Ircam documentation

- Etudes et recherches
- Œuvres musicales
- Logiciels

PatchWork

Programming Guide

r e f e r e n c e

Second Edition, April 1996

IRCAM Centre Georges Pompidou

This manual was written by Camilo Rueda, in collaboration with Curtis Roads, under the editorial responsibility of Marc Battier - Marketing Office, Ircam.

Patchwork was conceived and programmed by
Mikael Laurson, Camilo Rueda, and Jacques Duthen.

Second Edition of the documentation, April 1996.
This documentation corresponds to version 2.0 or higher of PatchWork.

Ircam
1, place Igor-Stravinsky
F-75004 Paris
Tel. (33) (1) 44 78 49 62
Fax (33) (1) 42 77 29 47
E-mail ircam-doc@ircam.fr

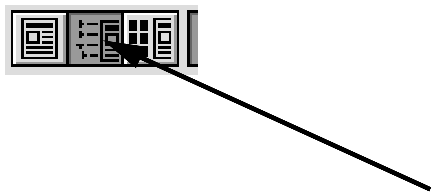
IRCAM Users’ group

The use of this software and its documentation is restricted to members of the Ircam software users’ group. For any supplementary information, contact:

Département de la Valorisation
Ircam
Place Stravinsky, F-75004 Paris

Tel. (1) 44 78 49 62
Fax (1) 42 77 29 47
E-mail: bousac@ircam.fr

Send comments or suggestions to the editor:
E-mail: bam@ircam.fr
Mail: Marc Battier,
Ircam, Département de la Valorisation
Place Stravinsky, F-75004 Paris



To see the table of contents of this manual, click on the Bookmark Button located in the Viewing section of the Adobe Acrobat Reader toolbar.

Contents

Résumé

Ce manuel s'adresse au lecteur qui souhaite construire des fonctions PatchWork. Il est conseillé d'avoir déjà une connaissance de la programmation par objets, telle qu'elle est mise en œuvre par le Common Lisp Object System (CLOS).
Toutefois, les informations techniques sur les catégories de données d'entrée et de sortie employées par PatchWork seront utiles pour comprendre les fonctions offertes pat PatchWork et ses différentes librairies.

Introduction

The PatchWork environment, its associated libraries, and the underlying Common Lisp world provide a rich set of functions for computer-assisted composition. However, for some users, the time will come when they would like to create their own PatchWork module. This *Programming Guide* is intended to document the process of writing new PatchWork modules and integrating them into the environment.
In order to understand this process you must know something about object-oriented programming, as it is practiced in the Common Lisp Object System (CLOS). Thus we provide some theoretical sections that explain the basic principles involved. The first chapter teaches you how to create PatchWork functions (Lisp code) that you can then turn into graphical modules. The second chapter deals with the advanced topic of creating PatchWork classes and methods, while the third chapter explains how to make your own library, like the Esquisse library supplied with PatchWork. The appendix explains the details of PatchWork types, since the first order of business in defining a new module is specifying what types of input and output data it handles.
Consult the *PatchWork Introduction* for basic information on the PatchWork system itself.

Creating Your Own PatchWork Module

This chapter walks the reader through the process of creating your own module in PatchWork. This process can be divided into three stages:

1. Assigning input and output types and defining module behavior with **defunp**
2. Converting a PatchWork function into a graphical PatchWork module
3. Customizing the graphical appearance of a module

The next sections present each stage in turn, along with a special section on the package scheme used by Common Lisp and PatchWork.

Creating a User-defined PatchWork Function

In order to create a PatchWork module you must first create a PatchWork function. The function is not yet a graphical module. Later we present how to turn the function into a graphical module.
Creating a PatchWork function involves defining its input and output types and specifying its internal behavior. Appendix A describes PatchWork's typing scheme. Use the special macro form **defunp** to create a PatchWork function. The syntax of this form is the following:

```
(pw::defunp <function-name> (<input parameters type-specs list>)
  <output-type name>
  <documentation string>
  <function body>  )
```

where <*function-name*> is the name of the function being defined, <*function-body*> is its definition (code), <*documentation-string*> is any sequence of characters enclosed in double quotes (that is, a Lisp string), <*output-type name*> is one of the output type names given above, and <*input parameters type-specs list*> is the list of type specifications for each one of the input arguments of the function. The last could be defined more precisely as

```
<input parameters type-specs> ::= {(<argument name> <type form>)}*
```

that is, any number (this the meaning of "*") of pairs of an argument name followed by a type form enclosed in parenthesis. The <*type form*> is in turn a type specification for the argument name to its left, whose complete syntax will be detailed later. For the moment it will be assumed that the <*type form*> is simply an input type name. For example, the definition:

```
(pw::defunp example-function1 ((length fix>0) (element fix/float))
list
  "This function constructs a list of numbers. The length of the list
is given by the 'length' argument. The list is filled initially with
the number given by the 'element' argument."
  (make-list length :initial-element element) )
```

The forms created by **defunp** are not exactly the same as those created by **defun**. In particular, **defunp** assigns the types of the input arguments and the output result as a property of the function name.

