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.

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

The print name is a string, which may be obtained by the function symbol-name (page 132). This string serves as the *printed representation* (see section 23.1, page 506) of the symbol.

Each symbol has a *value*, which may be any Lisp object. This is the value of the symbol when regarded as a dynamic variable. 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 value. It is accessed by the symeval function (page 129), and updated by the set function (page 129). (That is, given a symbol, you use symeval to find out what its value is, and use set to change its value.)

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 130), and updated with fset (page 130), although usually the functions fdefinition and fdefine are employed (page 239).

The property list is a list of an even number of elements; it can be accessed directly by plist (page 131), and updated directly by setplist (page 131), although usually the functions get, putprop, and remprop (page 114) 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 27) and can be disregarded by the casual user.

The primitive function for creating symbols is make-symbol (page 133), 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 87), and updated with rplaca and rplacd (page 89). The primitive function for creating conses is cons (page 87).

There are several kinds of numbers in Zetalisp. *Fixnums* represent integers in the range of -2↑24 to 2↑24-1. *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. *Floats* are floating-point numbers. *Short floats* are

another kind of floating-point numbers, with less range and precision, but less computational overhead. *Ratios* are exact rational numbers that are represented with a numerator and a denominator, which are integers. *Complexnums* are numbers that have explicitly represented real and imaginary parts, which can be any real numbers of the same type. See chapter 7, page 135 for full details of these types and the conversions between them.

A *character object* is much like a fixnum except that its type is distinguishable. Common Lisp programs use character objects to represent characters. Traditional programs usually use fixnums to represent characters, although they can create an manipulate character objects when they desire. Character objects behave like fixnums when used in arithmetic; only a few operations make any distinction. They do, however, print distinctively. See section 10.1, page 204 for more information.

The usual form of compiled, executable code is a Lisp object, called a "Function Entry Frame" or "FEF" for historical reasons. 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 17, page 301), 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 kind of Lisp object that 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 242), **args-info** (page 243), **describe** (page 791), and **disassemble** (page 792).

A *locative* (see chapter 14, page 267) 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 87) and updated by **rplacd** (see page 89).

An *array* (see chapter 8, page 162) 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 that may contain any object, while others (numeric arrays) may only contain small positive numbers. Strings are a type of array; the elements are character objects.

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 86.

2.2 Data Type Predicates

A *predicate* is a function that 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. The following predicates are for testing what data type an object has.

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 *object*
t if *object* is a symbol, otherwise nil.

nsymbolp *object*
nil if *object* is a symbol, otherwise t.

listp *object*
t if *object* 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 to work like cli:listp. Since the current definition of listp is identical to that of consp, all uses of listp should be changed to consp unless the treatment of nil is not of concern.]

cli:listp *object*
The Common Lisp version of listp returns t if *object* is nil or a cons.

nlistp *object*
t if *object* is anything besides a cons, otherwise nil. (nlistp nil) returns t.

[This may be changed in the future, if and when listp is changed. Since the current definition of nlistp is identical to that of atom, all uses of nlistp should be changed to atom unless the treatment of nil is not of concern.]

atom *object*
t if *object* is not a cons, otherwise nil. This is the same as (not (consp *object*)).

consp *object*
t if *object* 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 *object*
t if *object* is any kind of number, otherwise nil.

integerp *object*
fixp *object*
Return t if *object* is a representation of an integer, i.e. a fixnum or a bignum, otherwise nil.

floatp *object*
t if *object* is a floating-point number, i.e. a full-size or short float, otherwise nil.

fixnump *object*
t if *object* is a fixnum, otherwise nil.

bigp *object*
t if *object* is a bignum, otherwise nil.

flonump *object*
t if *object* is a full-size float, otherwise nil.

small-floatp *object*
t if *object* is a short float, otherwise nil.

rationalp *object*
t if *object* is an exact representation of a rational number; that is, if it is a fixnum, a bignum or a ratio. Otherwise nil.

complexp *object*
t if *object* is a complexnum, a number explicitly represented as complex. Otherwise nil.

realp *object*
t if *object* is a number whose value is real, otherwise nil. Any fixnum, bignum, float (of either format) or ratio satisfies this predicate. So does a complexnum whose imaginary part is zero.

characterp *object*
t if *object* is a character object, otherwise nil.

stringp *object*
t if *object* is a string, otherwise nil.

