The kill Message

```
(ask <object> kill:)
```

`<object>` is deleted.

```
(ask kurt kill:)

==> kurt

(ask kurt age)

error: no object kurt
```

6. Slot Access Methods

For every slot of a class, there implicitly exists a slot access method that is activated by a message of the form:

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

In this access method, further branching is achieved on the basis of the **<message>**.

The slot access can also be made to a slot at the end of a slot path. The messages then have the form:

```
(ask <object> <slot1> <slot2> ... <slotn> <message>)
```

This is an abbreviated form for:

```
(ask ... ,(ask ,(ask <object> <slot1>)
               <slot2>)
     ... <slotn> <message>)
```

6.1 Slot Inquiry

```
(ask <object> <slot>)
```

returns the value of <slot>.

```
(ask mike age)

==> 35

(ask ,(ask mike father) age)

==> 60

(ask mike father age)

==> 60
```

6.2 Slot Assignment

```
(ask <object> <slot> = <value>)
```

assigns <slot> the value <value>.

```
(ask mike age = 36)

==> 36

(ask mike father age = 70)

==> 70
```

6.3 Value Deletion

```
(ask <object> <slot> forget:)
```

deletes the value from `<slot>`.

```
(ask mike age forget:)

==> age
```

6.4 Filler Description

```
(ask <object> <slot> describe:)
```

returns the filler description of `<slot>`.

```
(ask mike sex describe:)

==> (sex (default 'female))
```

6.5 Coreferences

```
(ask <object> <slot> == <object2> <slot2>)
```

makes the two slots of the two objects "coref" to each other. That is, they will always have the same value.

```
(ask mike father == ,(ask mike brother) father)
```

---

7. The Objects class and object

The foundation of ***ObjTalk*** consists of the classes `object` and `class`. The definition of these objects with their slots and methods describe the behavior of the base system.

7.1 The Object class

`class` realizes the different aspects with the slots `superc`, `descr`, `methods`, `corefs`, and `constraints`. The default value for `superc` is `(object)`. By the slots `descr` and `methods`, initializing functions are called that transform the slot descriptions and methods into an internal structure. The methods with filters were described in Chapter 4. The method bodies consist of complicated functions.

```
(ask class renew: class
     (superc object)
     (descr (constraints (default nil))
            (corefs (default nil))
            (superc (default (list object))
                    (if-changed
                      (instead
                        (lambda (<superc>)
                          (mapcar (function object-of) <superc>)))
                      (after
                        (lambda (<superc>)
                          ,!(hierarchie = ,(mkhierarchie ,!superc))))))
            (descr (default (list nil))
                   (if-changed
                     (instead mkdescr)
                     (after
                       (lambda (descrs)
                         (or (get ,!descr 'pname)
                             (putprop
                               descr
                               '(',(uconcat "<some_" ,!pname ">"))
                               'pname))))))
            (methods (default (list nil)) (if-changed (instead mkmethods)))
            (subclasses (default nil))
            (instances (default nil))
            (pname '"<some_class>"
                   (if-changed
                    (after
                     (lambda (name)
                       (let ((exname (exploden name)))
                         (and (> (car exname) 96)
                              (< (car exname) 123)
                              ,!(unique-name =
                                 ,(readlist
                                    (cons (- (car exname) 32)
                                          (cdr exname))))))))))))
     (methods (slots: => get-all-slots)
              (redefining-form: => sysredefinec)
              (remake: ,?<obj> =>=> sysremake)
              (remake: ,?<obj> with: ,*<msg> =>=> sysremake)
              (renew: ,?<obj> ,*<alist> =>=> sysrenew)
              (init: ,*<msgs> => sysinitc)
              (new: ,?<obj>:atom ,*<alist> => sysnew)
              (kill: => syskillc)
              (addslot: ,?<newslot>:atom ,*<newdescr> => sysaddslot)
              (repslot: ,?<newslot>:atom ,*<newdescr> => sysrepslot)
              (delslot: ,?<oldslot>:atom => sysdelslot)
              (replace: ,?<oldslot>:atom ,*<newdescr> => sysrepl)
              (make: ,?<obj>:atom with: ,*<msgs> => sysmake)
              (make: ,?<obj>:atom => sysmake)
              (addmethod: ,?<name>:atom ,*<method-rest> => sysaddmethodc)
              (delmethod: ,?<oldslot>:atom => sysdelmethodc)
              (repmethod: ,?<name>:atom ,*<method-rest> => sysrepmethodc)
              (describe: ,?<oldslot>:atom => sys??c)
              (instantiate: ,*<msgs> => sysinstwith)))

  ==> class
```

7.2 The Object object

`object` is the only class that has no superclass.

```
(ask class renew: object
     (superc)
     (descr (pname '"<some_object>"))
     (methods (get: ,*<msgs> =>=> getmultiple)
              (set: ,*<msgs> =>=> setmultiple)
              (redefining-form: => sysredefinei)
              (show: => (cdr (boundp 'pname)))
              (super: => (class-of self))
              (edit: =>
                     (or (getd 'edite) (load 'editor))
                     (eval (car (edite (ask self redefining-form:) nil nil))))
              (pp: =>
                   ($prdf (ask self redefining-form:) 0 0)
                   (terpri $outport$)
                   (terpri $outport$))
              (eval: ,?<expr> => syseval)
              (init: ,*<msgs> => sysiniti)
              (kill: => syskill)
              (viewed-as: ,?<class> ,*<msg> => sysviewed)))

  ==> object
```

---

8. ObjTalk Syntax

General form:

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

Messages to classes:

```
(ask <class> addmethod: <name> <filter> => <body>)

(ask <class> addslot: <slot> <filler-description>)

(ask <class> delmethod: <name>)

(ask <class> delslot: <slot>)

(ask <class> describe: <slot>)

(ask <class> instantiate: <slots-with-value>)

(ask <class> make: <object> with: <slots-with-value>)

(ask <class> new: <class>
       (superc <class> ...)
       (descr  <slot-description> ...)
```

```
            (methods <method> ...)
            (corefs  <coref> ...)
            (constraints <constraint> ...))

        <slot-description> ::= (<slotname> <filler-description> ...)

            <filler-description> ::= (default <s-expression>)
                                     (coref <path> ...)
                                     (if-changed (before <function>)
                                                 (instead <function>)
                                                 (after <function>))
                                     (if-forget (before <function>)
                                                (after <function>))
                                     (if-needed <function>)

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

            <type> ::=  =>  |  =>=>

            <filter element> ::= <constant> | <element variable> |
                                 <segment variable> | <class variable>

            <element variable> ::= ,?<symbol> | ,?<symbol>:<predicate>

            <segment variable> ::= ,*<symbol> | ,?<symbol>:<predicate>

            <class variable> ::= ,#<symbol>:<class>

            <method body> ::= <s-expression> ... | <function>

        (ask <class> replace: <slot> <filler-description>)


Messages to objects:

        (ask <object> <slot> <slotpath> <message>)

            <message> ::= <empty> | = <value> | forget: | describe: |
                          == <object> <slot>


        (ask <object> kill:)

        (ask <object> viewed-as: <class> <message>)

        (ask <object> pp:)

        (ask <object> edit:)
```