PCE/Lisp

PCE Common Lisp Interface
Edition 2.2 (May, 1993)
PCE 4.5.5 (May, 1993)

Anjo Anjewierden

This file documents the PCE/Lisp interface.

Copyright © 1993 University of Amsterdam

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission are preserved on all copies.

Correspondence address:

Anjo Anjewierden (anjo@swi.psy.uva.nl)
University of Amsterdam
Faculty of Psychology
Department of Social Science Informatics
Roetersstraat 15
1018 WB Amsterdam, The Netherlands

The software documented here is available through obtaining an XPCE licence. An on-line version of this document is available through selecting the '`pcelisp`' stack under Q-Cards.

1 PCE Common Lisp interface

PCE is an object oriented programming system, mainly used for the development of user interfaces. PCE is programmed from a symbolic programming language (e.g. Prolog or Lisp). PCE is written in the C language and runs under the X windowing system.

This document describes the interface between PCE and languages adhering to the Common Lisp standard *Steele:90*, the interface is called PCE/Lisp. Example programs that demonstrate the programming style have been included.

The interface has been implemented for Lucid Common Lisp and Harlequin LispWorks.

Refer to the PCE documentation (e.g. *PCE:prolog*, *PCE:overview*) and the PCE online documentation *PCE:online* for further details on the functionality provided by PCE itself.

PCE versions 1 (1984–1986) and 2 (1987) where written by the author, PCE 3 (1989) by Jan Wielemaker and the author. The current version, XPCE 4 (1992), was developed by Jan Wielemaker.

1.1 Version

The example programs in this document were tested with PCE version 4.5 and should work with later versions. The actual version number can be obtained with:

```
> (pce-get @pce :version)
:|4.5.5, May 1993|
```

PCE/Lisp will generally be installed as '`xpcelisp`' or '`xpceclos`' perhaps suffixed with a version number. Contact the system administrator for details. The "x" prefix indicates PCE is running under the X windowing system.

1.2 Release notes

PCE/Lisp is currently considered to be a mature system. Over the last two years the interface between PCE and Lisp has been improved in two ways. Firstly, new capabilities provided by PCE are taken advantage of. Secondly, the PCE/Lisp interface tries to be as close to what a Lisp programmer would expect.

The release notes listed below document the history of improving PCE/Lisp. Although we still cannot guarantee backward compatibility, the chances of incompatible changes are remote.

PCE 4.5.4. Any Lisp type can now be used as an argument to a PCE function. See Section 4.7 [Non-primitive Lisp types], page 19.

PCE 4.5.1. Improved debugging facilities. See Appendix E [Debugging facilities], page 57.

PCE 4.4.8. The syntax for PCE user defined classes (see Section B.6 [User defined classes], page 43) has been refined and extended.

PCE 4.4.6. Implemented the ability to send messages to "none" objects. See the function entry for `pce-send` and `pce-get`. Introduced the macro `pce-gc` to aid the internal PCE memory management functions (see Section 3.5 [Memory management], page 13). Added an optional `:resources` keyword argument to `defpceclass`.

PCE 4.4.4.. The most dramatic change is that PCE is now equipped with a mechanism to adapt to the "natural" lexicographic conventions of the host language. The natural case for Lisp is uppercase (as evidence of how old the language is) and the natural word break character is the hyphen. Existing PCE/Lisp programs will run under the new release provided that all underbar characters are replaced by hyphens and that keywords and object names did not contain mixed case. See Section 4.1 [Lexiographic conventions], page 17. An extension is that for each PCE class two Lisp functions have been added to the PCE/Lisp interface. For each class a function to create an instance of that class is defined (the function is called `pce-`*class-name*), and a predicate (called `pce-`*class-name*`-p`) tests whether the argument is an instance of the class or any of its sub-classes. Several programmers already used these conventions as a short hand to enhance readability, the conventions are now part of the PCE/Lisp interface itself. The programs in this document have been modified to use these conventions. The function `pce-class` has been renamed to `pce-class-name` (`pce-class` creates a new class). The undocumented function `pce-string` has been removed.

PCE 4.2. The current version of PCE/Lisp reflects several changes in the implementation of PCE. The most important change is that some primitive Lisp data types can now be stored directly in PCE and retrieved without conversion. See Chapter 4 [Type conversions], page 17.

1.3 Packages, features and naming conventions

1.3.1 PCE package

All functions belonging to PCE/Lisp reside in a package called `PCE`, the following line makes the functions available from other packages:

```
(use-package "PCE")
```

1.3.2 PCE feature

PCE defines a Lisp feature called `PCE`. The PCE feature can be used to determine whether PCE is available in a running Lisp program. The most frequent use of the PCE feature is as part of the Lisp initialisation file.

The following is part of my Lisp initialisation file:

```
#+PCE (pce-send (pce-error :no-behaviour) :kind :error)
```

1.3.3 Names of PCE functions

The names of the functions start with the character sequence `pce-` for functions that are exported, with `pcelisp-` for Lisp functions used internally (these may occasionally show up in a Lisp backtrace) and with `pcelisp-c-` for functions that call C functions in the interface.

1.4 Organisation

This document is organised as follows. In each of the following chapters a topic of relevance to PCE/Lisp is described. Part of the description involves the introduction of concepts, functions and sometimes a motivation. Where appropriate examples are included.

The PCE/Lisp functions are listed in an appendix. The function description in the appendix is often much more rigorous than in the main text and includes error conditions and other additional information.

1.5 Other PCE documents

The PCE documentation consists of documents about a particular topic of interest, such as this document, and an on-line manual which is referred to as the "PCE manual". The PCE manual is a PCE/Prolog program that contains a variety of utilities to learn more about PCE. The program is included on the PCE distribution tapes along with the SWI-Prolog system on which it depends. The contents of the manual is generated by both PCE itself (PCE knows what functionality is provided and what arguments are expected) and descriptions in hypertext format.

It is convenient to have the PCE manual running while programming in PCE/Lisp. There are many topics that are not discussed below. With some practice, the manual provides enough information to write significant programs without reading additional documentation.

The manual uses the Prolog notation (lowercase and underbars). Each time you see an underbar in the manual replace it by a hyphen when programming in PCE/Lisp.

1.6 Acknowledgements

Many have contributed to the current definition of PCE/Lisp, whether they wanted or not. Special thanks to Nigel Major, Han Reichgelt and Gertjan van Heijst for Lisp related help, and Jan Wielemaker for porting PCE to X and extensions that make a Lisp programmer's life a lot easier. Gertjan van Heijst improved the implementation of user defined classes from Lisp, see Section B.6 [User defined classes], page 43.

The earlier PCE/Lisp versions where developed as part of ACKnowledge (ESPRIT Project 2576). The current version takes into account the experience gained then.

2 Tutorial

This chapter describes PCE/Lisp through several examples illustrating the primary functions and the syntax of the interface. A more elaborate description of the functions is provided in later chapters.

The PCE/Lisp interface is defined in a Lisp package called `PCE`. The PCE package is imported into the `USER` package by default.

Throughout this document the Lisp prompt is the character sequence "`> `".

2.1 Introduction

The central notions in PCE are *objects*, *classes* and *messages*. An object is an instance of a class and an object's primary purpose is to act when it receives a message. Examples of classes built into PCE are window, point, circle, area, name, class (!?), message, and about 140 others.

There are three primary functions in PCE/Lisp. These functions directly access the PCE virtual machine. The functions are for creating objects given the name of a class and some initialising arguments (see Section 2.1.1 [Creating objects], page 5), activating behaviour on an object (see Section 2.1.2 [Activating objects], page 6) and retrieving values from an object (see Section 2.1.3 [Obtaining values], page 7).

2.1.1 Creating objects: `pce-new`

An object is created by specifying its class, initialising arguments and an optional object name. An instance of class point (for representing coordinates in a two dimensional plane) is created as follows:

```
> (pce-new :point 10 20)
@3300302 / POINT
```

where `:point` is the name of the class (all class names are given as Lisp keywords) and 10 20 the arguments required to initialise an instance of class point. `pce-new` returns the Lisp representation of the object created.

A read macro called `@` provides a convenient notation for referring to objects, see Chapter 3 [Representation], page 11. In the example above, the object can later be referred to as `@3300302`, which the `@` macro expands to an internal representation.[1]

For clarity, the name of the class is printed after the object reference. An object reference evaluates to itself, therefore:

---

[1] The object identifier (3300302) can be different when you try this. It is often a large number.

```
> @3300302
@3300302 / POINT
```

Some objects are much like global variables in Lisp. Named objects are created by stating the name as follows:

```
> (pce-new :name 'my-point :point 10 20)
@my-point / POINT
```

The `:name` keyword must be the first argument. Objects that have names associated with them are seldomly used in Lisp programs. You can always keep a reference to an object in a Lisp variable, for example:

```
> (setf my-point (pce-new :point 100 200))
@3303744 / POINT
```

Object names in PCE must be unique. PCE predefines several global objects: `@pce` for PCE itself, `@lisp` for the Lisp interpreter and `@`display for the current display. PCE also defines "special" objects that have a constant value: `@on` (for TRUE), `@off` (FALSE), `@nil`, and `@default` (use the default value) for example. In applications, named objects are only used for objects that are truly global (perhaps the main window of your application) and reusable objects (the gesture of dragging with the middle mouse button).

2.1.2 Activating objects: `pce-send`

Each object has a set of behaviour as defined by the class. Behaviour on an object is invoked by sending it a message. Invoking behaviour can be handled by the object in several ways: changing an internal value of the object, activating a procedure (method) associated with the object or a combination.

A behaviour is identified by a name. A point object has x behaviour to change the x-coordinate. Behaviour may be inherited from the super class, may be defined on the object itself and may also be delegated to another object. The PCE manual provides a full description of all the behaviours available.

A message is sent by specifying the *receiver* (an object), the *selector* (the name of the behaviour to invoke) and an optional list of arguments. The receiver is an object name, the selector a Lisp keyword and the types of the arguments are dependent on the behaviour. The return value is the receiver on success and the constant `NIL` on failure. Examples:[2]

```
> (pce-send @pce :info)          ; No arguments
...                              ; Lots of info on PCE
@pce / PCE
```

---

[2] The warning in the second example is printed by default for all undefined behaviours. The warnings can be turned off with: `(pce-send (pce-error :no-behaviour) :kind :ignored)`

```
> (pce-send @pce :shut-down)     ; Class pce has no shut-down behaviour
[PCE: SEND: No implementation for: @PCE/PCE ->SHUT-DOWN]
NIL                              ; Failed
```

It is obviously possible to embed other PCE/Lisp functions inside a call to `pce-send`, for example:

```
> (pce-send (pce-new :picture) :open (pce-new :point 100 100))
@3300687 / PICTURE
```

This creates a picture and then opens it at the position given by the point. The window appears on the display.

2.1.3 Obtaining values: `pce-get`

The `pce-send` function acts like a predicate: it either succeeds or fails. Obtaining values from an object is also achieved by sending a message with the function `pce-get`. PCE knows two types of behaviour, send behaviour to invoke some action and get behaviour to retrieve a value. Many classes define both of these behaviour types with the same name. Class point, for instance, has x send behaviour to change the x-coordinate and x get behaviour to retrieve the current value of the x-coordinate. Whereas send behaviour always has a side effect, the only possible side-effect of get behaviour is creating the value that is returned.

