DOME

Alter Programmer’s Reference Manual

Legal Notices

Trademarks

Contents

Introduction 1

Alter is a variant of the Scheme language as defined by the R^4 report[1] and this manual is derived from that same report, although the resemblence is remote. We offer here our sincere gratitude to the “Scheme team” for designing and describing a very useful language that is also easy to implement.

Alter came out of our desire to get out of the business of writing hard-wired back ends for DOME. We used to write them in Smalltalk-80, DOME’s host language, and this kept us busy but not very motivated. Every user wanted to generate something slightly different from his or her DOME models, and they soon flooded us with special requests. We needed to give our user’s a way of doing this sort of thing themselves without requiring them to purchase a Smalltalk development environment. Thus, Alter was born.

Our selection of a Scheme variant was influenced by the CAD Framework Initiative’s (CFI) selection of Scheme as their offical extension language. At that time, CFI was specifying interfaces for an object-oriented framework covering many of the same services we envisioned for DOME. Naturally, then, many of the selection criteria cited by CFI were also appropriate for us: simple syntax, well-defined semantics, extensibility, implementability, and adaptability to a variety of programming paradigms (e.g., functional, object-oriented, declarative). Our familiarity with the implementation of functional languages, esp. Common Lisp, also nudged us toward Scheme.

Alter is a nearly complete implementation of Scheme sitting within DOME. We say “nearly”, because there are a few things that we have omitted from the R^4 definition, most notably proper tail recursion and continuations. Future versions of Alter may remove these limitaions

Alter differs from R^4 Scheme in a few other ways, too.

- Complex numbers are not yet implemented.
- Several primitives have been added to support the manipulation of graph structures built with DOME.
- Alter operates in a windowed, graphics-capable environment, and has some extra primitives to support this.

1 Clinger, W. and Rees, J. (eds), “Revised^4 Report on the Algorithmic Language Scheme”

- Alter code can be used in several settings within DOME, including printing, displaying, animation, constraint-checking, model analysis, and interprocess communication.

This last point is very important. Alter is much more than a medium for writing code-generators or document generators. It is a general-purpose programming language that can be used within DOME in a variety of ways. One key to making better use of Alter is understanding DOME's architecture, especially the object structures used to represent graphs and their annotations. This information can be found in the GrapE Programmer's Reference Manual.

There are several ways to invoke Alter operations and procedures. These are all described in detail in the DOME Extension Manual.

- Directly through an Alter evaluator window
- Each time DOME starts up, by creating a DOME startup script
- As a user-defined plug-in function
- As a printer driver
- In a ProtoDOME model

Overview of Alter 2

Semantics

This chapter summarizes the semantics of Alter and explains the syntactic and lexical conventions of the language. Subsequent chapters describe special forms, numerous data abstractions, and facilities for input and output.

Throughout this manual, we will make frequent references to standard Scheme, which is the language defined by the document Revised^4 Report on the Algorithmic Language Scheme, by William Clinger, Jonathan Rees, et al., or by IEEE Std. 1178-1990, IEEE Standard for the Scheme Programming Language (in fact, several parts of this document are copied from the Revised Report). Alter is an extension of standard Scheme.

These are the significant semantic characteristics of the Alter language:

Variables are statically scoped

> Alter is a **statically scoped** programming language, which means that each use of a variable is associated with a lexically apparent binding of that variable. Algol is another statically scoped language.

Types are latent

> Alter has latent types as opposed to manifest types, which means that Alter associates types with values (or objects) rather than with variables. Other languages with latent types (also referred to as weakly typed or dynamically typed languages) include APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C.

Objects have unlimited extent

> All objects created during a Alter computation, including procedures and continuations, have unlimited extent; no Alter object is ever destroyed. The system doesn't run out of memory because the garbage collector reclaims the storage occupied by an object when the object cannot possibly be needed by a future computation. Other languages in which most objects have unlimited extent include Smalltalk and other Lisp dialects.

Procedures are objects

> Alter procedures are objects, which means that you can create them dynamically, store them in data structures, return them as the results of other procedures, and so on. Other languages with such procedure objects include Common Lisp and ML.

Continuations are explicit

> In most other languages, continuations operate behind the scenes. In Alter, continuations are objects; you can use continuations for implementing a variety of advanced control constructs, including non-local exits, backtracking, and coroutines.

Arguments are passed by value

> Arguments to Alter procedures are passed by value, which means that Alter evaluates the argument expressions before the procedure gains control, whether or not the procedure needs the result of the evaluations. ML, C, and APL are three other languages that pass arguments by value. In languages such as SASL and Algol 60, argument expressions are not evaluated unless the values are needed by the procedure.

