1 Issue 2 (PRINT-READ consistency)

Lisp reader uses `find-package` to read a symbol, and is affected by *local nicknames* of the *current package*. So in order to maintain **print-read** consistency it is required to use a correct *package prefix* - such prefix that calling `find-package` on it in the *current package* will return the symbol's *home package*.

There are several situations to consider:

1. There **is** a *local nickname* defined in the *current package* for the symbol's *home package*.

   In this case such local nickname can be used as the package prefix.

2. Symbol's home *package name* or one of its *global nicknames* is not shadowed by any *local nickname* defined in the *current package*.

   In this case that package name or global nickname can be used as the package prefix.

3. Symbol's home *package name* and all its *global nicknames* are shadowed by one of the *local nicknames* of the *current package* and there **is no** *local nickname* defined (in the *current package*) for the symbol's home package.

   It is not clear what should be done in this case (see proposals).

1.1 Example

```
(defpackage #:a (:use) (:export #:+))
(defpackage #:b (:local-nicknames (#:a #:cl)))
(let ((*package* (find-package '#:b)))
  (print 'a:+))
; => a:+ everywhere
;; but in #:b package a:+ would refer to cl:+

(defpackage #:a (:use) (:export #:+))
(defpackage #:b (:use) (:export #:+))
(defpackage #:c (:use) (:local-nicknames (#:a #:b) (#:b #:a)))
(let ((*package* (find-package '#:c)))
  (print 'a:+))
; => b:+  (sbcl, ccl, abcl)
; => a:+  (clasp, acl, ecl)
;; but in #:c package a:+ would refer to b:+
```

1.2 Currently

sbcl, ccl, abcl: print symbol with package name when all package's names and nicknames are shadowed by *current package*'s *local nicknames*.

ecl, acl, clasp: don't print with local nicknames at all.

1.3 Proposals

- The symbol must be printed using the `#.` syntax:

```
#.(cl:let ((cl:*package* (cl:find-package "KEYWORD")))
    (cl:find-symbol "BAR" "FOO"))
;; or
#.(cl:let ((cl:*package* (cl:find-package "KEYWORD")))
    (cl:intern "BAR" "FOO"))
```

Note that `#:KEYWORD` name is reserved for the `#:KEYWORD` package and cannot be used as a *local nickname* thus this expression will always evaluate to the symbol `foo::bar`.

- *Shinmera's idea.* In this case an extended `#:` syntax should be used:

```
#:(package name) and #::(package name)
```

- In this case the symbol must be printed using the `#‘` syntax for reading an expression ignoring *local nicknames* in the *current package*:

```
#‘foo:bar  and  #‘foo::bar
```

It can be implemented roughly as follows:

```
(defun |#‘-reader| (stream subchar arg)
  (declare (ignore subchar arg))
  (let* ((current-package *package*)
         (local-nicknames (package-local-nicknames current-package)))
    (loop for (nick . package) in local-nicknames
          do (remove-package-local-nickname nick current-package))
    (unwind-protect
      (read stream t nil t)
      (loop for (nick . package) in local-nicknames
            do (add-package-local-nickname nick package current-package)))))

(set-dispatch-macro-character #\# #\‘ #’|#‘-reader|)
```

It is implementation dependent whether *local nicknames* are actually removed from the *current package* or not.

- In this case the symbol must be printed unreadably (specifics are implementation dependent):

```
#<SYMBOL IN THE SHADOWED PACKAGE FOO:BAR>
#<SYMBOL IN THE SHADOWED PACKAGE FOO::BAR>
```

If `*print-readably*` is *true* must signal an error of type `print-not-readable` without printing anything.

- In this case the symbol must be printed using `:::` and `::::` syntax to lookup and intern ignoring *local nicknames* respectively:

```
foo:::bar  ; same as (cl:find-symbol "BAR" "FOO") in the #:KEYWORD package
foo::::bar  ; same as (cl:intern "BAR" "FOO") in #:KEYWORD package
```