Values can be obtained by the function `pce-get`. The definition and arguments are identical to those of `pce-send`. The return value is computed by the receiver:

```
> (pce-get @pce :self)                ; Returns itself
@pce / PCE

> (pce-get @pce :trace)               ; Current state of PCE tracer
:USER
```

Some further examples with points and pictures:[3]

```
> (setf pict (pce-new :picture)       ; Create picture
        pos (pce-new :point 100 100))  ; Create point

> (pce-get pos :x)                    ; What is the x-coordinate?
100

> (pce-send pict :open pos)           ; Open picture at position

> (pce-get pict :position)            ; What is the position?
@3312575 / POINT
```

---

[3] From now on, when `pce-send` succeeds, the result (the object reference) is not printed in the examples.

2.1.4 Short hand for `pce-new`

PCE/Lisp generates a short hand notation for creating objects. The short hand has been introduced to make nested calls shorter and easier to read. Nested calls to `pce-new` occur frequently when writing "messages" (see Chapter 5 [Call back], page 23).

On initialisation PCE/Lisp generates a Lisp function corresponding to `pce-new` for all classes in PCE. The name of the Lisp function `pce-` followed by the name of the class. Additionally, a Lisp predicate is generated for each PCE class. These predicates test whether the single argument is an instance of the class or any of its sub-classes. Opening a picture at a certain position using `pce-new` and the short hand notation are given below:

```
;;  Long
(pce-send (pce-new :picture) :open (pce-new :point 100 100))

;;  Short
(pce-send (pce-picture) :open (pce-point 100 100))
```

In the remainder of this document the short hand notation will be used. Note that named PCE objects can only be created with `pce-new`.

2.2 Sample programs

One of the aims of PCE is that simple (standard) user interfaces can be developed without too much programming. This section provides a small selection, thereby showing what PCE/Lisp code looks like and illustrating the usage of the primary PCE/Lisp functions.

2.2.1 Hello world

The "Hello world!" program can be written in many different ways depending on how the text should appear on the screen (on a console there is not much choice).

The easiest version of "Hello world" is a one-line program that makes use of the standard informers. An informer displays (`@display` is the object representing the bitmapped display) a text and then forces the user to click a mouse button (indicating the text has been read). In user interface terms, a modal dialog is initiated. Alternatively a confirmer could have been used (see class display in the manual).

```
(defun hello-world-1 ()
  (pce-send @display :inform "Hello world!"))
```

An alternative is to create a text window (called view in PCE) and write the text into it. A view is a window specialised for viewing and editing text (it is based on classes editor and text-buffer). Creating a window is not sufficient to make it visible on the display, the open behaviour (which has an optional point argument) achieves this. The append behaviour writes text at the end of the view.

```
(defun hello-world-2 ()
  (let ((view (pce-view "Hello world example")))
    (pce-send view :open)           ; Open window on display
    (pce-send view :append "Hello world!")
    view))
```

The final alternative for "Hello world" considered here is writing into a graphics window (called picture in PCE). Again a window is created and also a text object (instance of class text). Class text represents "strings" as graphical objects. Graphical objects are displayed with the display behaviour of class picture, the first argument is the object to display, the second argument is the position (the coordinate system has (0,0) in the top left-hand corner).

```
(defun hello-world-3 ()
  (let ((pict (pce-picture "Hello world example"))
        (text (pce-text "Hello world!")))
    (pce-send pict :open)           ; Open window on display
    (pce-send pict :display text (pce-point 100 100))
    pict))
```

2.2.2 Interactive graphics

The last version of the "Hello world" program can be enhanced by allowing the user to move the text around the picture. PCE supports various mechanisms for programming and handling user initiated events (e.g. keyboard input and clicking the mouse). A high-level mechanism is provided by class gesture. A gesture is a standardised sequence of events. One such sequence is the move gesture: mouse down, mouse drag, mouse up. A gesture is reusable in that it can be associated with any number of graphical objects, it therefore is worthwhile to create it as a named object:

```
(pce-new :name 'move-gesture :move-gesture))
```

Any number of gestures can be associated with a graphical object through the recogniser behaviour. Adding the line:

```
(pce-send text :recogniser @move-gesture)
```

achieves the desired effect. See under "recognisers" in the PCE manual for additional recognisers and gestures.

A useful function in PCE/Lisp for debugging graphical applications and changing them on the fly is `pce-hierarchy`. It displays a hierarchy of the graphical objects in the Lisp window. For example:

```
> (setf pict (hello-world-3))

> (pce-hierarchy pict)
@3333702 / PICTURE
|   @3334019 / TEXT
```

A window (view, picture, dialog) is a sub-window. A full window corresponds to a frame, available from any graphical object through the frame behaviour, therefore:

```
> (pce-hierarchy (pce-get pict :frame))
@3333833 / FRAME
|   @3333702 / PICTURE
|   |   @3334019 / TEXT
```

2.2.3 Dialog boxes

One of the most frequent elements of user interfaces is a dialog box with a variety of controls. Dialog boxes in PCE can, for example, contain text-items (in which the user can type text), buttons (to initiate an action) and menu's (to make a choice from several alternatives). See under class dialog-item for a full list.

Programming dialog boxes is relatively easy once you trust the automatic layout mechanism and the layout style it is based upon. The most significant thing to note about the next program is that no positions or placement constraints are specified. PCE can work out the layout for itself. Another significant fact is the labels of the text-items and the buttons, these are given as Lisp keywords, the reason is explained in see Section 4.10 [Type overview], page 21.

```
(defun name-and-telephone ()
  (let ((dialog (pce-dialog "Please enter name and phone")))
    (pce-send-list dialog
                   :append
                   (list (pce-text-item :name)
                         (pce-text-item :phone)
                         (pce-button :ok)
                         (pce-button :cancel)))
    (pce-send dialog :open)
    dialog))
```

The user can enter text into the text-items (`TAB` to the next field) and press the buttons. No actions are associated with the items or the buttons. See Chapter 5 [Call back], page 23. Most user interfaces developed with PCE are mode free, that is the user can initiate commands when he or she thinks it is convenient.

3 Representation

PCE objects, from the Lisp point of view, are alien data types. PCE/Lisp only represents references to PCE objects, the internal representation of an object is maintained by PCE itself. Because all objects have a unique identifier it is only necessary to keep the identifier in Lisp.

This chapter describes aspects of the representation of PCE objects in Lisp.

3.1 PCE objects in Lisp

An object is represented by the `pce-object` defstruct. This defstruct is further specialised as `pce-reference`, `pce-assoc` and `pce-special`. `pce-reference` is used for objects that have a numeric identifier, `pce-assoc` for objects that have a symbolic identifier and `pce-special` for the predefined special objects of PCE.

```
pce-object Generic PCE object
|   pce-reference       Numbered PCE object reference
|   pce-assoc           Named PCE object reference
|   pce-special Special, predefined, object
```

The two primary functions are `pce-object-p` which tests whether something is a PCE object in Lisp, and `pce-object-id` which returns the identifier (name) of a PCE object in Lisp.

Here are some examples to illustrate the representation:

```
> (setf p (pce-point 10 10))
@3312887

> (pce-object-p p)              ; Is it a PCE object?
T

> (pce-object-id p)             ; What is the identifier
3312887

> (pce-reference-p p)           ; Is it a numeric PCE object?
T

> (pce-assoc-p p)               ; Is it a symbolic PCE object?
NIL

> (pce-point 'zap 10 10)
@zap / POINT

> (pce-reference-p @zap)        ; Is it a numeric PCE object?
NIL

> (pce-assoc-p @zap)            ; Is it a symbolic PCE object?
T
```

```
> (pce-object-id @zap)          ; What is the identifier?
PCE::ZAP
```

In general, `pce-reference-p` and `pce-assoc-p` are not used in application programs, only the PCE/Lisp interface uses them. To test whether something is a PCE object the predicate `pce-object-p` can be used. When `pce-object-p` succeeds there is no guarantee that the object exists, it merely means that the argument is represented in Lisp as a PCE object reference. The predicate `pce-exists-p` tests for an object's existence.

3.2 Equality of PCE objects

Lisp provides various functions to test whether two entities are equal. Something similar applies to the equality of two PCE object references. The rules for testing on the equality of PCE objects are:

1. If one of the objects is a symbolic reference, use `eq`. This most frequently happens for testing whether something is `@nil`, `@on`, etc.
2. If one of the objects is registered with `pce-register` (see Section 3.3 [Registering objects], page 12), again `eq` may be used.
3. In all other cases use the predicate `pce-equal`.

The predicate `pce-equal` compares the object identifiers for equality. This ensures equality as symbolic object references are interned into the `PCE` package. Like Lisp, PCE/Lisp makes a distinction between upper and lower case for object identifiers. Therefore, `@PCE`, `@|Pce|` and `@|pce|` refer to different objects. PCE/Lisp prefers objects in uppercase, avoid the use of vertical bars around object names.

3.3 Registering primary objects

In many applications one can make a distinction between primary and secondary objects. Primary objects are defined to be objects the application cannot work without, whereas secondary objects are objects created out of necessity (or in other words, because everything in PCE is an object). For example, in a drawing program the principal objects would be circles, boxes, lines and secondary objects would be points, areas and so forth.

PCE/Lisp provides a mechanism through which the application can register an object to be primary. The following facilities are provided for primary objects:

1. Primary objects always compare `eq`. This is both a performance bonus and also uses less memory.
2. Primary objects are never destroyed by PCE's garbage collector.[1]

The function `pce-register` registers an object. Note that `pce-register` is called automatically for symbolic object references, and that calls to `pce-register` should not be scattered throughout an application but be part of the design. For large applications it is worthwhile to specialise the `pce-reference` defstruct.

3.4 Objects and object references

A PCE object and a reference to a PCE object are obviously not one and the same. A PCE object is maintained internally in PCE and from Lisp one can only refer to it through its unique identifier. In the previous sections the relevant Lisp functions have been described from a practical point of view, here we are somewhat more rigorous.

The notation `@`*ref* is implemented as a read-macro and expands to a call on `pce-at` (named after the `@` symbol). `pce-at` converts *ref* into a defstruct (either `pce-reference` or `pce-assoc`) and returns that defstruct. If *ref* is a symbol, the symbol is interned in the `PCE` package and then looked up in a hash table. If the symbol appears the corresponding hash table entry is returned, otherwise a new entry is created. This explains why symbolic object references always compare `eq`. The function `pce-register` puts *ref* in the hash table also if it is an integer. If *ref* is an integer and it does not appear in the hash table a new defstruct is created each time *ref* is encountered. In an application, there is only a need to use `@`*ref* where *ref* is a symbol, numeric object references are handled implicitly. So, `@`*ref* is just syntactic sugar evaluating to an object reference.

Creating an object reference is not sufficient to create an object (see `pce-new`):

```
> @foo
@FOO / freed
```

`@foo` is created as an object reference and since it does not refer to an existing object, the class of which it is an instance cannot be found (PCE/Lisp assumes the object has been freed).

Functions like `pce-new` and `pce-get` automatically create an object reference for the object they return.