Alter uses a parenthesized-list Polish notation to describe programs and (other) data. The syntax of Alter, like that of most Lisp dialects, provides for great expressive power, largely due to its simplicity. An important consequence of this simplicity is the susceptibility of Alter programs and data to uniform treatment by other Alter programs. As with other Lisp dialects, the *read* primitive parses its input; that is, it performs syntactic as well as lexical decomposition of what it reads.

Notational Conventions

This section details the notational conventions used throughout the rest of this document.

Errors

When this manual uses the phrase “an error will be signalled,” it means that Alter will call error, which normally halts execution of the program and prints an error message.

When this manual uses the phrase “it is an error,” it means that the specified action is not valid in Alter, but the system may or may not signal the error. When this manual says that something “must be,” it means that violating the requirement is an error.

This manual gives many examples showing the evaluation of expressions. The examples have a common format that shows the expression being evaluated on the left hand side, an "arrow" in the middle, and the value of the expression written on the right. For example:

```
(+ 1 2)          =>  3
```

Sometimes the arrow and value will be moved under the expression, due to lack of space. Occasionally we will not care what the value is, in which case both the arrow and the value are omitted.

If an example shows an evaluation that results in an error, an error message is shown, prefaced by ‘error-->’:

```
(+ 1 'foo)       error--> Illegal datum
```

An example that shows printed output marks it with `-|':

```
(begin (write 'foo) 'bar)
     -| foo
     => bar
```

When this manual indicates that the value returned by some expression is unspecified, it means that the expression will evaluate to some object without signalling an error, but that programs should not depend on the value in any way.

Alter Concepts

Variable Bindings

Any identifier that is not a syntactic keyword may be used as a variable (see section Identifiers). A variable may name a location where a value can be stored. A variable that does so is said to be bound to the location. The value stored in the location to which a variable is bound is called the variable's value. (The variable is sometimes said to name the value or to be bound to the value.)

A variable may be bound but still not have a value; such a variable is said to be unassigned. Referencing an unassigned variable is an error. When this error is signalled, it is a condition of type condition-type:unassigned-variable; sometimes the compiler does not generate code to signal the error. Unassigned variables are useful only in combination with side effects (see section Assignments).

Environment Concepts

An environment is a set of variable bindings. If an environment has no binding for a variable, that variable is said to be unbound in that environment. Referencing an unbound variable signals a condition of type condition-type:unbound-variable.

A new environment can be created by extending an existing environment with a set of new bindings. Note that “extending an environment” does not modify the environment; rather, it creates a new environment that contains the new bindings and the old ones. The new bindings shadow the old ones; that is, if an environment that contains a binding for x is extended with a new binding for x, then only the new binding is seen when x is looked up in the extended environment. Sometimes we say that the original environment is the parent of the new one, or that the new environment is a child of the old one, or that the new environment inherits the bindings in the old one.

Procedure calls extend an environment, as do *let*, *let**, *letrec*, and *do* expressions. Internal definitions (see section Internal Definitions) also extend an environment. (Actually, all the constructs that extend environments can be expressed in terms of procedure calls, so there is really just one fundamental mechanism for environment extension.) A top-level definition may add a binding to an existing environment.

Initial and Current Environments

Alter provides an initial environment that contains all of the variable bindings described in this manual. Most environments are ultimately extensions of this initial environment. In Alter, the environment in which your programs execute is actually a child (extension) of the environment containing the system's bindings. Thus, system names are visible to your programs, but your names do not interfere with system programs.

The environment in effect at some point in a program is called the current environment at that point. In particular, every REP loop has a current environment. (REP stands for “read-eval-print”; the REP loop is the Alter program that reads your input, evaluates it, and prints the result.) When a new REP loop is created, its environment is determined by the program that creates it.

Static Scoping

Alter is a statically scoped language with block structure. In this respect, it is like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp.

The fact that Alter is statically scoped (rather than dynamically bound) means that the environment that is extended (and becomes current) when a procedure is called is the environment in which the procedure was created (i.e. in which the procedure’s defining lambda expression was evaluated), not the environment in which the procedure is called.

Because all the other Alter binding expressions can be expressed in terms of procedures, this determines how all bindings behave.