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

vectorp *object*
t if *object* is an array of rank 1.

bit-vector-p *object*
t if *object* is an array of rank 1 that allows only 0 and 1 as elements.

simple-vector-p *object*
t if *object* is an array of rank 1, with no fill pointer and not displaced, that can have any Lisp object as an element.

simple-bit-vector-p *object*
t if *object* is an array of rank 1, with no fill pointer and not displaced, that allows only 0 and 1 as elements.

simple-string-p *object*
t if *object* is a string with no fill pointer and not displaced.

functionp *object* &optional *allow-special-forms*
t if *object* is a function (essentially, something that is acceptable as the first argument to apply), otherwise nil. In addition to interpreted, compiled, and microcoded functions, functionp is true of closures, select-methods (see page 232), and symbols whose function definition is functionp.

functionp is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances. 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.

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.

compiled-function-p *object*
subrp *object*
t if *object* is any compiled code object, otherwise nil. The name subrp is for Maclisp compatibility.

special-form-p *symbol*
t if *symbol* is defined as a function that takes some unevaluated args. Macros do not count as special forms.

macro-function can be used to test whether a symbol is defined as a macro, but you must be careful because it also returns a non-nil value for certain special forms. See the definition macro-function (page 344) to find out how to do this properly.

closurep *object*
t if *object* is a closure, otherwise nil.

entityp *object*
t if *object* is an entity, otherwise nil. See section 12.4, page 255 for information about entities.

locativep *object*
t if *object* is a locative, otherwise nil.

commonp *object*
t if *object* is of a type that Common Lisp defines operations on. See the type specifier common (page 18).

Other standard type predicates include packagep (see page 656), random-state-p (see page 157), hash-table-p (page 119), pathnamep (page 545), streamp (page 459) and readtablep (page 536). defstruct can define additional type predicates automatically (page 378).

2.3 Type Specifiers

Data types can be represented symbolically by Lisp objects called *type specifiers*. A type specifier describes a class of possible Lisp objects; the function typep tells whether a given object matches a given type specifier.

Built-in type specifiers exist for the actual Lisp Machine data types. The user can define additional type specifiers to represent arbitrary classifications of data. Type specifiers can also be combined into specifiers for more complex types.

Some type specifiers are symbols: for example, number, cons, symbol, integer, character, compiled-function, array, vector. Their meanings are mostly obvious, but a table follows below. Type specifiers that are symbols are called *simple* type specifiers.

Lists can also be type specifiers. They are usually combinations or restrictions of other type specifiers. The car of the list is the key to understanding what it means. An example of a combination is (or array symbol), which matches any array or any symbol. An example of a restriction type is (integer 0 6), which matches only integers between 0 and 6 (inclusive).

2.3.1 Standard Type Specifiers

Basic Data Types

cons	non-nil lists.
symbol	symbols.
array	all arrays, including strings.
number	numbers of all kinds.
instance	all instances of any flavor.
structure	named structures of any structure type.
locative	locatives.
closure	closures.
entity	entities.

stack-group stack groups.

compiled-function
macrocode functions such as the compiler makes.

microcode-function
built-in functions implemented by the microcode.

select select-method functions (defined by **defselect** or **defselect-incremental**).

character character objects.

Other Useful Simple Types

t all Lisp objects belongs to this type.

nil nothing belongs to this type.

string-char characters that can go in strings.

standard-char
characters defined by Common Lisp. These are the 95 ASCII printing characters (including **Space**), together with **Return**.

null **nil** is the only object that belongs to type **null**.

list lists, including **nil**. This type is the union of the types **null** and **cons**.

sequence lists and vectors. Many Common Lisp functions accept either a list or a vector as a way of describing a sequence of elements.

keyword keywords (symbols belonging to package **keyword**).

atom anything but conses.

Simple Number Types

integer fixnums and bignums.

ratio explicit rational numbers, such as **1\2** (**1/2** in Common Lisp syntax).

rational integers and ratios.

fixnum small integers, whose **%data-type** is **dtp-fix** and which occupy no storage.

bignum larger integers, which occupy storage.

bit very small integers—only **0** and **1** belong to this type.

float any floating point number regardless of format.

short-float short floats

single-float full-size floats

double-float
long-float defined by Common Lisp, but on the Lisp Machine synonymous with **single-float**.

real any number whose value is real.

complex a number explicitly stored as complex. It is possible for such a number to have zero as an imaginary part but only if it is a floating point zero.

noncomplex a number which is not explicitly stored as complex. This is a subtype of real.