3.5 Memory management

Memory management in PCE is drastically different from that in Lisp. Unless you go for it, creating a dangling reference in Lisp is impossible. The Lisp garbage collector can determine whether something is or is not accessible. For PCE this is not possible. It could be the case that, say, Lisp holds a reference to an object that is not referenced internally in PCE. PCE's memory

[1] This is more important than it may appear now. Objects can disappear when they are not referenced by any other objects. This is hardly what you would expect for primary objects, for secondary objects this is a good strategy. See Section 3.5 [Memory management], page 13 for more details.

manager could thus decide to remove the object, and the reference from Lisp would be dangling. This section describes the pragmatics of memory management in PCE/Lisp.

It is important to remember that `pce-object-p` tests whether something is a valid PCE object reference, whereas `pce-exists-p` is an object reference that refers to an existing object.

Objects created with `pce-new` exist as long as necessary. They can be explicitly freed with free behaviour (defined on class object). That is:

```
> (setf p (pce-point 10 10))

> (pce-send p :free)

> (pce-exists-p p)
NIL
```

Objects can be freed implicitly as the result of freeing another object. Class chain is the PCE equivalent of a Lisp list:

```
> (setf ch (pce-chain)          ; Create a chain (PCE list)
        p (pce-point))          ; Create a point

> (pce-send ch :append p)       ; Append point to chain

> (pce-send ch :free)           ; Free the chain

> (pce-exists-p ch)             ; Does chain exist?
NIL

> (pce-exists-p p)              ; Does point exist?
NIL
```

This is an example where an object (the chain) is the only object holding a reference to some other object (the point). When the chain is freed, it is no longer necessary to keep the point and it will be freed as well.

Free behaviour should only be used when you are absolutely certain the object is no longer required. Examples where absolute certainty is more or less guaranteed are windows, graphicals and other objects you created with `pce-new` (and derived functions). For all other objects it is more appropriate to let PCE figure out whether an object should be freed (i.e. has no internal references). The later is achieved with done behaviour. The meaning of done is the same as free when the object has no references, and a no-op otherwise. An example. Suppose you are interested in the available displays:

```
> (setf displays (pce-get @display-manager :members))

> (pce-chain-list displays)
(@DISPLAY / DISPLAY)
```

According to the documentation *displays* is now a chain of all available displays. A question is

whether this chain was constructed as a result of the members behaviour, or resides as a slot value inside the `@display-manager` object. This question cannot be answered without looking at the source code of PCE. After the chain has been inspected, indicate you are done with it:

```
> (pce-send displays :done)
```

and it will be freed if it was the result of the `pce-get` and be retained otherwise. In short, never use free behaviour on values computed by `pce-get`, use done behaviour instead.

A final aspect of memory management in PCE is cleaning up the garbage. The mechanisms described earlier in this section do this implicitly. There is also an explicit mechanism (implemented by the Lisp macro `pce-gc`) which can be used in certain sections of your application. Unreferenced objects (i.e. garbage) are often the result of creating other objects. It is a good idea to use `pce-gc` around the Lisp code that initialises the user interface of an application. Not using `pce-gc` will cause the application to grow very large over time and slow it down. PCE will automatically perform a `pce-gc` while processing user events, Lisp code called from PCE does not require calls to `pce-gc`.

See class PCE for information on the amount of memory currently in use, especially:

```
> (pce-send @pce :info)
```

```
> (pce-get @pce :core-usage)
```

4 Type conversions

This chapter describes how PCE/Lisp handles the exchange of information between PCE and Lisp (and vice versa). It is necessary to agree on a standard way of dealing with Lisp symbols, keywords and strings, also upper and lower case require a uniform approach.

The type conversions listed below only apply to arguments of the Lisp functions implementing the PCE virtual machine (e.g. `pce-send`, `pce-get`, `pce-new` and derivates).

4.1 Lexiographic conventions

PCE is written in the C language and was first connected to Prolog. This history has resulted in the convention that all internally defined names are represented in lower case (a convention in both C and Prolog) and the underbar character (`_`) is used to separate words in a single name. For example, the concept "display position" would be written down as `display_position` in the implementation of PCE, whereas Lisp programmers might prefer `DISPLAY-POSITION`.

The current version of PCE/Lisp provides the natural lexicographic conventions for Lisp. Lisp programmers can write down `DISPLAY-POSITION`, which PCE will interpret as being equivalent to `display_position`. There is no performance penalty as PCE/Lisp tells PCE to perform a conversion precisely once at initialisation. The PCE on-line manual is written in Prolog and still uses lower case and underbars.

4.2 Lisp numbers

An integral number is passed unmodified over PCE/Lisp. PCE does not provide extremely large numbers, but it can represent numbers of the Lisp type fixnum. The PCE class for floating point numbers is class real, PCE/Lisp automatically converts `floatp` to an instance of class real and back. Class number represents fixnums as objects.

PCE/Lisp treats Lisp numbers as objects, you can do:

```
> (pce-send 2 :plus 5)
2                               ; Still returns the receiver!

> (pce-get 2 :plus 5)
@3476536 / NUMBER               ; Returns instance of class number

> (pce-get @3476536 :value)
7                               ; Result computed correctly
```

Sending messages to numbers can be useful when you have to do computations inside PCE.

4.3 Lisp keywords

PCE contains a predefined class for a character sequence constant. The class is called "name" or sometimes "atom" (after its Prolog counterpart). In contrast with previous implementations of PCE/Lisp the symbol name of a keyword is passed unmodified to PCE (and vice versa). Thus, `:hello`, `:|Hello|` and `:|hELLO|` are all different.

So, the rule is that each time the PCE documentation mentions a name or atom as the argument, a Lisp keyword should be used.

The selector of a message (second argument in `pce-send` and `pce-get`) and the name of a class are always expressed as keywords. Examples of the use of keywords:

```
> (pce-get @pce :class-name)
:PCE

> (pce-name "FRED")
:FRED
```

The last example may seem somewhat strange at first. First, the `:name` keyword of `pce-new` is normally used to create named objects (see Section 2.1.1 [Creating objects], page 5). Here, `:name` refers to the PCE class name. The return value is not a PCE object because all instances of class name are converted to keywords by PCE/Lisp.

A note on efficiency. Passing keywords in PCE/Lisp is about as fast as passing integers. Passing the characters making up the keyword and the case conversion is only performed once for every keyword and cached thereafter.

4.4 Lisp symbols

Lisp symbols present an additional problem. A symbol has more attributes than its name. In PCE/Lisp both the name and the package name of a symbol are preserved. This is implemented as follows. Each time an argument is given as a Lisp symbol, PCE/Lisp converts the symbol to an instance of class lisp-symbol. The opposite conversion is performed from PCE to Lisp. Class lisp-symbol is a class defined by PCE/Lisp, it is a sub-class of name. This implies that when PCE expects a name, a Lisp symbol may also be provided. Some examples:

```
> (setf sh (pce-sheet))

> (pce-send sh :hello 'hello)
@3300300 / SHEET

> (pce-get sh :hello)
HELLO

> (eq 'hello (pce-get sh :hello))
T
```

Retaining package information has the disadvantage that two symbols with the same name but in different packages are not identical. Lisp symbols should therefore only be used when they are to be retained as Lisp symbols, otherwise keywords or strings are more appropriate. Note that function names are also symbols.

4.5 Lisp strings

PCE defines class string. This is a proper class and instances are represented as object references:

```
> (pce-string "Hello")
@3300748 / STRING
```

Often, it is required that a Lisp string is not converted to a PCE string but is retained as a Lisp string. PCE/Lisp defines class lisp-string for this purpose. Each Lisp string encountered as an argument to a PCE virtual machine instruction is converted to an instance of class lisp-string internally in PCE:

```
> (pce-send sh :string "Hello")

> (pce-get sh :string)
"Hello"
```

There is a small caveat here. Obtaining the value of a PCE string returns a keyword because the value is implemented as a PCE name:

```
> (pce-get (pce-string "Hello") :value)
:HELLO
```

Most likely, this is not what is expected. See Section 4.9 [Explicit conversion], page 20.

4.6 Lisp conses

Lisp conses (the building blocks of lists) can also be represented. PCE/Lisp defines class lisp-cons for this purpose. Conversion of a cons has to be done explicitly with the functions `lisp-to-pce` and `pce-to-lisp`. The elements in a cons may be of any of the types recognised by PCE/Lisp. Some examples:

```
> (lisp-to-pce '(this :is 1 "list" (containing . a) 1.0 cons))
@3301224

> (pce-to-lisp @3301224)
(THIS :IS 1 "list" (CONTAINING . A) 1.0 CONS)
```

4.7 Non-primitive Lisp types

Non-primitive Lisp types can also be passed as arguments to PCE functions. PCE has no knowledge

of Lisp types, with the exception of the primitive Lisp types mentioned above, so it can hardly do anyting else than keep a pointer to them and hand them back to Lisp when appropriate.

The most frequent case when handing other Lisp types is useful, is as an argument to a function called back from PCE. For example, if you use CLOS, then CLOS objects can be passed as an argument.

The implementation of this facility requires that the Lisp datum being passed is not reclaimed by the garbage collector. All Lisp datums passed are stored in a hash-table part of PCE/Lisp. These datums are never deleted.

There is a caveat. Non-primitive Lisp types cannot be saved to disk by PCE's saved state facility (`save-in-file` on class object).

4.8 Type coercion

Where possible PCE will automatically coerce a given argument to a type it expects. The expected type is listed in the PCE manual. The primitive types name, string and integer are automatically coerced if possible. More fancier coercions include using a file name where an instance of class file was expected.

4.9 Explicit conversion

The type conventions described above work well in most situations, sometimes it is necessary to explicitly convert a value returned by PCE. One example is the representation of file and directory names, these are represented as instances of class name inside PCE.

```
> (setf f (pce-file "~/.Xdefaults"))

> (pce-get f :name)
:|/staff/anjo/.Xdefaults|
```

Here, the default conversion is applied: a PCE name is converted to a Lisp keyword.

PCE/Lisp provides both a specialised and a general purpose conversion mechanism. The specialised version (`pce-get-string`) converts to Lisp strings, this is the most frequent case. The syntax and semantics are identical to `pce-get` except that the return value is converted to a Lisp string. Given the file example:

```
> (pce-get-string f :name)
"/staff/anjo/.Xdefaults"
```

The general version is called `pce-get-as` and is again like `pce-get` but takes a type to convert to as the first argument. The above example can also be written as:

```
> (pce-get-as :lisp-string f :name)
"/staff/anjo/.Xdefaults"
```

The type (first argument of `pce-get-as`) must be a PCE class name. If the conversion could not be carried out, a warning message is printed on the console and the result is not converted (i.e. the result is the same as would be obtained with `pce-get`). Another example is converting to an integer:

```
> (pce-get-as :int (pce-string "1992") :value)
1992
```

4.10 Type overview

The following table lists what PCE/Lisp does when a certain Lisp type is encountered as an argument to a PCE virtual machine instruction. The first column lists the type in Lisp, the second the type in PCE and the third what goes on internally in PCE/Lisp. The conversions are bi-directional and automatic.

