6. Symbols

This chapter discusses the symbol as a Lisp data type. The Lisp system uses symbols as variables and function names, but these applications of symbols are discussed in chapter 3.

6.1 The Value Cell

Each symbol has associated with it a *value cell*, which refers to one Lisp object. This object is called the symbol's *value*, since it is what you get when you evaluate the symbol as a dynamic variable in a program. Variables and how they work are described in section 3.1, page 25. We also say the the symbol *is bound to* the object which is its value. The symbols nil and t are always bound to themselves; they may not be assigned, bound, or otherwise used as variables. The same is true of symbols in the keyword package.

The value cell can also be *void*, referring to *no* Lisp object, in which case the symbol is said to be void or *unbound*. This is the initial state of a symbol when it is created. An attempt to evaluate a void symbol causes an error.

Lexical variable bindings are not stored in symbol value cells. The functions in this section have no interaction with lexical bindings.

symeval *symbol*
symbol-value *symbol*

symeval is the basic primitive for retrieving a symbol's value. (symeval *symbol*) returns *symbol*'s current binding. This is the function called by eval when it is given a symbol to evaluate. If the symbol is unbound, then symeval signals an error. symbol-value is the Common Lisp name for this function.

set *symbol value*

set is the primitive for assignment of symbols. The *symbol*'s value is changed to *value*; *value* may be any Lisp object. set returns *value*.
Example:
```
(set (cond ((eq a b) 'c)
           (t 'd))
     'foo)
```
either sets c to foo or sets d to foo.

(setf (symeval *symbol*) *value*) is a more modern way to do this.

boundp *symbol*

t if *symbol*'s value cell is not void.

makunbound *symbol*

Makes *symbol*'s value cell void.

Example:
```
        (setq a 1)
        a => 1
        (makunbound 'a)
        a => causes an error.
```
makunbound returns its argument.

value-cell-location *symbol*

Returns a locative pointer to *symbol*'s value cell. See the section on locatives (chapter 14, page 267). It is preferable to write
```
        (locf (symeval symbol))
```
which is equivalent, instead of calling this function explicitly.

This is actually the internal value cell; there can also be an external value cell. For details, see the section on closures (chapter 12, page 250).

For historical compatibility, value-cell-location of a quoted symbol is recognized specially by the compiler and treated like variable-location (page 30). However, such usage results in a compiler warning, and eventually this compatibility feature will be removed.

6.2 The Function Cell

Every symbol also has associated with it a *function cell*. The *function* cell is similar to the *value* cell; it refers to a Lisp object. When a function is referred to by name, that is, when a symbol is passed to apply or appears as the car of a form to be evaluated, that symbol's function cell is used to find its *definition*, the functional object which is to be applied. For example, when evaluating (+ 5 6), the evaluator looks in +'s function cell to find the definition of +, in this case a compiled function object, to apply to 5 and 6.

Maclisp does not have function cells; instead, it looks for special properties on the property list. This is one of the major incompatibilities between the two dialects.

Like the value cell, a function cell can be void, and it can be bound or assigned. (However, to bind a function cell you must use the %bind subprimitive; see page 284.) The following functions are analogous to the value-cell-related functions in the previous section.

fsymeval *symbol*
symbol-function *symbol*

Returns *symbol*'s definition, the contents of its function cell. If the function cell is void, fsymeval signals an error. symbol-function is the Common Lisp name for this function.

fset *symbol definition*

Stores *definition*, which may be any Lisp object, into *symbol*'s function cell. It returns *definition*.

(setf (fsymeval *symbol*) *definition*) is a more modern way to do this.

fboundp *symbol*

nil if *symbol*'s function cell is void, i.e. if *symbol* is undefined. Otherwise it returns t.

fmakunbound *symbol*

Causes *symbol* to be undefined, i.e. its function cell to be void. It returns *symbol*.

function-cell-location *symbol*

Returns a locative pointer to *symbol*'s function cell. See the section on locatives (chapter 14, page 267). It is preferable to write

```
        (locf (fsymeval symbol))
```

rather than calling this function explicitly.

Since functions are the basic building block of Lisp programs, the system provides a variety of facilities for dealing with functions. Refer to chapter 11 for details.