Restriction Types for Numbers

(complex *type-spec*)

complex numbers whose components match *type-spec*. Thus, (complex rational) is the type of complex numbers with rational components. (complex t) is equivalent to complex.

(integer *low high*)

integers between *low* and *high*. *low* can be:

integer *integer* is an inclusive lower limit

(*integer*) *integer* is an exclusive lower limit.

* There is no lower limit.

high has the same sorts of possibilities. If *high* is omitted, it defaults to *. If both *low* and *high* are omitted, you have (integer), which is equivalent to plain integer. Examples:

```
(integer 0 *)      matches any nonnegative integer.
(integer 0)        matches any nonnegative integer.
(integer -4 3)     matches any integer between -4 and 3, inclusive.
(integer -4 (4))   matches any integer between -4 and 3, inclusive.
```

bit is equivalent to (integer 0 1).

(rational *low high*)
(float *low high*)
(short-float *low high*)
(single-float *low high*)
(double-float *low high*)
(long-float *low high*)
(noncomplex *low high*)

These specify restrictive bounds for the types rational, float and so on. The bounds work on these types just the way they do on integer. Exclusive and inclusive bounds make a useful difference here:

```
(float (-4) (3))   matches any float between -4 and 3, exclusive.
```

No possible inclusive bounds could provide the same effect.

(mod *high*) nonnegative integers less than *high*. *high* should be an integer. (mod), (mod *) and plain mod are allowed, but are equivalent to (integer 0).

(signed-byte *size*)

integers that fit into a byte of *size* bits, of which one bit is the sign bit. (signed-byte 4) is equivalent to (integer -8 7). (signed-byte *) and plain signed-byte are equivalent to integer.

(unsigned-byte *size*)

nonnegative integers that fit into a byte of *size* bits, with no sign bit. (unsigned-byte 3) is equivalent to (integer 0 7). (unsigned-byte *) and plain unsigned-

byte are equivalent to (integer 0).

Simple Types for Arrays

array all arrays.

simple-array arrays that are not displaced and have no fill pointers. (Displaced arrays are defined in section 8.2.1, page 166 and fill pointers on page 166).

vector arrays of rank one.

bit-vector art-1b arrays of rank one.

string strings; art-string and art-fat-string arrays of rank one.

simple-bit-vector
bit vectors that are simple arrays.

simple-string strings that are simple arrays.

simple-vector simple-arrays of rank one, whose elements' types are unrestricted. This is not the same as (and vector simple-array)!

Restriction Types for Arrays

(array *element-type dimensions*)
arrays whose rank and dimensions fit the restrictions described by *dimensions* and whose nature restricts possible elements to match *element-type*.

The array elements condition has nothing to do with the actual values of the elements. Rather, it is a question of whether the array's own type permits exactly such elements as would match *element-type*. If anything could be stored in the array that would not match *element-type*, then the array does not match. If anything that would match *element-type* could not be stored in the array, then the array does not match.

For example, if *element-type* is (signed-byte 4), the array must be an art-4b array. An art-1b array will not do, even though its elements all do match (signed-byte 4), because some objects such as the number 12 match (signed-byte 4) but could not be stored in an art-1b array. Likewise an art-q array whose elements all happen to match (signed-byte 4) will not do, since new elements such as nil or 231 which fail to match could potentially be stored in the array.

If *element-type* is t, the type to which all objects belong, then the array must be one in which any object can be stored: art-q or art-q-list.

* as *element-type* means "no restriction". Any type of array is then allowed, whether it restricts its elements or not.

dimensions can be *, an integer or a list. If it is *, the rank and dimensions are not restricted. If it is an integer, it specifies the rank of the array. Then any array of that rank matches.

If *dimensions* is a list, its length specifies the rank, and each element of *dimensions* restricts one dimension. If the element is an integer, that dimension's length must equal it. If the element is *, that dimension's length is not restricted.

(simple-array *element-type dimensions*)
: the restrictions work as in (array *element-type dimensions*), but in addition the array must be a simple array.

(vector *element-type size*)
: *element-type* works as above. The array must be a vector. *size* must be an integer or *; if it is an integer, the array's length must equal *size*.

(bit-vector *size*)
(simple-vector *size*)
(simple-bit-vector *size*)
(string *size*)
(simple-string *size*)
: These require the array to match type bit-vector, simple-vector, etc. This implicitly restricts the element type, so there is no point in allowing an *element-type* to be given in the type specifier. *size* works as in vector.