Lisp type	PCE type	Internal
cons	lisp-cons	(lisp-to-pce *cons*)
		(pce-to-lisp *lisp-cons*)
fixnum	int	---
float	real	(pce-real *fixnum*)
keyword	name	(pce-name *keyword*)
string	lisp-string	(pce-lisp-string *string*)
symbol	lisp-symbol	(pce-lisp-symbol *symbol package*)

Given that PCE performs automatic type coercion where possible, Lisp programmers have a choice as to what Lisp type to use. For example, all of the following would be perfectly legal:

```
(pce-button :hello)
(pce-button 'hello)
(pce-button "hello")
```

The PCE documentation states that the first argument of a button (the label) is a name. This suggests that a Lisp keyword would be most appropriate. There are no problems with case here because PCE has a built-in notion of how to capitalise the labels of user interface objects (in order to ensure typographical consistency).

Informally the rules and their motivation are:

1. Use a keyword when the PCE documentation states that the argument is a name. Keywords are downcased and PCE will capitalise on the display according to its internal rules (these state that the first character and the first character after a hyphen are capitalised, the hyphen is replaced by a space). Keywords also have the advantage of being defined precisely once (both in Lisp and in PCE), possibly minimising the amount of memory used.
2. Use a symbol when you are using PCE for storage and need to preserve package information. Using a symbol where PCE expects a name is often inconsistent because the package information will be stripped off and is therefore redundant.

3. Use a string when you are using PCE for storage and would use a string in Lisp as well.

5 Calling Lisp functions from PCE

PCE uses the "call-back" paradigm for interacting between the user and the application program. User interface objects have associated messages which are activated when the user performs an agreed gesture (for example clicking the mouse). The action can stay inside PCE or it can call back to Lisp.

Application defined actions are often implemented through instances of class message. Messages are much like quoted Lisp function calls which are evaluated when the user activates the user interface object with which the messages are associated.

5.1 The Lisp object

PCE defines a special class called host. PCE/Lisp defines a single instance of this class called `@lisp` which represents the Lisp interpreter. An example:

```
> (defun zap () (format t "Zap called"))
ZAP

> (setf call-zap (pce-message @lisp 'zap))

> (pce-send call-zap :forward)
Zap called
NIL
```

A message that calls `@lisp` is created. The only argument of the message (`'zap`) is the name of the function to be called. Additional arguments could have been given. Forwarding a message results in the message being executed, in the example the Lisp function `ZAP` is called. This mechanism is very convenient for creating user interfaces and associating Lisp code with user interface objects. The next example creates a software button on the display and associates a message to call the `ZAP` function:

```
(defun zap-button ()
  (let ((dialog (pce-dialog))           ; Create dialog box
        (button (pce-button :call-zap   ; Create button calling zap
                            (pce-message @lisp 'zap))))
    (pce-send dialog :append button)    ; Put button in dialog
    (pce-send dialog :open)))           ; Open dialog on display
```

The general form of a message that calls back to Lisp when executed is:

```
(pce-message @lisp function-name &rest args)
```

Note that *function-name* must be the name of a function (as a Lisp symbol) that exists at the time the message is executed. Lisp symbols retain package information (see Section 4.4 [Lisp symbols], page 18), so the function must be accessible from the current package when the message is created or refer to the package explicitly with the `::` qualifier.

The sequence `(pce-message @lisp ...)` may be abbreviated to `(pce-funcall ...)`.

5.2 Passing arguments to functions

Arguments to Lisp functions called from PCE are passed by providing additional arguments when creating the message object. When the value of an argument is known when the message is created it can be given directly. If the value is unknown the PCE special objects `@arg1`, `@arg2`, ... may be used. Argument substitution is as follows. The first argument provided with `:forward` on the message is referred to as `@arg1` in the message, the second argument is referred to as `@arg2` and so forth.

Passing arguments is performed automatically by PCE when appropriate. An example is entering a piece of text into an instance of class text-item. The text entered by the user is forwarded as `@arg1` when the user hits the return key to signal finishing typing.

The general principle is illustrated by the program below.

```
(defun ask-name (name)
  (format t "You have entered ``~A'' as your name~%" name)
  t)


(defun name-dialog ()
  (let ((dialog (pce-dialog "Please enter your name"))
        (item (pce-text-item :name "" (pce-funcall 'ask-name @arg1))))
    (pce-send dialog :append item)
    (pce-send dialog :open)
    dialog))
```

The message associated with the text-item is

```
(pce-funcall 'ask-name @arg1)
```

That is, the Lisp function `ask-name` is called with one argument. The value of the argument is forwarded by the text-item when the user hits the return key; it is the text typed. Suppose the user typed "fred", the following PCE virtual machine instructions are then generated:

```
(pce-send <message> :forward "fred")     ; By the text-item
|   (pce-send @lisp 'ask-name "fred")    ; Executing the message
|   |                                    ;   after substituting
|   |                                    ;   @arg1 by "fred"
|   |   (ask-name "fred")                ; By Lisp
```

It is not significant that `@arg1` is the first argument of the message. The program can be improved by throwing away the dialog after the user has finished typing.

```
(defun ask-name-2 (dialog name)
  (format t "You have entered ``~A'' as your name~%" name)
  (pce-send dialog :free)
  t)
```

```
(defun name-dialog-2 ()
  (let* ((dialog (pce-dialog "Please enter your name"))
         (item (pce-text-item :name ""
                              (pce-funcall 'ask-name-2 dialog @arg1))))
    (pce-send dialog :append item)
    (pce-send dialog :open)
    dialog))
```

Some general remarks. The program also illustrates some of the decisions regarding the Lisp types used. The line

```
(pce-text-item :name "" <message>)
```

creates a text-item. The first argument (i.e. `:name`) is the label after which the user can enter text. The value of the label is unlikely to change during the execution of the program and is therefore given as a keyword. The second argument (i.e. `""`) is the current value typed by the user, in this example it is empty and a Lisp string. Providing a keyword (i.e. `:||`) would not be appropriate because the user will change the value. The text typed by the user is returned to Lisp using the same type as originally provided when the text-item was created (i.e. as a Lisp string in this example). This can be important. Consider that you require the user to enter the year of birth:

```
(pce-text-item :year 1992 <message>)
```

The user now has to enter a number (with 1992 being the initial value). The text-item will flash when the user enters a non-number.

5.3 Computing arguments: obtainers

The next program is slightly more complicated once again. It requires the user to enter her name and a password. When finished the user hits an "Ok" button, a Lisp function then checks whether the name and password are valid. The password checking function is

```
(defun check-password (login password)
  (format t "Logged in ``~A'' with password ``~A''~%" login password)
  t)
```

The difficulty here is that the login-name and the password are entered into two different text-items, so associating a message which each of the text-items does not achieve the desired effect. A kludge would be to pass the text-items themselves to `check-password`, which would then read:

```
(defun check-password (login-name-item password-item)
  (format t "Logged in ``~A'' with password ``~A''~%"
          (pce-get login-name-item :selection)
          (pce-get password-item :selection))
  t)
```

From a software design point of view this is barely acceptable because user interface and application

code is intertwined.

The proper solution in PCE/Lisp is to use the PCE equivalent of delayed evaluation. Delayed evaluation (like quoting in Lisp) is achieved by instances of class obtainer. For historical reasons the class name of an obtainer is denoted `?`. Obtainers behave like messages, but rather than internally executing a `pce-send`, they execute a `pce-get` and then replace themselves by the value computed. For example, the login-name is computed by the following obtainer:

```
(pce-? login-name-item :selection)
```

When this obtainer is executed it is replaced by the value of the selection. The resulting program is given below.

```
(defun ask-password ()
  (let* ((dialog (pce-dialog "Please enter name and password"))
         (name (pce-text-item :name ""))
         (password (pce-text-item :password ""))
         (ok (pce-button :ok
                         (pce-funcall 'check-password
                                      (pce-? name :selection)
                                      (pce-? password :selection)))))
    (pce-send-list dialog :append
                   (list name password ok))
    (pce-send dialog :open)))
```

5.4 Multiple messages

In the examples so far, a single message was associated with a user interface object. It is also possible to associate multiple messages and even conditional messages. The password program has one serious flaw, the dialog box stays on the screen after the user has entered the information. Specifically, we want to modify the program such that (a) the user gets a chance to correct typing errors; (b) destroy the dialog when a correct login-name and password have been provided.

There are several PCE classes that group messages. Lisp programmers will be familiar with many of them.

The PCE class block represents the notion of multiple messages. Each message in a block is executed sequentially until one of the messages fails. Failure can either be the result of an internal failure in PCE (comparable to `pce-send` returning `NIL`) or a Lisp function returning `NIL` (for messages to `@lisp`). A version of the password program incorporating blocks is given below.

```
(defun check-password (login-name password)
  (format t "Validated ``~A'' with password ``~A''~%" login-name password)
  (string= login-name "anjo"))


(defun ask-password-2 ()
  (let* ((dialog (pce-dialog "Please enter name and password"))
```

```
        (name (pce-text-item :name ""))
        (password (pce-text-item :password ""))
        (ok (pce-button :ok
                    (pce-block
                            (pce-funcall 'check-password
                                    (pce-? name :selection)
                                    (pce-? password :selection))
                            (pce-message dialog :free)))))
    (pce-send-list dialog :append (list name password ok))
    (pce-send dialog :open)
    dialog))
```

`check-password` has been modified to only succeed on certain login-names, and the block in `ask-password-2` first checks the password and then destroys the dialog box. The second message is only reached when `check-password` succeeds.

6 Support for large applications

Both PCE and PCE/Lisp offer some support to facilitate the development of large applications. For large applications design issues become much more important than for small (prototype) applications, especially regarding organisation of the source code, consistency, memory usage and efficiency.

6.1 Attributes

A very simple and useful feature is that any object can be "decorated" with user defined attributes. An object normally obtains its attributes from the class of which it is an instance, but it may also have private attributes. The private attributes can be used to hold application specific information. For example, if you have a window for which it is important to know who is the user and when it was created, there is a choice between storing this information in Lisp or creating additional attributes for the window. The latter is achieved with the utility function `pce-create-attribute`:

```
;; Assume view is known
(pce-create-attribute view :user "Fred Flintstone")
(pce-create-attribute view :date-created (pce-date))
```

The first argument is the name of the attribute (which should obviously by different from attributes defined by the class) and the second argument is the initial value. `pce-send` and `pce-get` may be used to change and retrieve the values of attributes in the usual way:

```
> (pce-send view :user "Wilma Flintstone")

> (pce-get view :user)
"Wilma Flintstone"
```

See classes sheet and attribute for further information.

6.2 PCE storage classes

PCE defines a wide range of general purpose storage (representation) classes. Because of the genericity it can be more efficient to use a more dedicated class for some applications. This is similar to using Lisp lists (a very general representation) or a particular defstruct (very specific).

PCE/Lisp provides a simple mechanism to define new PCE storage classes. A storage class is defined to be a class that has no behaviour except creating instances and modifying the values of attributes. An example storage class is keeping track of personal information. Each person has a name, address and telephone number. The obvious way to code this is in plain PCE is as follows (see Section 6.1 [Attributes], page 29):

```
(defun new-person (name address phone)
  (let ((person (pce-sheet)))
```

```
    (pce-create-attribute person :name name)
    (pce-create-attribute person :address address)
    (pce-create-attribute person :phone phone)
    person))
```

A PCE sheet is like an association list in Lisp, it represents a collection of key / value pairs. The virtual machine instructions work on sheets:

```
> (setf fred (new-person "Fred Flintstone"
                         "Pebble way 12"
                         "7812"))

> (pce-send fred :phone "001 - 7812")

> (pce-get fred :name)
"Fred Flintstone"
```

The only disadvantage is that, as with association lists, the names of the attributes are repeated for every single person.