Converting a Function to a PatchWork Module

The previously-defined function can be converted to a PatchWork module box in either of two ways:

1. By selecting the menu item PWoper, and then: Lisp function...
   This evaluates the named PatchWork function (i.e., the name of the function you have previously defined) and puts a graphical module with the name of the function in an open PatchWork window.

2. By calling the function **PW-addmenu**. This adds the module to the UserLib menu where you can select it repeatedly with the mouse.

The syntax of **PW-addmenu** is the following:

```
(pw::PW-addmenu <menu> <list of functions>)
```

where <*menu*> is an existing menu object and <*list of functions*> is the list of function names to be added to that PatchWork menu. Menu objects are complex data structures that represent items in a Macintosh menu. For the purpose of the PatchWork user, it is probably enough to know that there is already a defined menu object that can be accessed by the Lisp function call **pw::the-user-menu** to which user-defined functions can be added. This menu object corresponds to the User Lib item in the PatchWork menu bar. The function defined previously could then be inserted in this menu as follows:

```
(pw::PW-addmenu (pw::the-user-menu) '(example-function1))
```

Note on the the pw:: Package

You might have noticed that the PatchWork forms **defunp** and **PW-addmenu** have been prepended with **pw::** in the examples given previously. This is done because PatchWork groups its symbols (variables and function names) in what is called a *package*. The prefix **pw::** actually means that the symbol that follows it belongs to the group (package) called **pw**. Common Lisp groups symbols in this way to avoid naming conflicts when a variety of users are allowed to define their own symbols, as is the case in PatchWork. Naming conflicts are avoided because two symbols of the same name are considered different by Lisp if they belong to different user-named packages, such as **user::foo** and **pw::foo**. Even though there is a lot more to the notion of a Lisp package than this, this note should suffice for the moment.

We have said that PatchWork's typing scheme is intended for checking valid connections in the graphical box representation of functions. It is also controls the box representation itself, as we explain next.

Adjusting the Graphical Representation of a Module

A PatchWork box defined as described previously has a graphical representation and behavior that depends on the types specified for the function it computes. The following definition, for example

```
(pw::defunp nth-fromlast ((n fix>0) (a-list list)) all-types
"This function returns the n-th element of the list 'a-list', counting
from the end"
  (nth (- (length a-list) 1 n) a-list))
```

implies a graphical representation of a PatchWork box with two small input boxes whose names are shown as *n* and *a-list* and whose initial values are 1 and (1 2), respectively. Also, the current value of each input box can be changed by dragging the mouse over it. The names are taken from the name of the arguments of the function.

The other two properties, initial value and mouse dragging behavior, are implied by the specific types *fix>0* and *fix/float*. PatchWork types can easily encompass these capabilities because they are not just names, but symbols that name objects. A PatchWork type could then be more appropriately defined as an object. An object is a particular Lisp (CLOS, actually) structure that includes both data and operations. The initial value of the type is part of its data. The possibility of changing values by mouse dragging is part of its operation. How to include a particular behavior in a type will not be detailed here, since this entails both a certain experience in Lisp programming and a good knowledge of the PatchWork internals.

On the other hand, manipulating the data of a type (graphical appearance, default value, etc.) is a rather frequent user need. How this is done will be explained later. Shown next are the different kinds of data that are specific to each type.

Type name (input)	Type data list
integer	(:view-size #@(36 14) :value 0 :min-val -9999 :max-val 999999 :type-list (fixnum))
fix/float	(:view-size #@(36 14) :value 0 :min-val -9999 :max-val 999999:type-list (fixnum float))
midic	(:view-size #@(36 14) :value 6000 :min-val 0 :max-val 12700 :type-list (fixnum list))
fix/fl/list	(:view-size #@(36 14) :value 0 :min-val -9999 :max-val 999999:type-list (fixnum float list))
approx	(:view-size #@(36 14) :value 1 :min-val 1 :max-val 16 :type-list (fixnum))
nilNum	(:view-size #@(36 14) :value 0 :min-val -9999 :max-val 999999 :type-list ())
Pfix	(:view-size #@(36 14) :value 100 :min-val 0 :max-val 999999 :type-list (fixnum))
fixs	(:view-size #@(36 14) :value "(1 2)" :type-list (list))
list-eval	(:view-size #@(36 14) :value "lst" :type-list (list))
object	(:view-size #@(36 14) :value "self" :type-list ())

