2. Primitive Object Types

2.1 Data Types

This section enumerates some of the various different primitive types of objects in Zetalisp. The types explained below include symbols, conses, various types of numbers, two kinds of compiled code objects, locatives, arrays, stack groups, and closures. With each is given the associated symbolic name, which is returned by the function **data-type** (page 201).

A *symbol* (these are sometimes called "atoms" or "atomic symbols" by other texts) has a *print name*, a *binding*, a *definition*, a *property list*, and a *package*.

The print name is a string, which may be obtained by the function **get-pname** (page 99). This string serves as the *printed representation* (see section 21.2.1, page 367) of the symbol. Each symbol has a *binding* (sometimes also called the "value"), which may be any Lisp object. It is also referred to sometimes as the "contents of the value cell", since internally every symbol has a cell called the *value cell* which holds the binding. It is accessed by the **symeval** function (page 96), and updated by the **set** function (page 96). (That is, given a symbol, you use **symeval** to find out what its binding is, and use **set** to change its binding.) Each symbol has a *definition*, which may also be any Lisp object. It is also referred to as the "contents of the function cell", since internally every symbol has a cell called the *function cell* which holds the definition. The definition can be accessed by the **fsymeval** function (page 98), and updated with **fset** (page 98), although usually the functions **fdefinition** and **fdefine** are employed (page 169). The property list is a list of an even number of elements; it can be accessed directly by **plist** (page 99), and updated directly by **setplist** (page 99), although usually the functions **get**, **putprop**, and **remprop** (page 82) are used. The property list is used to associate any number of additional attributes with a symbol—attributes not used frequently enough to deserve their own cells as the value and definition do. Symbols also have a package cell, which indicates which "package" of names the symbol belongs to. This is explained further in the section on packages (chapter 24) and can be disregarded by the casual user.

The primitive function for creating symbols is **make-symbol** (page 101), although most symbols are created by **read**, **intern**, or **fasload** (which call **make-symbol** themselves.)

A *cons* is an object that cares about two other objects, arbitrarily named the *car* and the *cdr*. These objects can be accessed with **car** and **cdr** (page 61), and updated with **rplaca** and **rplacd** (page 70). The primitive function for creating conses is **cons** (page 61).

There are several kinds of numbers in Zetalisp. *Fixnums* represent integers in the range of -2↑23 to 2↑23-1. *Bignums* represent integers of arbitrary size, but they are more expensive to use than fixnums because they occupy storage and are slower. The system automatically converts between fixnums and bignums as required. *Flonums* are floating-point numbers. *Small-flonums* are another kind of floating-point numbers, with less range and precision, but less computational overhead. *Rationalnums* are exact rational numbers which are represented with a numerator and a denominator which are integers. *Complexnums* are numbers which have explicitly represented real and imaginary parts, which can be any real numbers. See chapter 7, page 102 for full details of these types and the conversions between them.

The usual form of compiled, executable code is a Lisp object called a "Function Entry Frame" or "FEF". A FEF contains the code for one function. This is analogous to what Maclisp calls a "subr pointer". FEFs are produced by the Lisp Compiler (chapter 16, page 228), and are usually found as the definitions of symbols. The printed representation of a FEF includes its name so that it can be identified.

Another Lisp object which represents executable code is a "microcode entry". These are the microcoded primitive functions of the Lisp system, and any user functions compiled into microcode.

About the only useful thing to do with any of these compiled code objects is to *apply* it to arguments. However, some functions are provided for examining such objects, for user convenience. See **arglist** (page 172), **args-info** (page 174), **describe** (page 641), and **disassemble** (page 641).

A *locative* (see chapter 13, page 197) is a kind of a pointer to a single memory cell anywhere in the system. The contents of this cell can be accessed by **cdr** (see page 61) and updated by **rplacd** (see page 70).

An *array* (see chapter 8, page 121) is a set of cells indexed by a tuple of integer subscripts. The contents of the cells may be accessed and changed individually. There are several types of arrays. Some have cells which may contain any object, while others (numeric arrays) may only contain small positive numbers. Strings are a type of array; the elements are 8-bit unsigned numbers which encode characters.

A *list* is not a primitive data type, but rather a data structure made up out of conses and the symbol **nil**. See chapter 5, page 60.

2.2 Predicates

A *predicate* is a function which tests for some condition involving its arguments and returns the symbol **t** if the condition is true, or the symbol **nil** if it is not true. Most of the following predicates are for testing what data type an object has; some other general-purpose predicates are also explained.