A storage class equivalent to the sheet representation above is:

```
(pce-storage-class
   :person               ; Class name
   :super :object ; Super class name
          ; attribute   type      access
   :slots `((:name    :lisp-string :both)
            (:address :lisp-string :both)
            (:phone   :lisp-string :both)))
```

This defines "person" as a new PCE class. It is a sub-class of class object (the most general PCE class) and has three attributes all of which take a Lisp string as the value and can be accessed with both `pce-send` and `pce-get`. An instance of class person can now be created with `pce-person`:

```
> (setf fred (pce-person "Fred Flintstone"
                         "Pebble way 12"
                         "001 - 7812"))

> (pce-class fred)
:PERSON
```

An instance of class person occupies less memory than the equivalent sheet, if the number of instances is large this can be a significant advantage. Also, PCE performs type checking on the attributes of a class (they have to be Lisp strings in the example) and not on the attributes of a sheet.

6.3 PCE argument types

The types of PCE values (e.g. attributes of instances, arguments of behaviours) are specified using a dedicated syntax. Specifying types is relevant when defining new classes. The syntax in its Lisp

form is given below:

Type	Description
`:point`	Instance of class point or its sub-classes
`:[point]`	A point or `@default`
`:point*`	A point or `@nil`
`:point?`	A point or an obtainer
`:[point]*`	A point or `@default` or `@nil`
`:int|point`	An integer or a point
`:|0..100|`	Integer in the range (0, 100) inclusive
`:|{HELLO,WORLD}`	One of `:hello` or `:world`

The most common types, including those defined by PCE/Lisp, are listed in the following table.

Type	Description
`:int`	Integer (fixnum)
`:name`	Name (Lisp keyword)
`:real`	Floating point number
`:object`	Any object
`:any`	Anything
`:unchecked`	No type checking
`:bool`	`@on` or `@off`
`:lisp-string`	Lisp string (PCE/Lisp)
`:lisp-cons`	Lisp cons (PCE/Lisp)
`:lisp-symbol`	Lisp symbol (PCE/Lisp)

6.4 User defined classes

The existing range of PCE classes can be extended by introducing new classes. PCE is a full fledged programming language, although most users do not need to be aware of this. Unfortunately, creating a new PCE class is relatively difficult and should be avoided unless absolutely necessary. PCE/Lisp defines a number of additional PCE classes (i.e. lisp-symbol, lisp-string and lisp-cons), the mechanism can be useful. This section describes the syntax of a set of macros to ease the specification of a new class. The contents of this section will only appeal to PCE veterans.

6.4.1 User defined behaviour

A class consists of a collection of attributes (instance variables) and a set of behaviour. Classes without behaviour are called storage classes. See Section 6.2 [Storage classes], page 29.

As an example we extend an existing class with new behaviour. The class is box (graphical rectangles) and the new behaviour is square (to turn the rectangle into a square). The functions `pce-send` and `pce-get` invoke send and get behaviour respectively. The behaviours are a protocol between the caller and the executor of the message. A behaviour is either implemented as an attribute or

as a piece of code called a method. Before defining a new behaviour that invokes code we need to define a corresponding method.

The square send behaviour takes one argument: the border size of the square in pixels. It should change both the width and height to the given size. Schematically we want the following:

```
(pce-send box :square 15)
  => (pce-send box :width 15)
  => (pce-send box :height 15)

(pce-get box :square)
  => (pce-get box :width)       ; assuming it is a square
```

The get method is given in the example below. Most of the arguments of the `defpcegetmethod` macro will be obvious. The `:returns` keyword argument indicates the type of the value to be returned (see Section 6.3 [PCE argument types], page 30 for details). The :body keyword argument is a list of instructions to be carried out. All of these instructions have an exclamation mark as the start of their name (no plain Lisp code may be used) and they all expand to a PCE instruction. `!get-self` performs a `pce-get` on the recipient of the method, note that methods are defined on a class and it is therefore impossible to refer to any specific instance.

```
(defpcegetmethod :box :square           ; Class and behaviour
  :documentation "Get size of a square" ; Documentation
  :arguments ()                         ; No arguments
  :returns :int                         ; Type of return value
  :body '((!get-self :width)))          ; Code to execute
```

The send method is similar, see below. It takes one argument, the size of the square (type integer). The `size` in the argument specification is a symbolic name valid within the scope of the body of a method definition.[1]

```
(defpcesendmethod :box :square          ; Class and behaviour
  :documentation "Set size of a square" ; Documentation
  :arguments '((size :int))             ; Argument name and type
  :body '((!send-self :width size)
          (!send-self :height size)))
```

6.4.2 Creating a PCE class

Creating a PCE class involves the definition of the name of the class, the super class and all slots and their types. The definition of class square is shown below.

---

[1] Names of arguments are globally replaced by the `defpcegetmethod` and `defpcesendmethod` macros. The programmer has to ensure the names are unique with respect to the function calls. It is, for example, a bad idea to give an argument the same name as a function called within the body. The body changes the width and height of the recipient, giving the size as an argument.

```
(defpceclass :square
  :super :box
  :locals '((:size :int)))

(defpcesendmethod :square :initialise
  :documentation "Create square from a size"
  :arguments '((size :int))
  :body '((!store :size size)
          (!send-super :initialise size size)))
```

The `:locals` keyword argument of `defpceclass` specifies the names of the instance variables of the class and their types. The names of these instance variables are significant, they can be accessed with `pce-send` and `pce-get`.

Each class has an initialise method. This method is called by `pce-new`. The initialise method is syntactically similar to a send method. Each body of a user defined class should contain a `!send-super` in order to give the super class of the current class a chance to initialise itself. For class square, initialise takes two arguments (the width and height of the box). The `!store` instruction changes the value of a slot.

Appendix A Glossary

attribute
A named element of the internal state of an object. Also called slot, instance variable.

behaviour
A named operation an object can respond to. See send behaviour and get behaviour.

class
The definition of a collection of objects.

get behaviour
A named operation an object that returns a value, implemented by the function `pce-get`. A get behaviour is normally side effect free, that is it only returns a value and never changes the internal state of an object.

message
An object that invokes send behaviour.

method
The code that implements a behaviour (see Section 6.4.1 [User defined behaviour], page 31).

object (PCE object)
An entity that can be manipulated by sending it a message. Objects are always instances of a class. Each object has a unique identifier which, in Lisp, is either an integer (object reference) or a symbol (object assoc).

object assoc
An object with a symbolic reference. Such an object succeeds on `pce-assoc-p` if it is not a special object (see special objects).

object reference
An object with a numeric reference. Such an object succeeds on `pce-reference-p`. PCE/Lisp prints such objects as

```
@12345 / CLASS
```

where *12345* is the unique number associated with the object and *CLASS* the name of the class of which the object is an instance.

obtainer
An object that invokes get behaviour and the replaces itself by the result of invoking that behaviour.

selector
The name of a behaviour.

send behaviour
A named operation on an object, implemented by the function `pce-send`. A send behaviour normally has a side effect, changing the internal state of the object and/or producing some other effect (e.g. modifying the contents of the screen).

special objects

PCE defines several special objects. It is not possible to send messages to these objects, they can only be used as arguments.

Appendix B Function summary

This section lists all functions defined by the PCE/Lisp interface. The functions reside in the `PCE` package.

B.1 Object representation

`@` *identifier* Macro

Read macro which converts the Lisp notation for PCE objects into the internal PCE/Lisp notation. The macro evaluates to `(pce-at identifier)`.

The *identifier* must either be an integer or a symbol.

The Lucid Common Lisp debugger does not evaluate macro's, and the `@` macro can therefore not be used inside the debugger, use the full form `pce-at` instead.

Explicit reference to the Lisp representation of the objects is not necessary. See `pce-at`, `pce-object-id`, `pce-object-p`.

The definition of the `@` macro modifies the Lisp readtable. When other code part of an application requires the `@` symbol to be used as a different macro-character, the readtable `pce::*standard-readtable*` may be used. The value of this variable is the standard readtable before the definition of the PCE `@` macro.

pce-at *identifier* Function

Similar to the `@` macro but defined as a function. Returns the Lisp representation of a PCE object. In contrast with the `@` macro a named PCE object must be quoted.

pce-object Defstruct

A Lisp defstruct defining PCE object references. The defstruct defines a single slot (called `id`) for the identifier and a print function that prints object references as "Object / Class".

The following table lists the specialisations and when these are used. Applications can refine `pce-reference`.

```
Defstruct          Purpose              Example

pce-object         PCE objects          ---
|   pce-reference  numeric references   @123456
|   pce-assoc      symbolic references  @pce, @lisp
|   pce-special    special objects      @on, @default
```

pce-reference Defstruct

A Lisp defstruct defining numeric PCE object references.

pce-assoc Defstruct

A Lisp defstruct defining symbolic PCE object references.

pce-special Defstruct

A Lisp defstruct defining special PCE objects. All instances of `pce-special` are predefined by PCE/Lisp. The print function of `pce-object` is redefined such that no class is printed.

pce-object-p *object* Predicate

Succeeds if *object* is syntactically a PCE object. Use `pce-exists-p` to determine whether the object actually exists.

pce-reference-p *object* Predicate

Succeeds if *object* is syntactically a numeric object reference.

pce-assoc-p *object* Predicate

Succeeds if *object* is syntactically a symbolic object reference. A symbolic reference is created by providing `pce-new` with the `:name` keyword argument.

An existing object can be given a symbolic name with

```
(pce-send object :name-reference :new-or-different-name)
```

Note that the object reference ("new-or-different-name") must be a Lisp keyword.

pce-special-p *object* Predicate

Succeeds if *object* is a special object. See under "Object Browser" in the PCE on-line manual for the available special objects and their purpose.

pce-exists-p *object* Predicate

Succeeds if *object* exists as a PCE object and fails otherwise. For each class in PCE a predicate called `pce-`*class*`-p` exists that tests whether the single argument is an instance of the class or one of its sub-classes. For example, `pce-point-p` only succeeds when the argument is an instance of class point, or one of the sub-classes of class point.

pce-object-id *object* Predicate

Returns the identifier (either a fixnum or a symbol) of *object*. Symbolic object names are always interned in the `PCE` package.

B.2 Virtual machine instructions

pce-new [`:name` *name*] *class arguments...* Function

Creates an instance of the given *class*. If the `:name` keyword is specified a symbolic object reference with *name* is returned, otherwise a numeric object reference is returned.

`pce-new` can fail, returning `NIL` when PCE cannot create an instance of *class* (for example because the class does not exist, the wrong number of arguments are provided, or the arguments are of the wrong type). On failure, PCE will normally either print an error message or enter the PCE tracer.

The name is given as a Lisp symbol, the class as a Lisp keyword. The number of arguments and their types depends on *class* (as specified in the PCE manual). The maximum number of arguments is specified by the variable `pce::*pce-max-args*` (current value is 100).