As we can see above, each item in the data list is qualified by a keyword (the symbol starting with a colon **:**) which indicates the item's use. The **:value** keyword corresponds to the initial (default) value appearing in an input box of that type. The **:min-val :max-val** keywords specify the minimum and maximum values allowed for the graphical representation. This only restricts the set of values that can be graphically put into an input box of the type but in no way restricts the set of values for computation. An integer less than -9999 cannot graphically be entered (by dragging the mouse) into an input box of type "integer". But a box connected to that input can compute as output an integer of any size and it will be taken as a valid value. The **:view-size** keyword specifies the graphical size of an input box of that type. It is given as a Lisp point, thus the syntax:

#@(<length> <height>)

whose first component is the length and whose second component is the height, both in pixel units. The numbers 36 and 14 give the standard dimension of most PatchWork input boxes.

The **:type-list** keyword is probably the most important from a user's perspective. It gives a list of names that is used by the PatchWork graphical connection mechanism to check type compatibility. A PatchWork box B1 can be connected to the input IN1 of a PatchWork box B2 if the **:type-list** of the output type of B1 is contained in the **:type-list** of the input type of the IN1 input of B2 . The only data item of a PatchWork output type is its **:type-list**. If "()" is given as **:type-list** this means all types will be accepted. The **:type-list** of predefined PatchWork output types is the following:

Output Types

TYPE NAME	Type Data List
all-types	()
nil	()
list	(list)
fix	(fixnum)
float	(float)
bool	(bool)
freq	(float fixnum)
midics?	(fixnum list)
freqs?	(fixnum float list)
number	(float fixnum)
numbers?	(float fixnum list)
ch-ob	(chord)
note-ob	(note-ob)
ch-line	(chord-line)
beat	(beat)
measure	(measure)
collector	(collector)
no-connection	(no-connection)

A PatchWork type thus has an associated type list. Any item in the type data list can be dynamically changed using the **defunp** form. This allows the definition of new PatchWork boxes with graphical representations adjusted according to the functions they compute. For example, the following function definition avoids the problem of having an argument's default value that would certainly produce an error for that particular function (division by zero):

```
(pw::defunp geo-means ((n integer (:value 1 :max-val 10000))
  (times fix>0)) float
"this function returns a list of (weird) geometric mean values"
```

```
            (let (result)
              (dotimes (i times (nreverse result)) (push (/ (sqrt (* i n (1+ n)))
          n) result))))
```

Even though **defunp** can be used to change any item (or items) of the type data list, it cannot change the output type in any way. A function is supplied for defining new output types:

```
            (pw::add-output-type <ouput-type name> <output-type list>)
```

For example,

```
            (pw::add-output-type 'my-own-out-type '(fixnum bool))
```

defines a new output type called **my-own-out-type** whose **:type-list** is **(fixnum bool)**. A function defined with this output type would have an associated PatchWork box whose output can be connected to an input of any other box with at least **(fixnum bool)** in its **:type-list.**

Taking all these possibilities into consideration, the precise definition of type specifications given before can be restated as follows:

```
            <input parameters type-specs> ::= {(<argument name> <type form>)}*
```

where

```
            <type form> ::= { <type name> | <type name> <type list>}
```

and

```
            <type list> ::= { ( { <type keyword> < keyword value> }* ) }
```

Thus, the type form is either a simple type name or a type name followed by a type list where a type list is any number of pairs of keyword-name and value enclosed in parenthesis.

Defining Modules that Can Handle Optional Arguments

A Lisp function can have, in addition to its required arguments, different kinds of *optional arguments*. The particular kinds of *optionals* define different ways of relating the arguments of the function to the actual values being given to them when the function is called. For example, one might define a function having a required argument called *a* and two optional arguments called *b* and *c*. This would mean that *b* and *c* both get NIL if the function is called with only one value, that *b* gets the second value and *c* gets NIL if the function is called with two values and that *b* and *c* take the second and third values if the function is called with three values.

But one might also define a function with one required argument and with only one optional argument that represents the list of all additional values other than the first one in the list of values given for the function call (see any description of the Common Lisp language for more details). These two kinds of optionals, identified by the symbols **&optional** and **&rest**, are handled in PatchWork. The graphical consequence of supplying these types of optionals in a PatchWork **defunp** form is the following:

1. For **&optional** arguments:
The corresponding input of the defined PatchWork box is not shown when the box is first drawn. Pressing the Option key and clicking on the box makes each one of these arguments appear in turn, in the order in which they were defined.