6.3 The Property List

Every symbol has an associated property list. See section 5.10, page 113 for documentation of property lists. When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For instance, the editor uses the property list of a symbol which is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

Because of the existence of print-name, value, function, and package cells, none of the Maclisp system property names (expr, fexpr, macro, array, subr, lsubr, fsubr, and in former times value and pname) exist in Zetalisp.

plist *symbol*
symbol-plist

Returns the list which represents the property list of *symbol*. Note that this is not actually a property list; you cannot do get on it. This value is like what would be the cdr of a property list.

symbol-plist is the Common Lisp name.

setplist *symbol list*

Sets the list which represents the property list of *symbol* to *list*. setplist is to be used with caution (or not at all), since property lists sometimes contain internal system properties, which are used by many useful system functions. Also it is inadvisable to have the property lists of two different symbols be eq, since the shared list structure will cause unexpected effects on one symbol if putprop or remprop is done to the other.

setplist is equivalent to

```
(setf (plist symbol) list)
```

property-cell-location *symbol*

Returns a locative pointer to the location of *symbol*'s property-list cell. This locative pointer may be passed to get or putprop with the same results as if as *symbol* itself had been passed. It is preferable to write

```
(locf (plist symbol))
```

rather than using this function.

6.4 The Print Name

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to read, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, print types out the print-name.

If a symbol is uninterned, #: is normally printed as a prefix before the symbol's print-name. If the symbol is interned, a package prefix may be printed, depending on the current package and how it relates to the symbol's home package.

For more information, see the sections on the *reader* (see section 23.3, page 516), *printer* (see section 23.1, page 506), and packages (see chapter 27, page 636).

symbol-name *symbol*
get-pname *symbol*

Returns the print-name of the symbol *symbol*.
Example:

```
(symbol-name 'xyz) => "XYZ"
```

get-pname is an older name for this function.

6.5 The Package Cell

Every symbol has a *package cell* which, for interned symbols, is used to point to the package which the symbol belongs to. For an uninterned symbol, the package cell contains nil. For information about packages in general, see the chapter on packages, chapter 27, page 636. For information about package cells, see page 639.

6.6 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the intern function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation (p.r.) of a symbol. When the reader sees such a p.r., it calls intern (see page 645), which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, read uses the already-existing symbol. If it does not, then intern creates a new symbol and puts it into the table; read uses that new symbol.

A symbol that has been put into such a table is called an *interned* symbol. Interned symbols are normally created automatically; the first time that someone (such as the reader) asks for a symbol with a given print-name, that symbol is automatically created.

These tables are called *packages*. For more information, turn to the chapter on packages (chapter 27, page 636).

An *uninterned* symbol is a symbol that has not been recorded or looked up in a package. It is used simply as a data object, with no special cataloging. An uninterned symbol prints with a prefix #: when escaping is in use, unless *print-gensym* is nil. This allows uninterned symbols to be distinguishable and to read back in as uninterned symbols. See page 515.

The following functions can be used to create uninterned symbols explicitly.

make-symbol *pname* &optional *permanent-p*

Creates a new uninterned symbol, whose print-name is the string *pname*. The value and function cells are void and the property list is empty. If *permanent-p* is specified, it is assumed that the symbol is going to be interned and probably kept around forever; in this case it and its pname are put in the proper areas. If *permanent-p* is nil (the default), the symbol goes in the default area and the pname is not copied. *permanent-p* is mostly for the use of intern itself.
Examples:

```
(setq a (make-symbol "foo")) => foo
(symeval a) => ERROR!
```

Note that the symbol is *not* interned; it is simply created and returned.

copysymbol *symbol copy-props*
copy-symbol *symbol copy-props*

Returns a new uninterned symbol with the same print-name as *symbol*. If *copy-props* is non-nil, then the value and function-definition of the new symbol are the same as those of *symbol*, and the property list of the new symbol is a copy of *symbol*'s. If *copy-props* is nil, then the new symbol's function and value are void, and its property list is empty.

gensym &optional *x*

Invents a print-name and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name is a prefix (the value of si:*gensym-prefix) followed by the decimal representation of a number (the value of si:*gensym-counter), e.g. g0001. The number is increased by one every time gensym is called.

If the argument *x* is present and is a fixnum, then si:*gensym-counter is set to *x*. If *x* is a string or a symbol, then si:*gensym-prefix is set to it, so it becomes the prefix for this and successive calls to gensym. After handling the argument, gensym creates a symbol as it would with no argument.
Examples:

```
if      (gensym) => #:g0007
then    (gensym 'foo) => #:foo0008
        (gensym 32.) => #:foo0032
        (gensym) => #:foo0033
```

Note that the number is in decimal and always has four digits. #: is the prefix normally printed before uninterned symbols.

gensym is usually used to create a symbol which should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye between two such symbols. The optional argument is rarely supplied. The name comes from 'generate symbol', and the symbols produced by it are often called "gensyms".

gentemp &optional (*prefix* "t") (*a-package* package)
Creates and returns a new symbol whose name starts with *prefix*, interned in *a-package*, and distinct from any symbol already present there. This is done by trying names one by one until a name not already in use is found, which may be very slow.