Consider the following definitions, made at the top-level REP loop (in the initial environment):

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y))
(f 5)                                     =>  3 ; not 7
```

Here f and g are bound to procedures created in the initial environment. Because Alter is statically scoped, the call to g from f extends the initial environment (the one in which g was created) with a binding of y to 2. In this extended environment, y is 2 and x is 1. (In a dynamically bound Lisp, the call to g would extend the environment in effect during the call to f, in which x is bound to 5 by the call to f, and the answer would be 7.)

Note that with static scoping, you can tell what binding a variable reference refers to just from looking at the text of the program; the referenced binding cannot depend on how the program is used. That is, the nesting of environments (their parent-child relationship) corresponds to the nesting of binding expressions in program text. (Because of this connection to the text of the program, static scoping is also called lexical scoping.) For each place where a variable is bound in a program there is a corresponding region of the program text within which the binding is effective. For example, the region of a binding established by a lambda expression is the entire body of the lambda expression. The documentation of each binding expression explains what the region of the bindings it makes is. A use of a variable (that is, a reference to or assignment of a variable) refers to the innermost binding of that variable whose region contains the variable use. If there is no such region, the use refers to the binding of the variable in the global environment (which is an ancestor of all other environments, and can be thought of as a region in which all your programs are contained).

True and False

In Alter, the boolean values true and false are denoted by #t and #f. However, any Alter value can be treated as a boolean for the purpose of a conditional test. This manual uses the word true to refer to any Alter value that counts as true, and the word false to refer to any Alter value that counts as false. In conditional tests, all values count as true except for #f, which counts as false (see section Conditionals).

External Representations

An important concept in Alter is that of the external representation of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters ‘28’, and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters ‘(8 13)’.

The external representation of an object is not necessarily unique. The integer 28 also has representations ‘#e28.000’ and ‘#x1c’, and the list in the previous paragraph also has the representations ‘(08 13)’ and ‘(8 . (13 . ( )))’.

Many objects have standard external representations, but some, such as procedures and circular data structures, do not have standard representations (although particular implementations may define representations for them). An external representation may be written in a program to obtain the corresponding object (see section Quoting).

External representations can also be used for input and output. The procedure read parses external representations, and the procedure write generates them. Together, they provide an elegant and powerful input/output facility.

Note that the sequence of characters ‘(+ 2 6)’ is not an external representation of the integer 8, even though it is an expression that evaluates to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol + and the integers 2 and 6. Alter’s syntax has the property that any sequence of characters that is an expression is also the external representation of some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data or data as programs.

Disjointness of Types

Every object satisfies at most one of the following predicates (but see section True and False, for an exception):

```
bit-string?    environment?   port?         symbol?
boolean?       null?          procedure?    vector?
cell?          number?        promise?      weak-pair?
char?          pair?          string?
condition?
```

Storage Model

This section describes a model that can be used to understand Alter’s use of storage.

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the string-set! procedure, but the string continues to denote the same locations as before. An object fetched from a location, by a variable reference or by a procedure such as car, vector-ref, or string-ref, is equivalent in the sense of eqv? to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this document speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

Lexical Conventions

This section describes Alter’s lexical conventions.

Whitespace

Whitespace characters are spaces, newlines, tabs, and page breaks. Whitespace is used to improve the readability of your programs and to separate tokens from each other, when necessary. (A token is an indivisible lexical unit such as an identifier or number.) Whitespace is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

Delimiters

All whitespace characters are delimiters. In addition, the following characters act as delimiters:

```
( ) ; " ’ ‘ |
```

Finally, these next characters act as delimiters, despite the fact that Alter does not define any special meaning for them:

```
[ ] { }
```

For example, if the value of the variable name is "max":

```
(list"Hi"name(+ 1 2)) =>  ("Hi" "max" 3)
```

Identifiers

An identifier is a sequence of one or more non-delimiter characters. Identifiers are used in several ways in Alter programs:

Certain identifiers are reserved for use as syntactic keywords; they should not be used as variables (for a list of the initial syntactic keywords, see section Special Form Syntax).

Any identifier that is not a syntactic keyword can be used as a variable.

When an identifier appears as a literal or within a literal, it denotes a symbol.

Alter accepts most of the identifiers that other programming languages allow. Alter allows all of the identifiers that standard Scheme does, plus many more.

Alter defines a potential identifier to be a sequence of non-delimiter characters that does not begin with either of the characters ‘#’ or ‘,’. Any such sequence of characters that is not a syntactically valid number is considered to be a valid identifier. Note that, although it is legal for ‘#’ and ‘,’ to appear in an identifier (other than in the first character position), it is poor programming practice.

Here are some examples of identifiers:

