6. Symbols

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 binding or value, since it is what you get when you evaluate the symbol. The binding of symbols to values allows symbols to be used as the implementation of variables in programs.

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

Symbols are often used as special variables. Variables and how they work are described in section 3.1, page 15. The symbols nil and t are always bound to themselves; they may not be assigned, bound, or otherwise used as variables. Attempting to change the value of nil or t usually causes an error.

The functions described here work on symbols, not variables in general. This means that the functions below won't work if you try to use them on local variables.

(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)

will either set c to foo or set d to foo.

(symeval 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 causes an error.

(boundp symbol)

boundp returns t if symbol is bound; otherwise, it returns nil.

(makunbound symbol)

makunbound causes symbol to become unbound.

Example:

(setq a 1)
(a => 1
(makunbound 'a)
(a => causes an error.
makunbound returns its argument.)
value-cell-location symbol

value-cell-location returns a locative pointer to symbol's value cell. See the section on locatives (chapter 13, page 197). It is preferable to write

(1ocf (syemeal symbol))

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 11, page 180).

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

variable-location symbol Special Form

Returns a locative to the cell in which the value of symbol is stored. symbol is an unevaluated argument, so the name of the symbol must appear explicitly in the code.

With ordinary special variables, this is equivalent to

(value-cell-location 'symbol)

However, the compiler does not always store the values of variables in the value cells of symbols. The compiler handles variable-location by producing code that returns a locative to the cell where the value is actually being kept. For a local variable, this will be a pointer into the function's stack frame. For a flavor instance variable, this will be a pointer into the instance which is self's value.

In addition, if symbol is a special variable which is closed over, the value returned will be an external value cell, the same as the value of locate-in-closure applied to the proper closure and symbol. This cell always contains the value which is current only while inside the closure. See page 181.

variable-boundp symbol Special Form

This is non-nil if symbol has a value. All symbols are initially unbound (their value cells are "empty") until they are set or bound to a value. While this is the case, variable-boundp returns nil.

It is equivalent to

(location-boundp (variable-location symbol))

symbol is not evaluated.

variable-makunbound symbol Special Form

This makes symbol's value cell "empty" again, making symbol unbound. Evaluating symbol henceforth will be an error unless symbol is later set or bound.

This is equivalent to

(location-makunbound (variable-location symbol))

symbol is not evaluated.
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 empty, and it can be bound or assigned. (However, to bind a function cell you must use the bind subprimitive; see page 212.) The following functions are analogous to the value-cell-related functions in the previous section.

fsymeval symbol
fsymeval returns symbol’s definition, the contents of its function cell. If the function cell is empty, fsymeval causes an error.

fset symbol definition
fset stores definition, which may be any Lisp object, into symbol’s function cell. It returns definition.

fboundp symbol
fboundp returns nil if symbol’s function cell is empty, i.e. if symbol is undefined. Otherwise it returns t.

fmakunbound symbol
fmakunbound causes symbol to be undefined, i.e. its function cell to be empty. It returns symbol.

function-cell-location symbol
function-cell-location returns a locative pointer to symbol’s function cell. See the section on locatives (chapter 13, page 197). It is preferable to write

(1ocf (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 10 for details.
6.3 The Property List

Every symbol has an associated property list. See section 5.9, page 81 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.

\texttt{plist symbol}

This returns the list which represents the property list of \texttt{symbol}. Note that this is not the property list itself; you cannot do \texttt{get} on it.

\texttt{setplist symbol list}

This sets the list which represents the property list of \texttt{symbol} to \texttt{list}. \texttt{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 \texttt{eq}, since the shared list structure will cause unexpected effects on one symbol if \texttt{putprop} or \texttt{remprop} is done to the other.

\texttt{property-cell-location symbol}

This returns a locative pointer to the location of \texttt{symbol}’s property-list cell. This locative pointer may be passed to \texttt{get} or \texttt{putprop} with the same results as if as \texttt{symbol} itself had been passed. It is preferable to write

\begin{verbatim}
(locf (plist symbol))
\end{verbatim}

rather than using this function.

6.4 The Print Name

Every symbol has an associated string called the \texttt{print-name}, or \texttt{pname} for short. This string is used as the external representation of the symbol: if the string is typed in to \texttt{read}, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, \texttt{print} types out the print-name. For more information, see the sections on the \texttt{reader} (see section 21.2.2, page 371) and \texttt{printer} (see section 21.2.1, page 367).

\texttt{get-pname symbol}

This returns the print-name of the symbol \texttt{symbol}.

Example:

\begin{verbatim}
(get-pname 'xyz) => "XYZ"
\end{verbatim}
samepnamenp sym1 sym2

This predicate returns t if the two symbols sym1 and sym2 have equal print-names; that is, if their printed representations are the same. Upper and lower case letters are normally considered the same. Strings are also accepted as arguments; their contents are used in the comparison. samepnamenp is useful for determining if two symbols would be the same except that they are in different packages (see chapter 24, page 506).

Examples:

(samepnamenp 'xyz (maknam '(x y z)) => t

(samepnamenp 'xyz (maknam '(w x y)) => nil

(samepnamenp 'xyz "XYZ") => t

This is the same function as string-equal (see page 145). samepnamenp is provided mainly so that you can write programs that will work in Maclisp as well as Zetalisp; in new programs, you should just use string-equal.

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 24, page 506. For information about package cells, see page 513.

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 512), 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. In Zetalisp, interned symbols are the province of the package system. Although interned symbols are the most commonly used, they will not be discussed further here. For more information, turn to the chapter on packages (chapter 24, page 506).

An uninterned symbol is a symbol used simply as a data object, with no special cataloging. An uninterned symbol prints the same as an interned symbol with the same print-name, but cannot be read back in.
The following functions can be used to create uninterned symbols explicitly.

**make-symbol pname &optional permanent-p**
This creates a new uninterned symbol, whose print-name is the string `pname`. The value
and function bindings will be unbound and the property list will be 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` will be 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:

```lisp
(setq a (make-symbol "foo")) => foo
(syseval a) => ERROR!
```
Note that the symbol is *not* interned; it is simply created and returned.

**copy-symbol symbol copy-props**
This 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 will be the same as
those of `symbol`, and the property list of the new symbol will be a copy of `symbol`'s. If
`copy-props` is `nil`, then the new symbol will be unbound and undefined, and its property
list will be empty.

**gensym &optional x**
`gensym` 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 character 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 the first character of the string
or of the symbol's print-name. After handling the argument, `gensym` creates a symbol as
it would with no argument.
Examples:

```lisp
    if (gensym) => g0007
    then (gensym 'foo) => f0008
    (gensym 32.) => f0032
    (gensym) => f0033
```

Note that the number is in decimal and always has four digits, and the prefix is always
one character.

`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".