23. Reader

23.1 Reader Concepts

23.1.1 Dynamic Control of the Lisp Reader

Various aspects of the *Lisp reader* can be controlled dynamically. See Section 2.1.1 (Readtables) and Section 2.1.2 (Variables that affect the Lisp Reader).

23.1.2 Effect of Readtable Case on the Lisp Reader

The *readtable case* of the *current readtable* affects the *Lisp reader* in the following ways:

`:upcase`
: When the *readtable case* is `:upcase`, unescaped constituent *characters* are converted to *uppercase*, as specified in Section 2.2 (Reader Algorithm).

`:downcase`
: When the *readtable case* is `:downcase`, unescaped constituent *characters* are converted to *lowercase*.

`:preserve`
: When the *readtable case* is `:preserve`, the case of all *characters* remains unchanged.

`:invert`
: When the *readtable case* is `:invert`, then if all of the unescaped letters in the extended token are of the same *case*, those (unescaped) letters are converted to the opposite *case*.

23.1.2.1 Examples of Effect of Readtable Case on the Lisp Reader

```
(defun test-readtable-case-reading ()
  (let ((*readtable* (copy-readtable nil)))
    (format t "READTABLE-CASE  Input   Symbol-name~
             ~%-----------------------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (input '("ZEBRA" "Zebra" "zebra"))
        (format t "~&:~A~16T~A~24T~A"
                (string-upcase readtable-case)
                input
                (symbol-name (read-from-string input)))))))
```

The output from `(test-readtable-case-reading)` should be as follows:

```
READTABLE-CASE     Input Symbol-name
-------------------------------------
   :UPCASE         ZEBRA   ZEBRA
   :UPCASE         Zebra   ZEBRA
   :UPCASE         zebra   ZEBRA
   :DOWNCASE       ZEBRA   zebra
   :DOWNCASE       Zebra   zebra
   :DOWNCASE       zebra   zebra
   :PRESERVE       ZEBRA   ZEBRA
   :PRESERVE       Zebra   Zebra
   :PRESERVE       zebra   zebra
   :INVERT         ZEBRA   zebra
   :INVERT         Zebra   Zebra
   :INVERT         zebra   ZEBRA
```

23.1.3 Argument Conventions of Some Reader Functions

23.1.3.1 The EOF-ERROR-P argument

Eof-error-p in input function calls controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is *true* (the default), an error of *type* **end-of-file** is signaled at end of file. If it is *false*, then no error is signaled, and instead the function returns *eof-value*.

Functions such as **read** that read the representation of an *object* rather than a single character always signals an error, regardless of *eof-error-p*, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** signals an error. If a file ends in a *symbol* or a *number* immediately followed by end-of-file, **read** reads the *symbol* or *number* successfully and when called again will act according to *eof-error-p*. Similarly, the *function* **read-line** successfully reads the last line of a file even if that line is terminated by end-of-file rather than the newline character. Ignorable text, such as lines containing only *whitespace*[2] or comments, are not considered to begin an *object*; if **read** begins to read an *expression* but sees only such ignorable text, it does not consider the file to end in the middle of an *object*. Thus an *eof-error-p* argument controls what happens when the file ends between *objects*.

23.1.3.2 The RECURSIVE-P argument

If *recursive-p* is supplied and not **nil**, it specifies that this function call is not an outermost call to **read** but an embedded call, typically from a *reader macro function*. It is important to distinguish such recursive calls for three reasons.

1. An outermost call establishes the context within which the `#`*n*`=` and `#`*n*`#` syntax is scoped. Consider, for example, the expression

```
(cons '#3=(p q r) '(x y . #3#))
```

If the *single-quote reader macro* were defined in this way:

```
(set-macro-character #\'       ;incorrect
   #'(lambda (stream char)
        (declare (ignore char))
        (list 'quote (read stream))))
```

then each call to the *single-quote reader macro function* would establish independent contexts for the scope of **read** information, including the scope of identifications between markers like "`#3=`" and "`#3#`". However, for this expression, the scope was clearly intended to be determined by the outer set of parentheses, so such a definition would be incorrect. The correct way to define the *single-quote reader macro* uses *recursive-p*:

```
(set-macro-character #\'       ;correct
   #'(lambda (stream char)
        (declare (ignore char))
        (list 'quote (read stream t nil t))))
```

2. A recursive call does not alter whether the reading process is to preserve *whitespace*[2] or not (as determined by whether the outermost call was to **read** or **read-preserving-whitespace**). Suppose again that *single-quote* were to be defined as shown above in the incorrect definition. Then a call to **read-preserving-whitespace** that read the expression `'foo<Space>` would fail to preserve the space character following the symbol `foo` because the *single-quote reader macro function* calls **read**, not **read-preserving-whitespace**, to read the following expression (in this case `foo`). The correct definition, which passes the value *true* for *recursive-p* to **read**, allows the outermost call to determine whether *whitespace*[2] is preserved.
3. When end-of-file is encountered and the *eof-error-p* argument is not **nil**, the kind of error that is signaled may depend on the value of *recursive-p*. If *recursive-p* is *true*, then the end-of-file is deemed to have occurred within the middle of a printed representation; if *recursive-p* is *false*, then the end-of-file may be deemed to have occurred

between *objects* rather than within the middle of one.