Chapter 6

Defining generic functions

The main tool in object-oriented programming using CLOS is that of a *generic function*. While the definition of an ordinary function, say using the `defun` macro, must have all the code that implements the function in the body of the `defun form`, this is not the case with a generic function. Instead, the implementation of a generic function is determined by one or more *methods* that can be textually separated from one another, often in different files.

When a method is defined, typically using the `defmethod` macro, the *name* supplied is what determines the generic function to which the method belongs.

Different methods on one generic function are typically distinguished by the nature of the requred arguments to which the method is *applicable*. Usually, that nature takes the form of a *class*.

For example, consider again a recursive implementation of the `length` function shown in Section 3.3, but this time defined as a generic function as shown in Code fragment 6.1.

The evaluation of the `defgeneric` form in Code fragment 6.1 results in the creation of a generic function named `length` with no methods on it. Any attempt to call a generic function with no methods on it will result in an error being signaled. The generic function `length` has a single required parameter named `list`, and no other parameters.

```
(defgeneric length (list))

(defmethod length ((list null))
  0)

(defmethod length ((list cons))
  (1+ (length (rest list))))
```

Code fragment 6.1: The `length` function defined as a generic function.

The first `defmethod` form in Code fragment 6.1 results in a method being created and then added to the generic function `length`. This method is applicable only when the argument given to the generic function is an instance of the system class named `null`. This system class has a single instance which is the object `nil` in this case used to mean the empty list. We can check the class of the object `nil` by typing the following form to the `read-eval-print` loop:

```
CL-USER> (class-of nil)
#<BUILT-IN-CLASS COMMON-LISP:NULL>
```

The second `defmethod` form in Code fragment 6.1 results in another method being created and then added to the generic function `length`. This method is applicable only when the argument given to the generic function is an instance of the system class named `cons`. This system class is the class of all `cons` cells. We can check that `cons` cells are indeed instances of this class by typing the following form:

```
CL-USER> (class-of '(a)
#<BUILT-IN-CLASS COMMON-LISP:CONS>
```

Now that we have two methods on the generic function named `length`, we can try it out:

```
CL-USER> (length '())
0
CL-USER> (length '(a b c))
3
```

We can also try out our function to see what happens if we give it an argument that is neither `nil` nor a `cons` cell:

```
CL-USER> (length 234)
```

What happens then is that an error is signaled, specifically the error condition named `no-applicable-method`, which is signaled precisely in this situation, namely that a call was attempted to a generic function, but the arguments given did not result in any method being applicable.

In the example in Code fragment 6.1 , the two methods are mutually exclusive, i.e., at most one method is applicable to a particular argument given to the generic function. But it is possible to have more than one method applicable for a given argument. We illustrate this possibility with a generic version of the standard function `describe` as shown in Code fragment 6.2.

```
(defgeneric describe (object))

(defmethod describe (object)
  (format t "This object is an instance of the class named ~s.~%"
          (class-name (class-of object))))

(defmethod describe ((object list))
  (format t "This object is a list.~%"))

(defmethod describe ((object null))
  (call-next-method)
  (format t "It has no elements.~%"))

(defmethod describe ((object cons))
  (call-next-method)
  (format t "It has ~d elements.~%" (length object)))
```

Code fragment 6.2: The `describe` function defined as a generic function.

In the example in Code fragment 6.2, we have a generic function named `describe` with four methods. In the first method, the parameter is not specialized. An unspecialized parameter is the same as a parameter specialized to the class named `T`. Since the class named `T` is the root of the Common Lisp class hierarchy, every object is an instance of that class. As a result, this first method is always applicable, no matter what argument is passed to the generic function.

The second method in Code fragment 6.2 is specialized to the system class named `list`. Recall that this class is a superclass of the classes named `null` and `cons`. In other words, a list is either a `cons` cell or the object `nil`. As a result, this method is applicable when the argument to the generic function `describe` is either a `cons` cell, or the object `nil`. So in this case, the first and the second methods are both applicable. However, the second one is *more specific* than the first one, because it has more restrictions on the objects that make it applicable. Therefore, when a list is given as an argument to this generic function, only second method will be called. but the first one will not.

The third method in Code fragment 6.2 is specialized to the system class named `null`. Recall that the only instance of this class is the object `nil`, which is also the object indicating the empty list.. When this generic function is called with `nil` as an argument, the first three methods are applicable. The third method is more specific than the second one, and the second one is more specific than the first one, so the third method will be called. However, the first thing that this third method does is to call the function named `call-next-method`. This is a *local* function that is automatically defined so that it can be used in method bodies. Its purpose is to call the next most specific method with the same arguments as the current method was called with. In our case, it therefore calls the second method with the object `nil` as an argument. When the call returns, the third method then prints a message that the list has no elements.

Similarly, the fourth method in Code fragment 6.2 is specialized to the system class named `cons`. Every `cons` cell is an instance of this class. When this generic function is called with a `cons` cell as an argument, the first, the second, and the fourth methods are applicable. The fourth method is more specific than the second one, and the second one is more specific than the first one, so the fourth method will be called. However, as with the third method, the first thing that this fourth method does is to call the function named `call-next-method`. In our case, it therefore calls the second method with the same `cons` as an argument as the fourth method was called with. When the call returns, the third method then prints a message with the number of elements of the list.