If the :name keyword is given, the object is locked against being freed by the PCE garbage collector. Otherwise the object is potentially garbage collectible.

For each class in PCE a function called `pce-`*class* is defined. This function creates an instance of the given class. For example,

```
(pce-point 10 10)
```

is functionaly equivalent to:

```
(pce-new :point 10 10)
```

See also `pce-make`, `pce-global`, `pce-register`.

pce-send *object selector arguments...* Function

Invokes behaviour named *selector* on the given *object*. The object itself is returned when the behaviour was successfully invoked, otherwise `NIL` is returned.

The object must be an existing PCE object (on which `pce-exists-p` succeeds), or any of the Lisp types PCE understands. The selector must be a Lisp keyword. The number of arguments and their type depends on the behaviour to be invoked. PCE can enter the tracer or print an error message when the number of arguments or their type is incorrect. The maximum number of arguments is given by the Lisp variable `pce::*pce-max-args*`.

The first argument may be of any type recognised by PCE/Lisp. The following examples are all legal (although not necessarily meaningful):

```
> (pce-send @nil :free)
NIL

> (pce-send 4 :plus 4)
4

> (pce-send :hello :larger :world)
NIL
```

A description of the behaviour of all classes can be found in the PCE manual. A behaviour can be defined on the object itself, on the class of the object, or the super class of the object (recursively). Note that the PCE manual uses underbars, which

should be replaced by hyphens in PCE/Lisp.

pce-get *object selector arguments...* Function

Values can be obtained by the function `pce-get`. The definition and arguments are identical to those of `pce-send`. The return value is computed by the behaviour.

The return value `NIL` normally means that `pce-get` failed for some reason. However, `NIL` can also be a proper return value:

```
> (setf sh (pce-sheet))

> (pce-send sh :zap nil)

> (pce-get sh :zap)
NIL
```

The first argument may be of any type recognised by PCE/Lisp. The following examples are all legal:

```
> (pce-get @nil :class)
@CONSTANT-CLASS / CLASS

> (pce-get 4 :plus 4)
@3500570 / NUMBER

> (pce-get :|hello| :upcase)
:HELLO
```

B.3 Primary functions

The Lisp functions in this section are mostly wrappers around the PCE virtual machine instructions. Most of these functions are very convenient or necessary to circumvent peculiarities of combining PCE and Lisp.

pce-make *class arg-list* Function

Like `pce-new` but takes the initialising arguments in a list.

```
> (pce-make :point (list 10 20))
```

pce-global *class arguments...* Function

Like `pce-new` but locks it against being freed implicitly when the reference count drops to zero (see Section 3.5 [Memory management], page 13).

pce-send-list *object selector list* Function

Calls `pce-send` on all elements of list. If one of the send's fails, `pce-send-list` also fails immediately.

pce-get-string *object selector arguments...* Function

Equivalent to `pce-get` except that the value is returned as a Lisp string.

When the value cannot be converted to a Lisp string, a warning is printed on the console and the unconverted value is returned.

pce-get-as *type object selector arguments...* Function

Equivalent to `pce-get` but converts the value to the type given by the first argument. The type is given as a Lisp keyword and must be a type recognised by PCE (the conversion is performed by PCE not by PCE/Lisp). The following types are relatively frequently used: :lisp-string (see `pce-get-string`), :int, :real, :file (converts file name to a file object).

pce-class-name *object* Function

Returns the *class-name* of the *object* as a Lisp keyword.

B.4 Debugging functions

pce-reset Function

Resets the internal state of PCE to top-level. Calling this function may be necessary when a program has been aborted, for example when returning to Lisp top-level while inside a call back from PCE.

Explicitly calling `pce-reset` is not necessary after the `abort` command in the PCE tracer. It is called automatically.

There are two good heuristics when PCE needs to be reset. The first is when PCE is showing interactive feedback in a window while Lisp is at top-level (examples of interactive feedback are greying an active button or highlighting in a menu bar). The second heuristic is by looking at the Lisp backtrace. If the backtrace contains `PCE-CALL-BACK` and you need to abort to Lisp top-level, `pce-reset` is necessary.

When `pce-reset` is not called when necessary, user interface actions do not respond.

pce-check-globals *argument...* Function

Runs an internal consistency check on all global (i.e. named) objects and the *argument*s. `pce-check-globals` prints a warning on the console for each internal inconsistency found. This is a rigorous way to find inconsistencies (for example wrong value in an object's slot) in the PCE data base. Note that `pce-check-globals` only checks the given objects and all objects accessible from these objects.

B.5 Lisp to PCE and vice versa

lisp-to-pce *expression* Function

Takes a Lisp *expression* and returns it as a PCE object. The Lisp expression may contain conses, symbols, keywords, strings, integers, floats, and PCE object references. The *object* returned may be stored inside PCE.

pce-to-lisp *object* Function

Takes an *object* and returns it as a Lisp expression. The *object* has normally been created by `lisp-to-pce`.

pce-chain-list *chain* [*return-as*] Function

Class chain is PCE's idea of a list. Chains are frequently used, and `pce-chain-list` converts a PCE chain to a Lisp list.

There are several special cases. If the argument is `NIL` or `@nil`, then `NIL` is returned. This makes it possible to use `pce-chain-list` if something is either `@nil` or a chain (this sometimes happens internally in PCE).

The optional *return-as* argument specifies the type to which the elements are converted, the default is no conversion (type :unchecked). If the list consists of PCE strings, the default behaviour is:

```
> (setf ch (pce-chain (pce-string "Hello")
                      (pce-string "world!")))

> (pce-chain-list ch)
(@3380066 / STRING @3380114 / STRING)
```

With conversion the result is, in this case, more interesting:

```
> (pce-chain-list ch :lisp-string)
("Hello" "world!")
```

`pce-chain-list` returns `NIL` when the first argument is `NIL` or `@nil`. It also returns `NIL` (i.e. the empty list) when the first argument is a chain with no elements.

pce-get-list *object selector* Function

When `(pce-get object selector)` returns a chain, `pce-get-list` turns the chain into a list and returns it. In other words, `pce-get-list` is short for:

```
(pce-chain-list (pce-get object selector))
```

pce-funcall *function argument...* Function

Creates a *message* that calls the Lisp *function* with the given *arguments*. Short for:

```
(pce-message @lisp function argument...)
```

pce-apply *function argument-list* Function

The same as `pce-funcall` but with the arguments given as a list.

pce-eval *expression* Function

Creates a *message* which evaluates *expression* when executed.

B.6 User defined classes

pce-storage-class *name* [`:super` *super*] [`:slots` *slots*] Function

`pce-storage-class` is used for application specific information types. The function defines a PCE class with a predefined number of slots. User defined classes that also require behaviour should use the more complex `defpceclass`, `defsendmethod` and `defgetmethod` functions. It is obviously possible to use `pce-storage-class` to define a class and the use `defgetmethod` and `defsendmethod` to add behaviour.

The arguments of `pce-storage-class` are the *name* of the class (as a Lisp keyword), the name of the *super* class and a list describing the *slots* (or instance variables). The format of the each element in the list of slots is:

(*name type access*)

where *name* is the name of a slot, *type* the type of the slot and *access* the access rights for the PCE virtual machine of the slot. *access* is one of: code:send (`pce-send` may be used on the slot), `:get` (`pce-get` may be used), `:both` (both `pce-send` and `pce-get` may be used), `:none` (no access).

defpceclass *class-name* [`:super` *super*] *arguments...* Function

Creates *class* as a new user defined PCE class. The class is called *class-name*, the super class is given by *super-name*. A trivial example is:

```
> (defpceclass :my-dialog :super :dialog)
@MY-DIALOG-CLASS / CLASS
```

Without any further definitions, the new class my-dialog is functionally equivalent to class dialog. `defpceclass` defines the class creation and type checking functions `pce-my-dialog` and `pce-my-dialog-p`.

`defpceclass` accepts the following keyword arguments:

`:super` *super-class-name* `[:object]`
: The name of the super class of the class defined. It defaults to the root of the PCE class hierarchy.

`:locals '((`*name type*`) ...) [NIL]`
: A list of instance variables for the class. The value of these variables can be set with `pce-send` and retrieved with `pce-get`. The *name* of a variable is a Lisp keyword, the *type* restricts the values the variable may take.

`:slots '((`*name type*`) ...) [NIL]`
: A list of instance variables which are also used as initialising arguments when

creating an instance of the class. Essentially the definition of the existing PCE class point is:

```
(defpceclass :point :slots '((:x :int) (:y :int)))
```

For most classes creating an instance is performed by the initialise method defined separately. By default the initialise method is inherited from the super class.

`:keep-instances T | NIL [NIL]`
If `:keep-instances` is `T` the class keeps track of all it's instances. The instance chain is available through instances get behaviour.

`:create-method` *method* `[NIL]`
When specified, *method* (a keyword) is called on each instance created.

`:delete-method` *method* `[NIL]`
When specified, *method* (a keyword) is called on each instance deleted.

`:resources '((`*name type default*`) ...) [NIL]`
Defines a set of resources for the given class. The arguments are the name, type and default value respectively. Example:

```
:resources '((:width :int 200)
             (:height :int 100))
```

When a class is redefined with a different definition of instance variables PCE prints an error message and refuses to create the class. The only remedy is to exit PCE/Lisp and start again.

defpcesendmethod *class-name selector arguments...* Function

Defines a send method for *class-name*. The method is available with *selector*. The keyword arguments are:

`:documentation` *string* `[""]`
Documentation associated with the method, consult the summary line in the PCE manual for inspiration.

`:arguments '((`*symbol type*`) ...) [NIL]`
Defines the arguments for the method. Each *symbol* is the name of an argument, each *type* is the corresponding argument's type. The names of the symbols are arbitrary but should be chosen such that a global replacement of all symbols in the body is possible. That is, a symbol name should not be the same as the name of function inside the body.

`:body '(`*form ...*`)`
Each *form* in the body of a method definition should expand to an instance of PCE class code. That is, the body consists of PCE code objects. The forms are executed in the sequence they are written down. If a form fails, execution immediately stops and the entire method fails.

The most convenient way to write a body is to use the macros starting with a !. These macros expand to the appropriate PCE code instances.

```
(defpcesendmethod :pce :hello-world
  :documentation "Prints hello world!"
  :body
    '((!send-self :write-ln "Hello world!")))

> (pce-send @pce :hello-world)
Hello world!
```

defpcegetmethod *class-name selector arguments...* Function

Defines a get method for the given *class-name*. The method is accessible through the *selector* get behaviour.

`:documentation` *string* `[""]`
: Documentation associated with the method, normally a one-liner.

`:arguments '((`*symbol type*`) ...) [NIL]`
: See under `defpcesendmethod`.

`:returns` *type* `[:any]`
: The return type of the method is determined by *type*.

`:body '(`*form* `...) [NIL]`
: See also under `defpcesendmethod`. If all forms succeed the value of the last form is returned.

B.7 Instructions for user defined classes

The real difficulty of user defined classes in PCE starts when the method cannot be expressed by a straightforward sequence of sends and gets. PCE provides loops, conditionals and other control instructions as objects. Below is an overview of all instructions that are allowed as part of the body of a method.