2. For **&rest** argument (only one should be given):
The corresponding input of the defined PatchWork box is not shown when the box is first drawn. Pressing the Option key and clicking on the box will make an instance of (the graphical representation of) the argument to appear. Subsequent Option clicking will give an additional input of the same type.

For example, a function that computes the square root of a number or of the sum of any number of numbers could be defined as follows:

```
(pw::defunp all+1 ((num fix/float) &rest (other fix/float)) number
"this computes the square root of the sum of any number of numbers (at
least one)"
(sqrt (apply '+ (cons num other))))
```

On the other hand, defining a function that transposes a list of midicents either by one semitone or by a supplied number of semitones could be defined as

```
(pw::defunp trans-up ((notes list (:value '(6000 6300 6900)))
                      &optional (semitones fix>=0 (:value 1))) list
"transposes a list of midics by 'semitones' semitones or by one if none
is given"

(mapcar #'(lambda (note) (+ note (* semitones 100))) notes))
```

So finally, the complete precise specification of the parameters of the **defunp** form is

```
<input parameters type-specs> ::=  {(<argument name> <type form>)}*
                       [ &optional {(<argument name> <type form>)}* ]
                       [ &rest (<argument name> <type form>) ]
```

Where the expressions enclosed in square brackets may or may not be present. Note that the **&rest** part, if present, should be the last one.

Constructing PatchWork Classes

This chapter presents an advanced topic — a technique for defining PatchWork *classes* — objects that bind together data and the operations that operate on this data. PatchWork classes are an alternative to the PatchWork functions we've been presenting so far. Many of the core modules in the PatchWork system have been implemented as classes rather than functions.

After introducing the theory of PatchWork classes, this chapter details the steps you must go through to make classes, their associated methods, and the graphical modules. The basic procedure we outline takes the following form:

1. Define a class that inherits from the predefined class **pw::C-patch**.

2. Use **defmethodp** to define a method that computes the output of the box.

3. Use **defunt** to give it PatchWork types

4. Convert it to a module using the same procedures as with PatchWork functions

The Concept of Classes

Up to this point, we have followed a simple approach to both the notion of a type and to the construction of PatchWork boxes. Even though there is an implicit hierarchy of the basic type structure (defined by the set inclusion operation on the **:type-list** of the basic types given before), no explicit mechanism has been provided to construct hierarchies of complex types. In a sense, we have explored a static notion of types as sets. This simplification probably gives enough insight to develop PatchWork applications relevant to most situations. Nevertheless it encompasses only a small portion of the possible forms of computations in PatchWork. It embodies a concept of computing that could be called "forgetful" in the sense that a connected sequence of boxes defined in this way performs a particular data transformation that is independent of the past history of previous computations. There is no notion of *state*.

To cite a musical example, if we make a program to obtain a particular chord by exploring a combinatorial space of intervals, there is no means then of trying to obtain afterwards the next chord in that space. The **chord** module in PatchWork is an example of a PatchWork class. Each particular **chord** module is a different instance of the general class of **chord** modules.

Giving a precise meaning to this notion of state requires extending the concept of type, including in it not only a set of values but also the set of all operations that are possible on those values. The type *integer*, for example, would include not only the integer numbers but also the arithmetic operations. Complex types would include several sets of values and several sets of operations. For defining hierarchies, an ordering relation can be implied for two types such that the data sets and function (operations) sets of one are included in the other. Computer scientists have called these types *algebraic*. Other names used are *abstract data types* and *classes*.

A musical interval class (thus the type *interval*), for example, is something that must be defined in relation to some possible operations. It could be conceived as being subject to "expansion", "contraction", "inversion", etc, without any regard to its particular interpretation. It could well be applied to pitches or to rhythmic proportions (themselves probably defined in turn as types).

The terminology used in PatchWork for referring to classes, their associated data (sets of values), and operations is taken directly from CLOS—the object system extension to the Common Lisp. In this system, data sets of a class are referred to by names called *slots*. Operations are called *methods*. The syntax of CLOS provides a way of grouping the slots and methods of a class. Some details of this syntax are discussed later.

Instances

Once types are defined, entities can be associated with those types as was done before in **defunp** for the arguments of functions. Since these entities are rather complex (encompassing both the slots and methods of the underlying class), they have been given a particular name. They are called *instances* of the class. Of course it could also be said that a symbol of type *integer* is an instance of the integer class, and it would be both correct and coherent to say so, but usually the name *instance* is reserved for entities of a complex user defined type.