```
lambda              q
list->vector        soup
+                   V17a
<=?                 a34kTMNs
the-word-recursion-has-many-meanings
```

Uppercase and Lowercase

Alter doesn’t distinguish uppercase and lowercase forms of a letter except within character and string constants; in other words, Alter is case-insensitive. For example, ‘Foo’ is the same identifier as ‘FOO’, and ‘#x1AB’ is the same number as ‘#X1ab’. But ‘#\a’ and ‘#\A’ are different characters.

Naming Conventions

A predicate is a procedure that always returns a boolean value (#t or #f). By convention, predicates usually have names that end in ‘?’.

A mutation procedure is a procedure that alters a data structure. By convention, mutation procedures usually have names that end in ‘!’.

Comments

The beginning of a comment is indicated with a semicolon (;). Alter ignores everything on a line in which a semicolon appears, from the semicolon until the end of the line. The entire comment, including the newline character that terminates it, is treated as whitespace.

Additional Notations

The following list describes additional notations used in Alter. See section Numbers, for a description of the notations used for numbers.

+ - .

The plus sign, minus sign, and period are used in numbers, and may also occur in an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs and to indicate a "rest" parameter in a formal parameter list (see section Lambda Expressions).

( ) Parentheses are used for grouping and to notate lists (see section Lists).

" The double quote delimits strings.

\ The backslash is used in the syntax for character constants and as an escape character within string constants.

; The semicolon starts a comment.

’ The single quote indicates literal data; it suppresses evaluation (see section Quoting).

‘ The backquote indicates almost-constant data (see section Quoting).

, The comma is used in conjunction with the backquote (see section Quoting).

,@ A comma followed by an at-sign is used in conjunction with the backquote (see section Quoting).

# The sharp (or pound) sign has different uses, depending on the character that immediately follows it:

#t #f

These character sequences denote the boolean constants (see section Booleans).

#\ This character sequence introduces a character constant (see section Characters).

#(This character sequence introduces a vector constant . A close parenthesis, ‘)’, terminates a vector constant.

#e #i #b #o #d #l #s #x

These character sequences are used in the notation for numbers (see section Numbers).

< This character sequence is used to denote objects that do not have a readable external representation . A close angle, ‘>’, terminates the object's notation. This notation is an Alter extension.

Expressions

An Alter expression is a construct that returns a value. An expression may be a literal, a variable reference, a special form, or a procedure call.

Literal Expressions

Literal constants may be written by using an external representation of the data. In general, the external representation must be quoted; but some external representations can be used without quotation.

```
"abc"          =>  "abc"
145932         =>  145932
#t             =>  #t
#\a            =>  #\a
```

The external representation of numeric constants, string constants, character constants, and boolean constants evaluate to the constants themselves. Symbols, pairs, lists, and vectors require quoting.

Variable References

An expression consisting of an identifier is a variable reference; the identifier is the name of the variable being referenced. The value of the variable reference is the value stored in the location to which the variable is bound. An error is signalled if the referenced variable is unbound or unassigned.

```
(define x 28)
x                =>  28
```

Special Form Syntax

```
(keyword component ...)
```

A parenthesized expression that starts with a syntactic keyword is a special form. Each special form has its own syntax, which is described later in the manual. The following list contains all of the syntactic keywords that are defined when Alter is initialized:

=>	do	letrec
add-method	else	or
and	find-operation	quasiquote
begin	if	quote
case	lambda	set!
cond	let	unquote
define	let*	unquote-splicing
delay		

Procedure Call Syntax

```
    (operator operand ...)
```

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called (the operator) and the arguments to be passed to it (the operands). The operator and operand expressions are evaluated and the resulting procedure is passed the resulting arguments. See section Lambda Expressions, for a more complete description of this.

Another name for the procedure call expression is combination. This word is more specific in that it always refers to the expression; "procedure call" sometimes refers to the process of calling a procedure.

Unlike some other dialects of Lisp, Alter always evaluates the operator expression and the operand expressions with the same evaluation rules, and the order of evaluation is unspecified.

```
    (+ 3 4)                                         =>  7
    ((if #f = *) 3 4)                               =>  12
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables + and *. New procedures are created by evaluating lambda expressions.

If the operator is a syntactic keyword, then the expression is not treated as a procedure call: it is a special form. Thus you should not use syntactic keywords as procedure names. If you were to bind one of these keywords to a procedure, you would have to use apply to call the procedure. Alter signals an error when such a binding is attempted.

Notation Conventions 3

(**procedure-name** *argument-specification*) ⇒ *return-type*

(**operation-name** *argument-specification*) ⇒ *return-type*

An argument specification is a series of the following forms:

argtype	— type of the argument
[arg1 arg2 . . .]	— optional arguments
[n1..n2]	— numeric range (return value only)
argtype . . .	— zero or more arguments of type argtype

Arithmetic 4

(***** *numbers...*) ⇒ *number-or-point*

This procedure return the product of its arguments. This procedure is exactness preserving and accepts either numbers or points.

```
(* 2 17.5)        => 35.0
(* 4)             => 4
(*)               => 1
(* 2 '(3 . 7))    => '(6 . 14)
(* '(2 . 3) '(4 . 5))=> '(8 . 15)
```