By convention, the names of predicates usually end in the letter "p" (which stands for "predicate").

The following predicates are for testing data types. These predicates return **t** if the argument is of the type indicated by the name of the function, **nil** if it is of some other type.

symbolp *arg*

  symbolp returns **t** if its argument is a symbol, otherwise **nil**.

nsymbolp *arg*
nsymbolp returns nil if its argument is a symbol, otherwise t.

listp *arg*
listp returns t if its argument is a cons, otherwise nil. Note that this means (listp nil) is nil even though nil is the empty list. [This may be changed in the future.]

nlistp *arg*
nlistp returns t if its argument is anything besides a cons, otherwise nil. nlistp is identical to atom, and so (nlistp nil) returns t. [This may be changed in the future, if and when listp is changed.]

atom *arg*
The predicate atom returns t if its argument is not a cons, otherwise nil.

consp *arg*
This returns t if *arg* is a cons, otherwise nil. At the moment, this is the same as listp; but while listp may be changed, consp will *never* be true of nil.

numberp *arg*
numberp returns t if its argument is any kind of number, otherwise nil.

fixp *arg*
integerp *arg*
This returns t if its argument is a representation of an integer, i.e. a fixnum or a bignum, otherwise nil.

floatp *arg*
floatp returns t if its argument is a floating-point number, i.e. a flonum or a small flonum, otherwise nil.

fixnump *arg*
fixnump returns t if its argument is a fixnum, otherwise nil.

bigp *arg*
bigp returns t if *arg* is a bignum, otherwise nil.

flonump *arg*
flonump returns t if *arg* is a (large) flonum, otherwise nil.

small-floatp *arg*
small-floatp returns t if *arg* is a small flonum, otherwise nil.

rationalp *arg*
Returns t if *arg* is an exact representation of a rational number; that is, if it is a fixnum, a bignum or a rationalnum. Otherwise it returns nil.

complexp *arg*

Returns t if *arg* is a complexnum, a number which is explicitly represented as complex. Otherwise it returns **nil**.

Since the system will not in normal operation produce complexnums have a zero imaginary part, you can use this as a test for a number which is mathematically not a real number.

realp *arg*

Returns t if *arg* is a number whose imaginary part is zero. Any fixnum, bignum, flonum (of any size) or rationalnum satisfies this predicate. Otherwise it returns **nil**.

Since a complexnum never normally has a zero imaginary part, you can use this as a test for numbers that are mathematically real.

stringp *arg*

stringp returns **t** if its argument is a string, otherwise **nil**.

arrayp *arg*

arrayp returns **t** if its argument is an array, otherwise **nil**. Note that strings are arrays.

functionp *arg* &optional *allow-special-forms*

functionp returns t if its argument is a function (essentially, something that is acceptable as the first argument to **apply**), otherwise it returns **nil**. In addition to interpreted, compiled, and microcoded functions, **functionp** is true of closures, select-methods (see page 163), and symbols whose function definition is **functionp**. **functionp** is not true of objects which can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances. If *allow-special-forms* is specified and non-**nil**, then **functionp** will be true of macros and special-form functions (those with quoted arguments). Normally **functionp** returns **nil** for these since they do not behave like functions. As a special case, **functionp** of a symbol whose function definition is an array returns t, because in this case the array is being used as a function rather than as an object.

subrp *arg*

subrp returns **t** if its argument is any compiled code object, otherwise **nil**. The Lisp Machine system doesn't use the term "subr", but the name of this function comes from Maclisp.

closurep *arg*

closurep returns **t** if its argument is a closure, otherwise **nil**.

entityp *arg*

entityp returns **t** if its argument is an entity, otherwise nil. See section 11.4, page 185 for information about entities.

locativep *arg*

locativep returns t if its argument is a locative, otherwise nil.

typep *arg* &optional *type*

typep is really two different functions. With one argument, typep is not really a predicate; it returns a symbol describing the type of its argument. With two arguments, typep is a predicate which returns t if *arg* is of type *type*, and nil otherwise. Note that an object can be "of" more than one type, since one type can be a subset of another.

The symbols that can be returned by typep of one argument are:

:symbol — *arg* is a symbol.

:fixnum — *arg* is a fixnum (not a bignum).

:bignum — *arg* is a bignum.

:flonum — *arg* is a flonum (not a small-flonum).

:small-flonum — *arg* is a small flonum.

:rational — *arg* is a rationalnum.