Instances provide the means for keeping computation state because a class can be defined to name in a slot any data structure, including a representation of the effects of past method operations on any instance of it. The behavior of methods, then, can be defined to depend on the particular state produced by their past invocations. If a class **chord**, for example, is defined with a slot that names an ordered list of midi-cent values, a particular instance of that class can use the slot to keep track of the current list of midicents resulting after several invocations of a method **transpose**.

Class Definition in PatchWork

The general (simplified) syntax of a class definition in PatchWork and CLOS is the following:

```
(defclass <class-name> (<subclass-of>)
  ( (<slot-name1> :initform <value1>)
    (<slot-name2> :initform <value2>)
                 .
                 .
                 .
    (<slot-nameN> :initform <value2>)  ))
```

where <*class-name*> is the name of the class being defined, <*subclass-of*> (if given) specifies the name of an already defined class that is defined to contain the one being defined (thus all of its slots and methods are *inherited* into the one being defined), <*slot-name1*>*...*<*slot-nameN*> give names to each slot defined for the class and <*value1*>*...*<*valueN*> are the respective values which are initially assigned to any instance of this class. The values can be of any type. These could be simple numbers, lists or even instances of other classes. The **:initform** keyword is required for specifying these values. (See the CLOS documentation for more keywords.) The following example defines a **simple-chord** class:

```
(defclass simple-chord ()
  ( (notes :initform '(6000 6400 6700)) ))
```

The class name is **simple-chord**. It inherits from no other class and has only one slot called **notes.** Any instance of this class will have initially the notes C,E,G, expressed in midi-cents, contained in its **notes** slot.

An instance of a class is defined in CLOS with the function **make-instance**. Its (simplified) syntax is the following:

```
(make-instance <class>)
```

Where <*class*> is a Lisp form that evaluates to the name of a class. An instance of the class **simple-chord** can be then created as follows

```
(pw::defunp construct-simple-chord () all-types
"this is a simple function that constructs a simple-chord instance"
  (make-instance 'simple-chord))
```

Now there is a PatchWork function (thus a PatchWork box) that constructs instances of the class **simple-chord**. Even though no methods have been included yet in the class definition, it could be used and transformed (changing its notes, for example) with the PatchWork boxes **get-slot** and **set-slot,** which are defined precisely for this.

Defining a function for constructing instances is not necessary of course (the function **make-instance** could be used directly), but it is always a good programming methodology. These instance definition functions are called *constructors* in computer science jargon. Method definition will be next considered.

Defining a Method for a PatchWork Class

One of the problems of normal function boxes is that they always evaluate all their arguments. It is much more flexible to be able to check one or two of the input arguments before evaluating further. This requires a more object-oriented style of programming, which can be implemented in PatchWork by using method boxes instead of function boxes.

The (simplified) syntax for the **defmethod** form resembles that of function definitions in PatchWork using the **defunp** form, as can be seen below:

```
(defmethod <method-name> (<argument-class-pairs>) <method body> )
```

Where <*method-name*> is the name of the method being defined, <*method-body*> is any sequence of Lisp (or CLOS) forms that define the behavior of the method and <*argument-class-pairs*> is any number of pairs of an argument name and its associated class (or type). The only difference with **defunp** is that here the class (or type) refers to any Lisp class (or type), not a PatchWork type of the kind explained previously. Also, the class (or type) is not mandatory as is the case for **defunp**. The following is a method definition for the **simple-chord** defined previously:

```
(defmethod transpose ((self simple-chord) (semitones number))
  (setf (slot-value self 'notes)
        (pw::l+ (* semitones 100) (slot-value self 'notes)))
  self)
```

This defines a method called **transpose** whose first argument is called *self* and is of the class (or type) **simple-chord**, and whose second argument, called *semitones* is of the type (or class) **number**. The method body uses the standard CLOS function **slot-value**, which takes two arguments: an instance of a class (here it is the argument *self*) and the name of a slot of the class where the instance belongs to (here *notes* because *self* represents an instance of the class **simple-chord**).

The Lisp function **setf**, which is the same as **setq**, assigns the value of its second argument to the place referenced in its first argument, thus to the *notes* slot of the instance represented by *self.* The second argument of **setf** takes the current contents of the *notes* slot of *self* and adds the semitones (expressed as midi-cents, thus the multiplication by 100) to each of the elements in the *notes* slot (this is what the PatchWork function **l+** does). The effect is to save the result of transposing the
simple-chord instance represented by *self.*

The names of the arguments in a method definition are totally arbitrary, but for historical reasons the name *self* is always used to represent instances of the class to which the method is said to belong, thus to the class **simple-**

chord in the above example. Saying that the defined method belongs to the class **simple-chord** is also arbitrary and not formally correct because it can also be said to belong to the class **number** of its second argument. In fact it belongs to both classes, though one can harmlessly decide to ignore the fact that it also belongs to the **number** class.