(**+** *numbers...*) ⇒ *number-or-point*

This procedure returns the sum of its arguments and accepts both numbers and points. This procedure is exactness preserving.

```
(+ 3 4)           => 7
(+ 3 4.0)         => 7.0
(+ 3)             => 3
(+)               => 0
(+ '(2 . 1) 3)    => '(5 . 4)
(+ '(1 . 2) '(3 . 4))=> '(4 . 6)
```

(**-** *number -...*) ⇒ *number*

With two or more arguments, this procedure returns the difference of its arguments, associating to the left. It accepts both numbers and points. With one argument, however, it returns the additive inverse of its argument. This procedure is exactness preserving.

```
(- 3 4)           => -1
(- 3 4 5)         => -6
(- 3)             => -3
(- '(2 . 3) 1)    => '(1 . 2)
(- '(3 . 5) '(1 . 2))=> '(2 . 3)
```

(**/** *number rest...*) ⇒ *number*

With two or more arguments, this procedure returns the quotient of its arguments, associating to the left. It accepts both numbers and points. With one argument, however, it returns the multiplicative inverse of its argument.

This procedure is exactness preserving, except that division may coerce its result to inexact in implementations that do not support ratnums.

```
(/ 3 4 5)          => 3/20
(/ 3)              => 1/3
(/ 1.5 3)          => 0.5
(/ '(4 . 6) 2)     => '(2 . 3)
(/ '(4 . 6) '(2 . 3))=> '(2 . 2)
```

(**<** *num1 num2 rest...*) ⇒ *#t or #f*

This procedures returns #t if its arguments are monotonically increasing. < is transitive.

The traditional implementations of < in Lisp-like languages are not transitive.

While it is not an error to compare inexact numbers using <, the results may be unreliable because a small inaccuracy may affect the result. Both numbers and points may be supplied as arguments.

(**<=** *num1 num2 rest...*) ⇒ *#t or #f*

This procedures returns #t if its arguments are monotonically nondecreasing. <= is transitive.

The traditional implementations of <= in Lisp-like languages are not transitive.

While it is not an error to compare inexact numbers using <=, the results may be unreliable because a small inaccuracy may affect the result. Both numbers and points may be supplied as arguments.

(**>** *num1 num2 num3...*) ⇒ *#t or #f*

This procedures returns #t if its arguments are monotonically decreasing. > is transitive.

The traditional implementations of > in Lisp-like languages are not transitive.

While it is not an error to compare inexact numbers using >, the results may be unreliable because a small inaccuracy may affect the result. Both numbers and points may be supplied as arguments.

(**>=** *num1 num2 rest...*) ⇒ *#t or #f*

This procedure returns #t if its arguments are monotonically nonincreasing. >= is transitive.

The traditional implementations of >= in Lisp-like languages are not transitive.

While it is not an error to compare inexact numbers using >=, the results may be unreliable because a small inaccuracy may affect the result. Both numbers and points may be supplied as arguments.

(**abs** *number*) ⇒ *number*

Abs returns the magnitude of its argument, which may be either a point or a number. Abs is exactness preserving when its argument is real.

```
(abs -7)          => 7
(abs '(-1 . 4))   => '(1 . 4)
```

(**ceiling** *number*) ⇒ *number*

Ceiling returns the smallest integer not smaller than number. The result is always exact. Ceiling also works with points.

```
(ceiling -4.3)    => -4
(ceiling 3.5)     => 4
(ceiling '(2.5 . -3.7))=> '(3 . -3)
```

(**denominator** *fraction*) ⇒ *integer*

This procedure returns the denominator of its argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(denominator (/ 6 4))=> 2
(denominator (exact->inexact (/ 6 4)))=> 2.0
```

(**floor** *number*) ⇒ *number*

Floor returns the largest integer not larger than number. The result is always exact. Floor also works with points.

```
(floor -4.3)      => -5
(floor 3.5)       => 3
(floor '(2.5 . -3.7))=> '(2 . -4)
```

(**gcd** *number...*) ⇒ *number*

This procedure returns the greatest common divisor of its arguments. The result is always non-negative. This procedure is exactness preserving.

```
(gcd 32 -36)      => 4
(gcd)             => 0
```

(**lcm** *number...*) ⇒ *number*

This procedure returns the greatest least common multiple of its arguments. The result is always non-negative. This procedure is exactness preserving.

```
(lcm 32 -36)      => 288
(lcm 32.0 -36)    => 288.0  ; inexact
(lcm)             => 1
```

(**max** *number -...*) ⇒ *number-or-point*

This procedure returns the maximum of its arguments. It accepts both numbers and points.