We can try out this function at the `read-eval-print` loop as follows:

```
CL-USER> (describe 1/2)
This object is an instance of the class named RATIO.
```

```
NIL
CL-USER> (describe '())
This object is a list.
It has no elements.
NIL
CL-USER> (describe '(a b c))
This object is a list.
It has 3 elements.
NIL
CL-USER>
```

The use of `call-next-method` makes it possible to avoid code duplication by doing some `code factoring`. In our case, the code that prints the message "The object is a list." should be executed both when the argument to the generic function is the object `nil` and when the argument is a `cons` cell. We could duplicate this code in the third and in the fourth methods, and just not define the second method. But such code duplication should be avoided, because if the code needs to be modified in the future, the person doing the modification, i.e., the *maintainer* would have to find all instances of the duplicated code, and they are not always close together as in our example. And if the code in question is not as small as the one in our example, duplicating it is worse, because the maintainer could make a mistake when modifying one of the instances of the code. So the use of a method that is applicable both when the argument is the object `nil` and when the argument is a `cons` cell, and the use of `call-next-method` is preferable.

In the preceding examples, each generic function had a single parameter. Generic functions, like ordinary functions, can of course have more than one parameter, and it can have optional parameters and keyword parameters just like ordinary functions can. But, contrary to methods in common object-oriented programming languages such as Java or C#, Common Lisp generic functions can dispatch at run time on more than one argument. In common object-oriented languages, it is even the case that the syntax has been chosen so that the argument being dispatched on is distinguished from other arguments, often by preceding the name of the method as in `x.m(...)` where `x` is a class instance being dispatched on. An analogous expression in Common Lisp would be written `m(x, ...)` instead. The fact that the class instance is just another function argument in the case of Common Lisp suggests two things, namely that the class instance can be any parameter; not necessarily the first one, and that more than one class instance can be dispatched on. Both are true as we shall see in the following examples.

Let's say we represent a *set* of numbers by a list. We want to write a `convenient-union` function that can take either two sets, a number and a set (in any order), or two numbers, and we want to return the resulting set. In the first case, the function returns the set that is the normal union of the two sets. In the other cases, we want to consider the number as being a singleton set and then return the resulting set. We can define `convenient-union` as a generic function as shown in Code fragment 6.3.

```
(defgeneric convenient-union (set1 set2))

(defmethod convenient-union ((set1 list) (set2 list))
  (union set1 set2))

(defmethod convenient-union ((set1 number) (set2 list))
  (union (list set1) set2))

(defmethod convenient-union ((set1 list) (set2 number))
  (union set1 (list set2)))

(defmethod convenient-union ((set1 number) (set2 number))
  (union (list set1) (list set2)))
```

Code fragment 6.3: A generic `convenient-union` function.

The generic function named `convenient-union` has two required parameters named `set1` and `set2`. It has four methods, each method specializing to both required parameters. The first method is applicable when both arguments are lists. The second method is applicable with the first argument is a number and the second argument is a list. The third method is applicable when the first argument is a list and the second argument is a number. Finally, the fourth method is applicable with both arguments are numbers. We can test this function at the `read-eval-print` loop as follows:

```
CL-USER> (convenient-union '(3 2 1) '(2 3 4))
(1 2 3 4)
CL-USER> (convenient-union 1 '(2 3 4))
(4 3 2 1)
CL-USER> (convenient-union '(2 3 4) 1)
(4 3 2 1)
```

```
CL-USER> (convenient-union 2 1)
(2 1)
```

In the preceding examples, a method specializer is always a class, either explicitly mentioned or the class named `t` when there is no explicit specializer. A Common Lisp generic function can also specialize to a particular object. This case is obtained with the use of a so-called `eql` specializer. We can illustrate this case with a version of the standard `documentation` function as shown in Code fragment 6.4.

```
(defgeneric documentation (object type))

(defmethod documentation ((object (eql 'list)) (type (eql 'function)))
  "This function returns a list of all its arguments.")

(defmethod documentation ((object (eql 'list)) (type (eql 'type)))
  "This class is a superclass of the classes CONS and NULL.")
```

Code fragment 6.4: A generic `generic-documentation` function.

Th generic function named `documentation` has two required parameters. It has two methods, each method specializes to both parameters, and each specializer is an `eql` specializer. The first method is applicable when the generic function is given the symbol `list` and the symbol `function` as arguments. The second method is appliable when the generic function is given the symbol `list` and the symbol `type` as arguments. We can test this generic function at the `read-eval-print` loop as follows:

```
CL-USER> (documentation 'list 'type)
"This class is a superclass of the classes CONS and NULL."
CL-USER> (documentation 'list 'function)
"This function returns a list of all its arguments."
```

It is of course also possible to have some required parameters with class specializers and some required parameters with `eql` specializers, in any order.