Converting Methods to Patchwork Boxes

Methods (defined in text) can be converted to graphical PatchWork boxes using a different form called **defunt**, whose syntax is exactly the same as **defunp**. The difference is that only type definitions and not behavior can be given using **defunt** (this distinction is only historical and can change at any moment, but for the time being it has to be considered). After having defined the above method using **defmethod**, one can give it PatchWork types as follows:

```
(pw::defunt transpose ((self nilNum) (semitones fix>=0)) all-types)
```

Care should be taken to give in this definition exactly the same names as those used for arguments in the **def-method** definition. The method can then be converted to a PatchWork box exactly as for normal functions, for instance by calling:

```
(pw::PW-addmenu (pw::the-user-menu)
      '(transpose construct-simple-chord))
```

which inserts menu items transpose and construct-simple-chord in the UserLib menu of PatchWork.

Type specification for all arguments in **defmethod** is not mandatory. The above definition could have been written:

```
(defmethod transpose ((self simple-chord) semitones) ...etc.)
```

In this case the argument *semitones* has not been given a type (or class). This means that **transpose** is now a method that clearly belongs to the class **simple-chord** because it is the only specificity imposed upon it. The *semitones* argument can be of any class.

A key concept of the structuring of types as classes is *inheritance*. This notion allows the organization of type hierarchies. Suppose for example that the above mentioned **simple-chord** class is refined to also refer to an attack point (in time) of the chord in a score. This could of course be done by simply adding an additional slot to the class definition, but this would then mean that all chords would contain the new slot so the notion of "simple-chord" would be lost. An alternative is to define a new class as follows:

```
  (defclass chord-with-attack (simple-chord) ((attack-time :initform
0)))
```

This class inherits from **simple-chord** all of its slots and methods. It also contains a new slot called *attack-time*. All instances of the class would have the *attack-time* set to zero when they are first defined. There could also be a constructor and a new method for this class

```
(pw::defunp construct-chord-with-attack () all-types
"this constructs an instance of 'chord-with-attack' "
(make-instance 'chord-with-attack))

(defmethod move-chord ((self chord-with-attack) (offset number))
  (setf (slot-value self 'attack-time)
        (+ offset (slot-value self 'attack-time))) )

(pw::defunt move-chord ((self nilNum) (offset integer)) all-types)
```

So chords-with-attack can not only be transposed but they can also be moved.

All classes defined up to here, although conceptually integrated with their methods, have been separated from them for their manipulation in PatchWork. The constructors defined with **defunp** and the methods typed for PatchWork with **defunt** form separate PatchWork boxes. This is only appropriate in very simple situations. More often what is needed is a PatchWork box that captures the notion of the class as a unit and hides this fact (which is really an implementation detail, most of the time) from the user. What is needed is to include in the class definition itself its association with a PatchWork box. This is done by using the mechanism of inheritance, as shown in the example below:

```
(defclass simple-chord (pw::C-patch)
  ( (notes :initform '(6000 6400 6700)) ))
```

The only difference with the definition given previously is the explicit inheritance from the class **C-patch**. This is a PatchWork class, as can be seen by the fact that the symbol is preceded by the package name **pw**. The class **C-patch** is the standard PatchWork class for a box. It contains slots and methods for handling all the editing and graphical manipulations possible in a PatchWork box. This means that **simple-chord** has also that capability by inheritance. The next step is to obtain (graphically) instances of the class that behave in a preestablished way when Option-clicked at their output boxes. That is, one wants the functional behavior of the box that encompasses the class **simple-chord** (or, for that matter, the class **chord-with-attack** which now also inherits, by transitivity, from **C-patch**). Specifying this functional behavior is done by using the **defmethodp** macro of PatchWork. Its syntax is the following:

```
(pw::defmethodp  <function-name> <class-name>
(<input parameters type-specs list>) <output-type name>
  <documentation string>
  <function body>  )
```

The only difference between this macro and **defunp** is an extra argument <*class-name*> which gives the name of the relevant class. The <*function body*> is the same as in **defunp** except that it can use a variable called *self* for referencing the particular instance of the class. Consider for example:

```
(pw::defmethodp move&transpose  chord-with-attack
                           ((offset integer) (semitones fix>=0)) list
"the box returns the list mentioned below"
  (move-chord self offset)
  (transpose self semitones)
  (list (slot-value self 'attack-time)
        (slot-value self 'notes)))
```

This method says that the box will return a list of two elements. The first element of this list is the value of the "attack-time" of the chord that results after moving the chord by an offset given by the "offset" parameter. The second element of the list is the list of midicents that results after transposing the chord by the number of semitones given by the "semitones" parameter.