```
(max 3 4)        => 4   ; exact
(max 3 4.1)      => 4.1 ; inexact
(max '(1 . 2) '(2 . 1))=> '(2 . 2)
(max '(3 . 4) 5) => '(5 . 5)
```

(**min** *number -...*) ⇒ *number-or-point*

This procedure returns the minimum of its arguments. It accepts both numbers and points.

```
(min 3 4)        => 3   ; exact
(min 3.9 4)      => 3.9 ; inexact
(min '(3 . 4) 2) => '(3 . 4)
(min '(3 . 5) 4) => '(3 . 4)
(min 4 '(3 . 5)) => '(3 . 5)
(min '(1 . 4) '(3 . 2))=> '(1 . 2)
```

If any argument is inexact, then the result will also be inexact. If min is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.

(**modulo** *numerator divisor*) ⇒ *integer*

This exactness-preserving procedure implement number-theoretic (integer) division: For positive integers n1 and n2, if n3 and n4 are integers such that n1=n2*n3+n4 and 0 <= n4 < n2, then

```
(modulo n1 n2)=> n4
```

provided all numbers involved in that computation are exact.

Remainder and modulo differ on negative arguments---the remainder is either zero or has the sign of the dividend, while the modulo always has the sign of the divisor:

```
(modulo 13 4)  => 1
(remainder 13 4)=> 1
(modulo -13 4)=> 3
(modulo 13 -4)=> -3
(modulo -13 -4)=> -1
```

(**numerator** *fraction*) ⇒ *integer*

This procedure returns the numerator of its argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive.

```
(numerator (/ 6 4))=> 3
```

(**quotient** *numerator divisor*) ⇒ *integer*

This exactness-preserving procedure implements number-theoretic (integer) division: For positive integers n1 and n2, if n3 and n4 are integers such that n1=n2*n3+n4 and 0 <= n4 < n2, then

```
(quotient n1 n2)=> n3
```

For integers n1 and n2 with n2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))=> #t
```

provided all numbers involved in that computation are exact.

The value returned by quotient always has the sign of the product of its arguments. Both numbers and points may be supplied as arguments.

(**remainder** *numerator divisor*) ⇒ *integer*

This exactness-preserving procedure implements number-theoretic (integer) division: For positive integers n1 and n2, if n3 and n4 are integers such that n1=n2*n3+n4 and 0 <= n4 < n2, then

```
(remainder n1 n2)=> n4
```

For integers n1 and n2 with n2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))=> #t
```

provided all numbers involved in that computation are exact.

Remainder and modulo differ on negative arguments---the remainder is either zero or has the sign of the dividend, while the modulo always has the sign of the divisor:

```
(remainder 13 4)=> 1
(modulo -13 4)=> 3
(remainder -13 4)=> -1
(remainder 13 -4)=> 1
(remainder -13 -4)=> -1
(remainder -13 -4.0)=> -1.0  ; inexact
```

(**round** *number*) ⇒ *integer*

Round returns the closest integer to number, rounding to even when number is halfway between two integers. Round rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard. The result is always exact. Round also works with points.

```
(round -4.3)     => -4
```

```
(round 3.5)      => 4
(round 7/2)      => 4
(round 7)        => 7
(round '(2.5 . 3.7))=> '(3 . 4)
```

(**sqrt** *number*) ⇒ *number*

Returns the principal square root of number. The result will have either positive real part, or zero real part and non-negative imaginary part.

(**truncate** *number*) ⇒ *integer*

Truncate returns the integer closest to number whose absolute value is not larger than the absolute value of number. The result is always exact. Truncate also works with points.

```
(truncate -4.3)  => -4
(truncate 3.5)   => 3
(truncate '(2.5 . 3.7))=> '(2 . 3)
```

Collections 5

(**append** *list1 list2...*) ⇒ *list*

(**append** *string1 string2...*) ⇒ *string*

Append is an operation defined on both strings and lists. (This is an extension to Scheme R4, which defines append as a procedure defined only on lists.)

Given list arguments, append returns a list consisting of the elements of the first argument list followed by the elements of the remaining argument lists.

Given string arguments, append returns a string consisting of the elements of the first string followed by the elements of the remaining string arguments.

```
(append '(x) '(y))=> (x y)
(append '(a) '(b c d))=> (a b c d)
(append '(a (b)) '((c)))=> (a (b) (c))
(append "a" "bcd")=> "abcd"
```

The resulting list or string is always newly allocated, except that in the case of lists, the result shares structure with the last list argument. With lists, the last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))=> (a b c . d)
(append '() 'a) => a
```

(**copy-without** *list object*) ⇒ *newlist*

Returns a copy of the list with all top-level references to object removed (using eq? test).

```
(copy-without '(a b c) 'b)=> (a c)
(copy-without '(a b c b) 'b)=> (a c)
(copy-without '(b) 'b)=> ()
(copy-without '(a b . c) 'b)=> (a . c)
(copy-without '(b . c) 'b)=> (nil . c)
(copy-without '(nil nil nil . c) 'nil)=> (nil . c)
(copy-without '("a" "b" "c") "b")=> ("a" "b" "c")
(copy-without '(a (b c)) 'b)=> (a (b c))
```

(**flatten** *list*) ⇒ *list*

Returns a list whose members are those elements that are either atoms or members of the result of applying flatten to an element that is a list.