More Obscure Types

package — packages, such as find-package might return.

readtable — structures such as can be the value of readtable.

pathname — pathnames (instances of the flavor pathname).

hash-table — hash-tables (instances of the flavor hash-table).

flavor-name — instances of that flavor, or of any flavor that contains it.

defstruct-name — named structures of that type, or of any structure that includes that one using :include.

Common Lisp Compatibility Types

random-state — random-states. See random (page 157). This is actually a special case of using a defstruct name as a type specifier, but it is mentioned specifically because Common Lisp defines this type.

common — All objects of types defined by Common Lisp. This is all Lisp objects except closures, entities, stack groups, locatives, instances, select-methods, and compiled and microcode functions. (A few kinds of instances, such as pathnames, are common, because Common Lisp does define how to manipulate pathnames, and it is considered irrelevant that the Lisp Machine happens to implement pathnames using instances.)

stream — Anything that looks like it might be a valid I/O stream. It is impossible to tell for certain whether an object is a stream, since any function with proper behavior may be used as a stream. Therefore, use of this type specifier is discouraged. It exists for the sake of Common Lisp.

Combination Type Specifiers

(member *objects*)
: any one of *objects*, as compared with eql. Thus, (member t nil x) is matched only by t, nil or x.

(satisfies *predicate*)
: objects on which the function *predicate* returns a non-nil value. Thus, (satisfies numberp) is equivalent as a type specifier to number (though the system could not tell that this is so). *predicate* must be a symbol, not a lambda-expression.

(and *type-specs...*)
: objects that match all of the *type-specs* individually. Thus, (and integer (satisfies oddp)) is the type of odd integers.

(or *type-specs...*)
: objects that match at least one of the *type-specs* individually. Thus, (or number array) includes all numbers and all arrays.

(not *type-spec*) objects that do not match *type-spec*.

2.3.2 User-Defined Type Specifiers

deftype *type-name lambda-list body...* *Macro*

Defines *type-name* as a type specifier by providing code to expand it into another type specifier—a sort of type specifier macro.

When a list starting with *type-name* is encountered as a type specifier, the *lambda-list* is matched against the cdr of the type specifier just as the lambda-list of an ordinary defmacro-defined macro is matched against the cdr of a form. Then the *body* is executed and should return a new type specifier to be used instead of the original one.

If there are optional arguments in *lambda-list* for which no default value is specified, they get * as a default value.

If *type-name* by itself is encountered as a type specifier, it is treated as if it were (*type-name*); that is to say, the *lambda-list* is matched against no arguments and then the *body* is executed. So each argument in the *lambda-list* gets its default value, and there is an error if they are not all optional.

Example:

```
(deftype vector (element-type size)
  '(array ,element-type (,size)))
```

could have been used to define vector.

```
(deftype odd-natural-number-below (n)
  '(and (integer 0 (,n)) (satisfies oddp)))

(typep 5 '(odd-natural-number-below 6)) => t
(typep 7 '(odd-natural-number-below 6)) => nil
```

2.3.3 Testing Types with Type Specifiers

type-of *object*

Returns a type specifier which *object* matches. Any given *object* matches many different type specifiers, including t, so you should not attempt to rely on knowing which type specifier would be returned for any particular object. The one actually returned is chosen so as to be informative for a human. Programs should generally use typep rather than type-of.

See also data-type, page 270.

typep *object type-spec*

t if *object* matches *type-spec*. The fundamental purpose of type specifiers is to be used in typep or other functions and constructs that use typep. Examples:

```
(typep 5 'number) => t
(typep 5 '(integer 0 7)) => t
(typep 5 'bit) => nil
(typep 5 'array) => nil
(typep "foo" 'array) => t
(typep nil 'list) => t
(typep '(a b) 'list) => t
(typep 'lose 'list) => nil
(typep 'x '(or symbol number)) => t
(typep 5 '(or symbol number)) => t
```

If the value of *type-spec* is known at compile time, the compiler optimizes typep so that it does not decode the argument at run time.

In Maclisp, typep is used with one argument. It returns a symbol describing the type of the object it is given. This is somewhat like what type-of does, except in Maclisp the intention was to compare the result with eq to test the type of an object. The Lisp Machine supports this usage of typep for compatibility, but the returned symbol is a keyword (such as :list, for conses) which makes it actually incompatible. This usage is considered obsolete and should be removed from programs.

typecase *key-form clauses...* *Macro*

Computes the value of *key-form* and then executes one (or none) of the *clauses* according to the type of the value (call it *key*).