!let (*symbol...*) *form...* Macro

Defines the *symbol*s to be local variables accessible in all *form*s. Each *symbol* can be assigned to with `!setq`.

!setq *symbol expression* Macro

Assigns the value of *expression* to *symbol*. The symbol must be declared with `let` or the name of an argument of the current method.

!store *slot value* Macro

Change the value of *slot* to *value* on the current object. The current object is the receiver of the behaviour that invoked the method. *slot* is the name of instance variable declared with `:locals`, `:slots` or inherited from the super class. `store` may

be necessary when `send-self` would generate unwanted side effects inside a method definition.

!fetch *slot* Macro

Computes the *value* of the given *slot* on the current object and replaces the expression by the value. For example:

```
(!store :height (!fetch :width))
```

could be used to set the height identical to width.

!new *class args...* Macro

Creates an instance of the given *class* and replaces the expression by the object created.

!send *object selector args...* Macro

Invoke send behaviour called *selector* on *object* with the given arguments. This is identical to `pce-send` in ordinary PCE code, but it may be difficult to find the right object. When the behaviour fails the surrounding block of instructions also fails.

!send-self *selector args...* Macro

Like `!send` but invokes the behaviour on the recipient of the method.

!send-super *selector args...* Macro

Like `!send` but starts the search on the super class of the recipient.

!get *object selector args...* Macro

Invoke get behaviour on the given *object*. The expression is replaced by the *value* computed.

!get-self *selector args...* Macro

Like `get` but executes on the recipient itself.

!if *condition* [*then*] [*else*] Macro

Like `if` in Lisp. The *condition* is evaluated and if it succeeds the *then* branch is taken, otherwise the optional *else* branch is taken. The result of the entire expression is the value of the branch. The default for both *then* and *else* is to succeed, omitting both is therefore equivalent to ignore the result of *condition*.

!when *condition* [*then*] [*else*] Macro

Returns the result of *then* when condition succeeds, *else* otherwise.

!while *condition body* Macro

While *condition* succeeds *body* is executed.

!not *condition* Macro

Succeeds when *condition* fails and vice versa.

!progn Macro

If the body of a get method consists of multiple forms, `progn` ensures that the value of the last form is returned. Using `let` in a get method implies a `progn`.

!or *args...* Macro

Succeeds when at least one of the arguments succeeds. `or` without any arguments is equivalent to failing.

!and *args...* Macro

Succeeds when all of the arguments succeed. `and` without any arguments is equivalent to succeeding.

!eq *left right* Macro

Succeeds when the two arguments are identical.

!neq *left right* Macro

Succeeds when the two arguments are different.

!> *left right* Macro

Succeeds when *left* is larger than *right*.

!< *left right* Macro

Succeeds when *left* is smaller than *right*.

!>= *left right* Macro

Succeeds when *left* is larger or equal than *right*.

!<= *left right* Macro

Succeeds when *left* is smaller or equal than *right*.

!+ *left right* Macro

Returns adding *left* and *right*.

!- *left right* Macro

Returns subtracting *right* from *left*.

!* *left right* Macro

Returns multiplying *left* and *right*.

!/ *left right* Macro

Returns dividing *left* and *right*.

B.8 Utility functions

The functions in this section are of secondary importance. They have been introduced because they capture a frequent notion which would otherwise have been reimplemented by many applications.

pce-printed-p *object object* Predicate

Succeeds when the two objects are `equal` when printed by Lisp. This is a rather expensive way to ignore differences between the classes of the two objects. In general, the application should know whether the objects are represented as PCE strings, Lisp strings, symbols or keywords.

pce-create-attribute *object attribute value* Function

Creates a new attribute for the object. The name of the attribute is *attribute* (a Lisp keyword), and sets the initial value. New attributes may be associated with any PCE object, care should be exercised that the attribute name is not already taken.

pce-parts *object* Function

Given that the object is a graphical object, returns all its graphical parts as a list. This function is convenient when debugging and user interfaces.

pce-whole *object* Function

The opposite of `pce-parts`.

pce-hierarchy *object* Function

Prints a hierarchy on the console of the graphical objects contained in *object*. Handy for debugging graphical applications.

pce-name-reference *object name* Function

Makes *name* the symbolic object reference for *object*. *name* may be either a keyword (preferred) or a Lisp symbol.

pce-all-globals Function

Returns a chain containing all objects that have a named reference.

pce-gc *expression* Macro

Calls *expression* (any unquoted Lisp code) and returns the *value* returned by expression, finally cleans up the unreferenced objects created as a result of calling *expression*. Calls to `pce-gc` may be nested, i.e. *expression* may contain calls to `pce-gc`.

pce-memory-usage *expression* Macro

Calls the Lisp *expression* and returns the number of *bytes* it consumed in PCE. This is a simple aid in determining whether a certain function consumes a lot of PCE memory and should be embraced by a `pce-gc`. Note that the value returned can also be negative, meaning that PCE has reclaimed memory as a result of evaluating *expression*.

pce-register *object* Function

Registers *object* as if it where a named object reference. All the advantages of named object references are then available (e.g. compares `eq`). The *object* is also protected against being automatically garbage collected.

pce-unregister *object* Function

Unregisters *object*, the opposite of `pce-register`. `pce-unregister` is normally executed just before destroying the object.

Appendix C Performance

PCE/Lisp has been engineered such that optimal performance is achieved while retaining the correctness of the interface and generating appropriate error messages on incorrect usage of the primary interface functions.

The following table lists performance figures for the most frequently used PCE/Lisp functions. These figures should only be considered to be an indication of the performance of the interface. The columns give the number of calls per second.

The figures where obtained using SunCommonLisp 4.0.1 on a SparcStation 2. None of the functions generate garbage on the Lisp heap.

Benchmark	LCL	Description
(pce-point 10 12)	2700	Creating objects
(pce-send @pce :succeed)	7200	Zero-argument send
(pce-send @pce :trace :user)	4600	Single argument send
(pce-get @pce :trace)	6000	Zero-argument get
(pce-at 'pce)	4500	Object reference evaluation
(pce-exists-p @pce)	62500	Object existence

Appendix D Error messages

PCE/Lisp tries to detect all errors that violate the syntactic rules of the PCE/Lisp functions. Real programming errors, for example sending a message to the wrong object, are not detected. PCE itself may also signal an error message, for example if the number of arguments to a behaviour is incorrect. Errors detected by PCE enter the PCE tracer.

D.1 Recovering from errors

When PCE/Lisp detects an error the Lisp debugger is entered and an appropriate error message is displayed. The programmer can only locate the source of the error (using the Lisp stack), abort to Lisp top level, correct the error, and try again.

Examining the stack trace is simple. Ignore all functions that have names starting with `PCELISP-`. These functions implement PCE/Lisp. An example to illustrate a typical error.

```
> (pce-send @foo :zap)          ; @foo does not exist

>>Error: PCE-SEND: @foo / freed does not exist

PCE::PCELISP-SEND-ERROR:
Original code: (NAMED-LAMBDA PCE::PCELISP-SEND-ERROR (PCE::PS PCE::RECEIVER
PCE::SELECTOR) (BLOCK PCE::PCELISP-SEND-ERROR (LET # #)))
   Required arg 0 (PS): #<Foreign-Pointer ADBC70 (:POINTER PCE::PCELISP-
STATUS)>
   Required arg 1 (RECEIVER): @foo / freed
   Required arg 2 (SELECTOR): :ZAP
:A  0: Abort to Lisp Top Level

->
```

The error message is at the top: `PCE-SEND: @foo / freed does not exist`. A stack backtrace gives:

```
-> :b                              ; Ask for stack backtrace
PCE::PCELISP-SEND-ERROR <- PCE-SEND <- EVAL <- SYSTEM:ENTER-TOP-LEVEL
```

The internal function `pce::pcelisp-send-error` generated the error message, as a result from incorrect arguments to `pce-send`. In this case you can abort to Lisp top level, correct the error and try again.

When the function `PCE-CALL-BACK` is on the stack trace the recovery procedure is slightly more complicated. The function `pce-call-back` is called as a result of PCE calling back to Lisp (see Chapter 5 [Call back], page 23). An example:

```
> (defun zap-ahoy () (format 12))       ; Incorrect

> (setf msg (pce-message @lisp 'zap-ahoy))
```

```
> (pce-send msg :forward)

>>Error: FORMAT called with 1 argument, but at least 2 arguments are
required

FORMAT:
   Required arg 0 (DESTINATION): 12
   Required arg 1 (CONTROL-STRING): ((#<Return-From-Closure 71A> . ZAP-AHOY)
NIL)
   Rest arg 2 (ARGUMENTS): NIL
:C  0: Supply missing arguments
:A  1: Abort to Lisp Top Level

->
```

The Lisp backtrace now gives:

```
-> :b                    ; Backtrace
FORMAT <- ZAP-AHOY <- PCE-CALL-BACK <- PCE::PCELISP-C-SEND
<- PCE-SEND <- EVAL <- SYSTEM:ENTER-TOP-LEVEL
```

Recover by first putting the `PCE-CALL-BACK` frame at the top of the stack and then return the value 1 (one) to PCE/Lisp.

```
-> :n pce-call-back             ; Go to PCE-CALL-BACK frame
PCE-CALL-BACK:
   Required arg 0 ( framevar ): -33555252
   Required arg 1 ( return-framevar ): -33555254
-> :r 1                         ; Return 1 to indicate success
>                               ; Back in Lisp top level
```

It is possible that PCE itself gets into an incorrect state because of the recovery described above (for example because it was in the middle of handling a mouse click). If the windows no longer respond try to reset PCE with the function `pce-reset` (no arguments).

D.2 Error message descriptions

Below are the PCE/Lisp interface error messages. The behaviour specific error messages should be obvious from the code that called it. Some errors are generic (i.e. they could result from several functions), they are prefixed with "PCE/Lisp". Otherwise, the name of the function is printed first. If an error should never occur it is labelled with "INTERNAL", this always means either there is a bug in PCE/Lisp itself or the system is totally garbled. When this occurs, rerun the program and if the error repeats report the problem. Words between double quotes are replaced by a value.

PCE-NEW At least one argument required

Insufficient number of arguments to `pce-new`.

PCE-AT Argument ("value") not an integer, symbol or string

Says it all.

PCE/Lisp (INTERNAL) Unknown error ("error-code")

Communication between PCE and Lisp was possibly corrupted. Exit from the debugger, and then try `pce-reset` to correct, or restart the image if this doesn't work. Inform the developer if this problem repeats consistently.

PCE/Lisp (INTERNAL) Unknown return type ("return-type")

See under unknown error above.

PCE/Lisp Argument type not recognised ("argument")

An argument to one of the PCE/Lisp functions is not of type recognised by the PCE/Lisp interface. The admissable types are given in Chapter 4 [Type conversions], page 17.

PCE/Lisp Keyword or symbol expected ("argument")

PCE expected a keyword or a symbol, for example as the selector of a message.

Appendix E Debugging facilities

PCE/Lisp is a mixed environment. Both PCE and Lisp have their own virtual machine interpreters, tracing facilities and error handling mechanisms. In order to aid the debugging of host language (e.g. Lisp) programs, PCE provides the ability to activate the host language tracer from the PCE tracer.