(**length** *list*) ⇒ *integer*

(**length** *string*) ⇒ *integer*

(**length** *vector*) ⇒ *integer*

Length is an operation in Alter, defined on strings, vectors and lists. (This is an extension to Scheme R4, which specifies 'length' as a procedure defined only on lists.) For strings, length behaves exactly like string-length. For vectors, length behaves exactly like vector-length.

```
(length "abcde")=> 5
(length #(1 2 3))=> 3
```

For lists, length returns the number of topmost cells in the argument list, as the following examples illustrate.

```
(length '(a b c))=> 3
(length '(a (b) (c d e)))=> 3
(length '())       => 0
```

(**substitute** *string substring replacement*) ⇒ *string*

Return a copy of the string such that all occurrences of substring are replaced with the replacement string.

```
(substitute "abcdefg" "cde" "newsubstring")=>
"abnewsubstringfg"
```

(**trim** *string*) ⇒ *string*

Returns a copy of the string with all white space removed from its end. White space characters include space, carriage return, tab, line feed, null, and form feed.

```
(trim "abcde   ")=> "abcde"
```

(**width** *string context*) ⇒ *number*

If given a rectangle, returns the width (in the x dimension) of the supplied rectangle (see "rectangle?"). If given a string and a graphics context, returns the width of the string in pixels based on the font currently installed on the graphics context.

```
(width '((3 . 8) . (5 . 12)))=> 2
(width #(#(3 8) #(5 12)))=> 2
(width "A" context)=> depends on current font
```

(**word-wrap** *string [width [ignorecrs]]*) ⇒ *list-of-string*

Word-wrap converts a string into a list of strings, each being at most n characters long, where n is the first optional argument (75 by default). If a second boolean argument is given and is true, then carriage returns are converted to spaces before the string is word-wrapped. If the second optional argument missing or false, a carriage return is represented as a zero-length string in the result. Regardless of the optional arguments, the trailing white space is ignored and not represented in the resulting list.

```
(word-wrap "Wrap these words to fit 10 colums." 10)
        => ("Wrap these" "words to" "fit 10"
"colums.")
```

Colors 6

(**blue** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the blue component of the given color value (see also red, green, make-color).

(**brightness** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the brightness level of the given color value (see also hue, saturation). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color.

(**cyan** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the cyan component of the given color value (see also magenta, yellow). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color

(**green** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the green component of the given color value (see also red, blue). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color

(**hue** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the hue component of the given color value (see also brightness, saturation). Color values are usually obtained via a Graphics-Context instance used when printing graphs through user-supplied print drivers. See make-color.

(**magenta** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the magenta component of the given color value (see also cyan, yellow). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color.

(**make-cmy-color** *cyan magenta yellow*) ⇒ *color*

Creates a color value object whose cyan value is the first argument, magenta value is the second argument and yellow value is the third argument.

(**make-color** *red green blue*) ⇒ *color*

Creates a color value object whose red value is the first argument, green value is the second argument and blue value is the third argument.

(**make-hsb-color** *hue saturation brightness*) ⇒ *color*

Creates a color value object whose hue value is the first argument, saturation value is the second argument and brightness value is the third argument.

(**make-rgb-color** *red green blue*) ⇒ *color*

Creates a color value object whose red value is the first argument, green value is the second argument and blue value is the third argument.

(**red** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the red component of the given color value (see also blue, green). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color.

(**saturation** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the saturation component of the given color value (see also hue, brightness). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color.

(**yellow** *color-type*) ⇒ *number*

Returns a number between 0 and 1 representing the yellow component of the given color value (see also cyan, magenta). Color values are usually obtained via a GraphicsContext instance used when printing graphs through user-supplied print drivers. See make-color.