Each clause starts with a type specifier, not evaluated, which could be the second argument to typep. In fact, that is how it is used. The rest of the clause is composed of forms. The type specifiers of the clauses are matched sequentially against *key*. If there is a match, the rest of that clause is executed and the values of the last form in it are returned from the typecase form. If no clause matches, the typecase form returns nil.

typecase, like typep is optimized carefully by the compiler.

Note that t, the type specifier that matches all objects, is useful in the last clause of a typecase. otherwise is also permitted instead of t by special dispensation, with the same meaning.

Example:

```
(typecase foo
  (symbol (get-pname foo))
  (string foo)
  (list (apply 'string-append (mapcar 'hack foo)))
  ((integer 0) (hack-positive-integer foo))
  (t (princ-to-string foo)))
```

etypecase *key-form clauses...* *Macro*

Like typecase except that an uncorrectable error is signaled if every clause fails. t or otherwise clauses are not allowed.

ctypecase *place clauses...* *Macro*

Like etypecase except that the error is correctable. The first argument is called *place* because it must be setf'able (see page 36). If the user proceeds from the error, a new value is read and stored into *place*; then the clauses are tested again using the new value. Errors repeat until a value is specified that makes some clause succeed.

2.3.4 Coercion with Type Specifiers

coerce *object type-spec*

Converts *object* to an "equivalent" object that matches *type-spec*. Common Lisp specifies exactly which types can be converted to which other types. In general, a conversion that would lose information, such as turning a float into an integer, is not allowed as a coercion. Here is a complete list of types you can coerce to.

complex

(complex *type*) Real numbers can be coerced to complex. If a rational is coerced to type complex, the result equals the rational, and is not complex at all. This is because complex numbers with rational components are canonicalized to real if possible. However, if a rational is coerced to (complex float) or (complex single-float) then an actual complex number does result.

It is permissible of course to coerce a complex number to a complex type. The real and imaginary parts are coerced individually to *type* if *type* is specified.

short-float
single-float Rational numbers can be coerced to floating point numbers and any kind of floating point number can be coerced to any other floating point format.

float Rational numbers are converted to single-float's; floats of all kinds are left alone.

character — Strings of length one can be coerced to characters. Symbols whose print-names have length one can also be. An integer can be coerced to a character; this results in a character whose character code is the specified integer.

list — Any vector can be coerced to type list. The resulting list has the same elements as the vector.

vector or array or any restricted array type.
Any sequence (list or vector) can be coerced to any array or vector type. The new array has rank one and the same elements as the original sequence.

If you specify a type of array with restricted element type, you may actually get an array which can hold other kinds of things as well. For example, the Lisp Machine does not provide anything of type (array symbol), but if you specify that, you will get an array which at least can hold symbols (but can hold other things as well). If an element of the original sequence does not fit in the new array, an error is signaled.

t — Any object can be coerced to type t, without change to the object.

If the value of *type-spec* is known at compile time, the compiler optimizes coerce so that it does not decode the argument at run time.

2.3.5 Comparing Type Specifiers

Since a type describes a set of possible objects, it is possible to ask whether one type is contained in another type. Another way to say this is, is one type a *subtype* of another?

subtypep *type1 type2*
t if *type1* is a subtype of *type2*.

The system cannot always tell whether *type1* is a subtype of *type2*. When satisfies type specifiers are in use, this question is mathematically undecidable. Because of this, it has not been considered worthwhile to make the system able to answer obscure subtype questions even when that is theoretically possible. If the answer is not known, subtypep returns nil.

Thus, nil could mean that *type1* is certainly not a subtype of *type2*, or it could mean that there is no way to tell whether it is a subtype. subtypep returns a second value to distinguish these two situations: the second value is t if subtypep's first value is definitive, nil if the system does not know the answer.

Examples:

```
(subtypep 'cons 'list) => t t
(subtypep 'null 'list) => t t
(subtypep 'symbol 'list) => nil t

(subtypep 'list 'number) => nil t
```
because not all lists are numbers (in fact, no lists are numbers).

```
(subtypep 'number 'rational) => nil t
```
because not all numbers are rational.

```
(subtypep '(satisfies foo) '(satisfies bar)) => nil nil
```
because the system does not attempt to figure out your code.