Adding the PatchWork box to the user menu is now done as follows:

```
(pw::PW-addmenu-fun (pw::the-user-menu)
  'move&transpose 'chord-with-attack)
```

As can be seen above, a different function is now used for creating a menu-item for the new box. The PatchWork function **PW-addmenu-fun** is like **PW-addmenu** but with the added feature that a class name can be specified so that the constructed box can be made an instance of that class.

Making PatchWork Libraries

User-defined PatchWork modules can be integrated into a library. PatchWork libraries are convenient for defining personalized extensions to the standard PatchWork environment. Creating a PatchWork library entails creating a MacIntosh folder containing Lisp files in a certain configuration. This configuration is:

1. One or more files containing definition of modules using the PatchWork form **defunp**.

2. One or more files containing the definition of a MacIntosh menu hierarchy where modules are to be inserted.

4. A file that loads all files in (1) and (2), in that order.

5. A file whose MacIntosh name extension is **.lib** and that loads the file in (4). This could be the same as file (4).

For example, a user library called **my-music** might have the following folder structure:

```
Folder:          My-music
Files:           my-music-modules.lisp
                 my-music-menus.lisp
                 my-music-load-all.lisp
                 my-music.lib
```

Now we shall describe the contents of these file types.

PatchWork Module Definition Files

Files of this type contain definitions of PatchWork modules by means of the **defunp** form. This is explained in Chapter 2, so only a simple example defining a chord transposition module will be shown below:

```
(pw:defunp chord-transpose ((chord midics?)
(interval fix>0 (:value 100))) midics?
"this module transposes <chord> by <interval>. <chord> is a list of
MIDIcents"
  (pw::g+ chord interval))
```

File for Menu Hierarchy Specification

User modules should all be included in submenus of the UserLib PatchWork menu. This menu can be accessed by calling the function **the-user-menu**. Submenus are created with the function **new-menu** and added to another menu with the function **add-menu-items.** Finally, modules are added to submenus with the function **pw-addmenu**. The following example is typical of how these functions are used.

```
(defvar *my-submenu1* (pw::new-menu "My-modules"))
(pw::PW-addmenu *my-submenu1* '(chord-transpose chord-reverse))
(add-menu-items (pw::the-user-menu) *my-submenu1*)
```

The first line creates a menu item whose title is **My-modules.** The created menu is stored in the variable ***my-submenu1*.** The second line adds to this menu two PatchWork modules (previously defined with **defunp**) called **chord-transpose** and **chord-reverse**. The fourth line inserts all this in the Patchwork UserLib menu.

Files of this type thus contain a set of Lisp instructions similar to those given above.

File for Loading All Files into the Library

This is simply a file with a set of Lisp directives for loading all files in the library. A typical example is

```
(load "MY-LIB:my-music-modules")
(load "MY-LIB:my-music-menus")
```

The first line loads the file **my-music-modules**, which contains module definitions. The second line loads the file **my-music-menus,** which contains menu hierarchy definitions.

The substring **MY-LIB:** used in the file name specification is what is called in Common Lisp a *logical directory*. It allows one to identify a file by giving only its relative position with respect to some base folder. Thus if the two files above are contained in a folder called **My-music**, the logical directory **MY-LIB** can be defined to represent the physical folder **My-music** so that the above Lisp commands refer to the right files. The way to associate logical to physical directories in PatchWork libraries is described below.

The .lib File

The **.lib** file is the file that a library user selects when loading a library. It contains two lines of code, one for linking a logical directory to a physical folder and one for actually loading the library. A typical example is the following:

```
(pw::define-logical-path-dir "MY-LIB" "PW-LIB:My-music")

(eval-enqueue '(pw::find&load-lib "MY-LIB" "my-music-load-all"
"My-music"))
```

In the first line the PatchWork function **define-logical-path-dir** associates the logical directory **MY-LIB** with the physical folder **PW-LIB:My-music**. The substring **PW-LIB** is the logical directory name of the standard PatchWork folder User-Library where all accessible PatchWork libraries are contained. So the string **PW-LIB:My-music** says that the logical directory **MY-LIB** is really an abbreviation for the folder **My-music** contained in the Patchwork folder User-Lib. The second line uses the PatchWork function **find&load-lib** to actually load the file called **my-music-load-all** contained within folder **My-music**. This file loads all library files, as have already been described.