Here is a sample session (with Lucid Common Lisp):

```
> (pce-new :point "Hello" 12)
[PCE WARNING: POINT ->INITIALISE: Argument 1 should be a [INT]
        in: M @3500539/POINT ->INITIALISE: @3500569/"Hello", 12]
PCE:  2 FAIL: M @3500539/POINT ->INITIALISE: @3500569/"Hello", 12 ? b
        ;;; User typed "b" to break into Lisp
>>Break: Break request from PCE -- Use :C [ONTINUE] to return

PCE::PCELISP-CALL-BACK:
   Required arg 0 ( framevar ): -67110056
   Required arg 1 ( return-framevar ): -67110058
:C  0: Return from Break
:A  1: Abort to Lisp Top Level

-> :c   ;;; After inspecting the source of the problem
        ;;;   user returns control to PCE
Return from Break
PCE:  2 FAIL: M @3500539/POINT ->INITIALISE: @3500569/"Hello", 12 ?
```

The PCE tracer has the following commands for interacting with the host language:

`a` (abort)
: Returns control to the host language top-level and resets PCE (with `pce-reset`). This option should only be used when continuing is not possible.

`b` (break)
: Temporarily switches control to the host languages and generates a new interactive top level. In Common Lisp it is only possible to enter the debugger. Control is returned to PCE with the continue command of the Lisp debugger.

`gh` (goals host)
: Temporarily switches control to the host language debugger. The intention of this command is to print a backtrace in the host language. Given that printing a backtrace is not part of Common Lisp, the user has to issue the implementation specific backtrace command. After inspecting the backtrace, control is returned to PCE with the continue command of the Lisp debugger.

A list of PCE tracer commands is displayed by typing `?`.

Appendix F Implementation dependent features

F.1 Harlequin LispWorks (LW)

The interface between PCE and LispWorks was developed using LispWorks 3.1. During the implementation it turned out that a number of patches to this LispWorks implementation where required. Later versions of LispWorks should incorporate these patches.

PCE only runs under the LispWorks graphical environment. See the LispWorks documentation for details.

Users have reported problems when running applications that use both PCE and the built-in X libraries. Whether these problems are caused by PCE or LispWorks itself is unknown. Contact support at Harlequin for details.

F.2 Lucid Common Lisp (LCL)

Lucid Common Lisp is the Lisp implementation we use for development.

The interface between PCE and Lucid Common Lisp is based on a generic interface provided by PCE. It may be obvious that the LCL interface depends on many intricate features of LCL itself.

This chapter gives a number of hints and suggestions for developing and debugging PCE/Lisp applications in LCL 3.0 and LCL 4.0.[1]

For some reason the LCL debugger does not accept the `@` macro, using it results in an error being signalled by the debugger. Loading files while in the debugger also results in errors.[2]

When there is a need to refer to symbolic object references in the debugger use `pce-at` rather than the `@` macro. Thus:

```
-> (pce-at 'pce)
@pce / PCE
```

---

[1] PCE/Lisp runs unmodified under both of these implementations. This is easily explained by the fact that the foreign function interface has not changed between these versions. PCE functions are somewhat faster under LCL 4.0.

[2] When this happens it is better to quit LCL and start again. The results are completely unpredictable.

Appendix G Concept index

:

:done behaviour 13
:free behaviour 13

@

@lisp 23

A

abort (PCE tracer command) 57
Argument types 30
Arguments of calls from PCE to Lisp 24
Attribute, PCE class 29
Attributes on objects 29

B

Behaviour (get) 7
Behaviour (send) 6
Behaviour defined by the user 31
Block, PCE class 26
break (PCE tracer command) 57

C

Call back 23
Changes, compared to earlier versions 1
Class 5
Class attribute 29
Class block 26
Class defined by the user 31
Class lisp-cons 19
Class lisp-string 19
Class lisp-symbol 18
Class name 18
Class number 17
Class real 17
Class sheet 29
Class specific creator functions 8
Class string 19
Class, creating a 32
Class, user-defined storage 29
Coercion of types 20
Computing arguments 25
Conses 19
Conversion, explicit 20
Converting types between PCE and Lisp 17
Create PCE object (pce-new) 5
Creating a PCE class 32
Creator function for a class 8

D

Debugging facilities 57
Defstruct pce-assoc 11
Defstruct pce-object 11
Defstruct pce-reference 11
Defstruct pce-special 11
Description of error messages 54
Destroying objects 13
Dialog boxes 10
Documentation, PCE 3

E

Earlier versions, changes 1
Equality of PCE objects 12
Error message descriptions 54
Error messages 53
Errors, recovering from 53
Example PCE programs 8
Existence of an object (pce-exists-p) 13
Explicit conversion 20

F

Feature, PCE 2
Freeing objects 13
Functions, name of PCE 3

G

Get behaviour 7
goals host (PCE tracer command) 57
Graphics, interactive 9

H

Harlequin LispWorks 59

I

Informers 8
Interactive graphics 9

K

Keywords 18
Keywords vs. symbols and strings 21

L

Large applications 29
LCL 59
Lexiographic conventions 17
Lisp functions, called from PCE 23
Lisp object (`@lisp`) 23
LispWorks (Harlequin) 59
Lists 19
Lucid Common Lisp 59

M

Memory management 13
Message, sending a (`pce-get`) 7
Message, sending a (`pce-send`) 6
Multiple messages 26

N

Name, of PCE/Lisp executable 1
Names of PCE functions 3
Non-primitive types 19
Numbers 17

O

Object 5
Object attributes 29
Object, activating an 6
Object, existence (`pce-exists-p`) 13
Object, obtaining value from 7
Objects and object references 13
Objects, creating 5
Obtainers 25
Obtaining values from objects 7
On-line PCE documentation 3

P

Package, PCE 2
Passing arguments (`@arg1`) 24
PCE documentation 3
PCE feature 2
PCE function names 3
PCE object, equality of 12
PCE package 2
PCE version 1
`pce-`*class* 8
`pce-`*class*`-p` 8
PCE/Lisp, name of executable 1
Performance of PCE/Lisp 51
Primary objects 12
Program, PCE/Lisp 1

R

Recovering from errors 53
Registering primary objects 12
Release notes 1
Representation of PCE objects in Lisp 11

S

Sample PCE programs 8
SCL 59
Secondary objects 12
Send behaviour 6
Sending a message 6
Sheet, PCE class 29
Storage classes 29
Strings 19
Strings vs. keywords and symbols 21
Sun Common Lisp 59
Support for large applications 29
Symbols 18
Symbols vs. keywords and strings 21

T

Tutorial 5
Type coercion 20
Type cons 19

Type conversions 17
Type fixnum 17
Type float 17
Type keyword 18
Type list 19
Type number 17
Type overview 21
Type string 19
Type symbol 18
Types of PCE arguments 30
Types, non-primitive Lisp 19

U

User defined behaviour 31
User defined classes 31

V

Version, of PCE 1
Versions, earlier 1

X

‘`xpceclos`’ 1
‘`xpcelisp`’ 1

Appendix H Function variable index

!

!* 47
!- 47
!/ 47
!+ 47
!> 47
!>= 47
!< 47
!<= 47
!and 47
!eq 47
!fetch 46
!get 46
!get-self 46
!if 46
!let 45
!neq 47
!new 46
!not 46
!or 47
!progn 47
!send 46
!send-self 46
!send-super 46
!setq 45
!store 45
!when 46
!while 46

@

@ 37

D

defpceclass 43
defpcegetmethod 45
defpcesendmethod 44

L

lisp-to-pce 42

P

pce-all-globals 48
pce-apply 42
pce-assoc 38
pce-assoc-p 38
pce-at 37
pce-chain-list 42
pce-check-globals 41
pce-class-name 41
pce-create-attribute 48
pce-eval 43
pce-exists-p 38
pce-funcall 42
pce-gc 48
pce-get 40
pce-get-as 41
pce-get-list 42
pce-get-string 40
pce-global 40
pce-hierarchy 48
pce-make 40
pce-memory-usage 48
pce-name-reference 48
pce-new 38
pce-object 37
pce-object-id 38
pce-object-p 38
pce-parts 48
pce-printed-p 48
pce-reference 37
pce-reference-p 38
pce-register 49
pce-reset 41
pce-send 39
pce-send-list 40
pce-special 38
pce-special-p 38
pce-storage-class 43
pce-to-lisp 42

pce-unregister.....49 pce-whole.....48

Short Contents

1 PCE Common Lisp interface 1
2 Tutorial 5
3 Representation 11
4 Type conversions 17
5 Calling Lisp functions from PCE 23
6 Support for large applications 29
Appendix A Glossary 35
Appendix B Function summary 37
Appendix C Performance 51
Appendix D Error messages 53
Appendix E Debugging facilities 57
Appendix F Implementation dependent features 59
Appendix G Concept index 61
Appendix H Function variable index 65

Table of Contents

1 PCE Common Lisp interface 1

1.1 Version 1
1.2 Release notes 1
1.3 Packages, features and naming conventions 2
1.3.1 PCE package 2
1.3.2 PCE feature 2
1.3.3 Names of PCE functions 3
1.4 Organisation 3
1.5 Other PCE documents 3
1.6 Acknowledgements 3

2 Tutorial 5

2.1 Introduction 5
2.1.1 Creating objects: `pce-new` 5
2.1.2 Activating objects: `pce-send` 6
2.1.3 Obtaining values: `pce-get` 7
2.1.4 Short hand for `pce-new` 8
2.2 Sample programs 8
2.2.1 Hello world 8
2.2.2 Interactive graphics 9
2.2.3 Dialog boxes 10

3 Representation 11

3.1 PCE objects in Lisp 11
3.2 Equality of PCE objects 12
3.3 Registering primary objects 12
3.4 Objects and object references 13
3.5 Memory management 13

4 Type conversions 17

4.1 Lexiographic conventions 17
4.2 Lisp numbers 17
4.3 Lisp keywords 18
4.4 Lisp symbols 18
4.5 Lisp strings 19
4.6 Lisp conses 19
4.7 Non-primitive Lisp types 19

4.8 Type coercion 20
4.9 Explicit conversion 20
4.10 Type overview 21

5 Calling Lisp functions from PCE 23

5.1 The Lisp object 23
5.2 Passing arguments to functions 24
5.3 Computing arguments: obtainers 25
5.4 Multiple messages 26

6 Support for large applications 29

6.1 Attributes 29
6.2 PCE storage classes 29
6.3 PCE argument types 30
6.4 User defined classes 31
6.4.1 User defined behaviour 31
6.4.2 Creating a PCE class 32

Appendix A Glossary 35

Appendix B Function summary 37

B.1 Object representation 37
B.2 Virtual machine instructions 38
B.3 Primary functions 40
B.4 Debugging functions 41
B.5 Lisp to PCE and vice versa 41
B.6 User defined classes 43
B.7 Instructions for user defined classes 45
B.8 Utility functions 48

Appendix C Performance 51

Appendix D Error messages 53

D.1 Recovering from errors 53
D.2 Error message descriptions 54

Appendix E Debugging facilities 57

Appendix F Implementation dependent features 59

F.1 Harlequin LispWorks (LW) 59
F.2 Lucid Common Lisp (LCL) 59

Appendix G Concept index 61

Appendix H Function variable index 65