:complex — *arg* is a complexnum.

:list — *arg* is a cons.

:locative — *arg* is a locative pointer (see chapter 13, page 197).

:compiled-function — *arg* is the machine code for a compiled function (sometimes called a FEF).

:microcode-function — *arg* is a function written in microcode.

:closure — *arg* is a closure (see chapter 11, page 180).

:select-method — *arg* is a select-method table (see page 163).

:stack-group — *arg* is a stack-group (see chapter 12, page 186).

:string — *arg* is a string.

:array — *arg* is an array that is not a string.

:random — Returned for any built-in data type that does not fit into one of the above categories.

foo — An object of user-defined data type *foo* (any symbol). The primitive type of the object could be array, instance, or entity. See Named Structures, page 312, Flavors, chapter 20, page 321, and Entities section 11.4, page 185.

The *type* argument to typep of two arguments can be any of the above keyword symbols (except for :random), the name of a user-defined data type (either a named structure or a flavor), or one of the following additional symbols:

:atom — Any atom (as determined by the atom predicate).

:integer — Any kind of fixed-point number (fixnum or bignum).

:fix — The same as :integer.

:rational — When :rational is used as the second argument to typep, integers are considered rational, even though typep of a single integer argument will return :fixnum or :bignum rather than :rational.

:float — Any kind of floating-point number (flonum or small-flonum).

:real — *arg* is a number but not a complexnum.

:number — Any kind of number.

:named-structure — *arg* is a named structure. typep of one argument returns the named structure type symbol.

:instance — An instance of any flavor. See chapter 20, page 321.

:entity — An entity. typep of one argument returns the name of the particular user-defined type of the entity, rather than :entity.

See also data-type, page 201.

Note that (typep nil) => :symbol, and (typep nil ':list) => nil; the latter may be changed.

The following functions are some other general purpose predicates.

eq *x y*

(eq *x y*) => t if and only if *x* and *y* are the same object. It should be noted that things that print the same are not necessarily eq to each other. In particular, numbers with the same value need not be eq, and two similar lists are usually not eq.
Examples:

```
(eq 'a 'b) => nil
(eq 'a 'a) => t
(eq (cons 'a 'b) (cons 'a 'b)) => nil
(setq x (cons 'a 'b)) (eq x x) => t
```

Note that in Zetalisp equal fixnums are eq; this is not true in Maclisp. Equality does not imply eq-ness for other types of numbers. To compare numbers, use =; see page 106.

neq *x y*

(neq *x y*) = (not (eq *x y*)). This is provided simply as an abbreviation for typing convenience.

eql *x y*

eql is the same as eq unless both arguments are numbers; in that case, it is the same as =.

equal *x y*

The equal predicate returns t if its arguments are similar (isomorphic) objects. (cf. eq) Two numbers are equal if they have the same value and type (for example, a flonum is never equal to a fixnum, even if = is true of them). For conses, equal is defined recursively as the two cars being equal and the two cdrs being equal. Two strings are equal if they have the same length, and the characters composing them are the same; see string-equal, page 145. Alphabetic case is ignored (but see alphabetic-case-affects-string-comparison, page 144). All other objects are equal if and only if they are eq. Thus equal could have been defined by:

```
(defun equal (x y)
  (cond ((eq x y) t)
        ((neq (typep x) (typep y)) nil)
        ((numberp x) (= x y))
        ((stringp x) (string-equal x y))
        ((listp x) (and (equal (car x) (car y))
                        (equal (cdr x) (cdr y))))))
```

As a consequence of the above definition, it can be seen that equal may compute forever when applied to looped list structure. In addition, eq always implies equal; that is, if (eq a b) then (equal a b). An intuitive definition of equal (which is not quite correct) is that two objects are equal if they look the same when printed out. For example:

```
(setq a '(1 2 3))
(setq b '(1 2 3))
(eq a b) => nil
(equal a b) => t
(equal "Foo" "foo") => t
```

not *x*
null *x*

not returns t if *x* is nil, else nil. null is the same as not; both functions are included for the sake of clarity. Use null to check whether something is nil; use not to invert the sense of a logical value. Even though Lisp uses the symbol nil to represent falseness, you shouldn't make understanding of your program depend on this fortuitously. For example, one often writes:

```
(cond ((not (null lst)) ... )
      ( ... ))
```

rather than

```
(cond (lst ... )
      ( ... ))
```

There is no loss of efficiency, since these will compile into exactly the same instructions.