The function **find&load-lib** thus takes three arguments. The first (**MY-LIB**) is the name of the logical directory for this library. The second (**my-music-load-all**) is the name of the file that loads all the files in the library. The third (**My-music**) gives the folders path where the file **my-music-load-all.lisp** is located. In this case it is right inside folder **My-music**. If, say, file **my-music-load-all.lisp** were inside a folder called **My-loader**, which was itself inside the folder **My-music**, then the second line would read

```
(eval-enqueue '(pw::find&load-lib "MY-LIB" "my-music-load-all"
"My-music;My-loader"))
```

Note that in the above functions the file name extension **.lisp** is not given.

So the two lines of code given above would be included in a file with **.lib** as name extension. This file could be named **My-music.lib** for the example given.

Appendix - PatchWork Types

This appendix deals with PatchWork data types. This subject is important since the first order of business in defining a new PatchWork module is specifying what types of data it is expected to take in and generate. The type specification affects the graphical appearance and behavior of a PatchWork module. For example, the graphic module box allows no connections to its arguments unless they come from boxes that produce values of the correct type. It is thus a *static* type checking scheme. Note that PatchWork's typing scheme affects only the graphical representation of functions. There is no type checking done within the associated Lisp function itself.

Theory of Types

A programming language defines the domain of values that it can manipulate. A *typing scheme* is a structuring of this domain of values into different sets, each set corresponding to a *type* of the language. This structuring is established so that a precise meaning can be specified for each different operation in the language. An arithmetic operation such as multiply, for example, has a different meaning depending on whether it is being applied to a pair of integers or to a pair of matrices. A programming language usually defines some types as being "basic" in the sense that they form a collection from which all additional types must be constructed. A new type can be constructed by operating in predefined ways on the basic types, such as by merging two of them (that is, performing a set union operation).

Types can impose restrictions on operations that make no sense for values other than those of a precise type. For example, the operation **next-number-higher-than-3** probably cannot be assigned a meaning for values of other than the type "integer". Some languages choose to represent types by a name, **integer**, for example, whereas others might choose to represent them explicitly, enumerating the collection of values that make up the set, for example [0,1,2,.....]. Usually, each type also has a lexical representation that is a convention for writing a value of that type — for example, the known decimal notation for numbers. Programmers must be concerned either by the lexical representation, the name representation, or the whole typing structure, depending on the degree of sophistication of their programs. The following is a description of the PatchWork type structuring scheme.

PatchWork Input Types

From a user's perspective, types in PatchWork have names. The type name behaves as a mnemonic for the set of values of the type. PatchWork makes a distinction between *input* and *output types*. Input types represent the set of values for the arguments of an operation. Output types represent the values produced by that operation. The following are the PatchWork basic input type names:

INPUT type names

NAME	SET REPRESENTED
integer	integer numbers
fix/float	reals
fix/fl/list	reals OR lists of reals
Pfix	integers not less than zero
fixs	lists
midic	integer (or list of integers) between 0 and 12700 (MIDI cent)
approx	integer between 1 and 16
nilNum	any set
list-eval	lists that can be Lisp-evaluated
ch-ob	chord objects
note-ob	note objects
ch-line	chord sequence objects
menu	enumerated list

In addition, there are a certain number of predefined *aliases* for these basic types. An alias is just a different name for a basic type, or a particular restriction on the set of values for a basic type. The following are the PatchWork predefined Alias types:

Type aliases

ALIAS NAME	BASIC TYPE	SET REPRESENTED (if restricted)
list	fixs	
symbol	fixs	

freq	fix/float	(default element of the set is 440)
cents	midic	
freqs?	freq	freq or list of freqs
midics?	midic	
fix>=0	pfix	
fix>0	pfix	integers greater than zero
fix>0s?	fix>0	as above, plus lists of them
float	fix/float	
floats	fix/fl/list	
all-types	nilnum	
numbers?	fix/fl/list	

PatchWork Output Types

The following chart enumerates the result or output types possible for PatchWork modules.

Output Types

TYPE NAME	SET REPRESENTED
all-types	all values
nil	all values
list	lists
fix	integers
float	reals
bool	{t, NIL}
freq	real or integer
midic	integers OR lists of integers
freqs?	reals OR integers OR lists of them
number	reals OR integers
numbers?	reals OR integers OR lists of them
ch-ob	chord objects
note-ob	note objects
ch-line	chord sequence objects
beat	beat objects
measure	measure objects
collector	collector objects
no-connection	empty set

These are PatchWork's basic input and output types. Any number of other types can be defined using these, as will be explained below. An example of the differentiation between input and output types is an operation which takes a list of numbers as input, and returns either the sublist of negative numbers or the length of the input list if there are no negatives. It might then be defined with *list* as input type and with *numbers?* as output type.

Index