Control 7

(**apply** *proc list*) ⇒ *object*

Calls proc with the elements of list as the actual arguments.

```
(apply + (list 3 4))=> 7
(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))
((compose sqrt *) 12 75)=> 30
```

(**begin** *body...*) ⇒ *object or nil*

The expression are evaluated sequentially from left to right, and the value of the last expression is returned. This expression type is used to sequence side effects such as input and output.

```
(define x 0)
(begin (set! x 5)
       (+ x 1))      => 6
(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))=> unspecified
. . . and prints  4 plus 1 equals 5
```

(**call-with-current-continuation** *proc*) ⇒ *value*

(**call/cc** *proc*) ⇒ *value*

Proc must be a procedure of one argument. The procedure call-with-current-continuation packages up the current continuation (see the rationale below) as an ‘‘escape procedure” and passes it as an argument to proc. The escape procedure is a Scheme procedure of one argument that, if it is later passed a value, will ignore whatever continuation is in effect at that later time and will give the value instead to the continuation that was in effect when the escape procedure was created.

In standard Scheme, the escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. But this is not the case in Alter, where the escape procedure’s extent is limited to the duration of the activation of the call-with-current-continuation that defined it. The escape procedure may be stored in variables or data structures.

The following examples show only the most common uses of call-with-current-continuation. If all real programs were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

```
(call-with-current-continuation
 (lambda (exit)
  (for-each (lambda (x)
            (if (negative? x)
                (exit x)))
           '(54 0 37 -3 245 19))
  #\t)) => -3


(define list-length
 (lambda (obj)
  (call-with-current-continuation
   (lambda (return)
    (letrec ((r
             (lambda (obj)
              (cond ((null? obj) 0)
                    ((pair? obj)
                     (+ (r (cdr obj)) 1))
                    (else (return #f))))))
     (r obj))))))
(list-length '(1 2 3 4))=> 4
(list-length '(a b . c))=> #f
```

A common use of call-with-current-continuation is for structured, non-local exits from loops or procedure bodies, but in fact call-with-current-continuation is extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don't think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. Call-with-current-continuation allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like ’exit’, ’return’, or even ’goto’. In 1965, however, Peter Landin invented a general purpose escape operator called the J-operator. John Reynolds described a simpler but equally powerful construct in 1972. The "catch" special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds’s construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the "catch" construct could be provided by a procedure instead of by a special syntactic construct, and the name call-with-current-continuation was coined in 1982. Call-with-current-continuation and call/cc refer to the exact same procedure.

(**cond** *clause...*) ⇒ *object or nil*

Each clause should be of the form

```
(test expression . . .)
```

where test is any expression. The last clause may be an ‘‘else clause,” which has the form

```
(else expression1 expression2 . . .)
```

A cond expression is evaluated by evaluating the test expressions of successive clauses in order until one of them evaluates to a true value. When a test evaluates to a true value, then the remaining expressions in its clause are evaluated in order, and the result of the last expression in the clause is returned as the result of the entire cond expression. If the selected clause contains only the test and no expressions, then the value of the test is returned as the result. If all tests evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its expressions are evaluated, and the value of the last one is returned.

```
(cond ((> 3 2) ’greater)
    ((< 3 2) ’less))=> greater
(cond ((> 3 3) ’greater)
    ((< 3 3) ’less)
    (else ’equal))=> equal
```

(**do** *bindings terminator body...*) ⇒ *object or nil*

Detailed syntax is as follows:

```
(do ((variable1 init1 step1)
 . . .)
```

```
    (test expression . . .)
    command . . .)
```

Do is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits with a specified result value.

Do expressions are evaluated as follows:

The init expressions are evaluated (in some unspecified order), the variables are bound to fresh locations, the results of the init expressions are stored in the bindings of the variables, and then the iteration phase begins.

Each iteration begins by evaluating test; if the result is false, then the command expressions are evaluated in order for effect, the step expressions are evaluated in some unspecified order, the variables are bound to fresh locations, the results of the steps are stored in the bindings of the variables, and the next iteration begins.

If test evaluates to a true value, then the expressions are evaluated from left to right and the value of the last expression is returned as the value of the do expression. If no expressions are present, then the value of the do expression is unspecified.

The region of the binding of a variable consists of the entire do expression except for the inits. It is an error for a variable to appear more than once in the list of do variables.

A step may be omitted, in which case the effect is the same as if (variable init variable) had been written instead of (variable init).

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))=> #(0 1 2 3 4)


(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))=> 25
```

(**eval** *object*) ⇒ *object*

Evaluates the expression (a second time) and returns the result of that evaluation.

```
(eval 5)          => 5
(eval 'eq?)       => a procedure
```

```
(eval (list '+ 2 7))=> 9
```

(**for-each** *proc list1 list2...*) ⇒ *nil*

The arguments to for-each are like the arguments to map, but for-each calls proc for its side effects rather than for its values. Unlike map, for-each is guaranteed to call proc on the elements of the lists in order from the first element to the last, and the value returned by for-each is unspecified.

```
(let ((v (make-vector 5)))
 (for-each (lambda (i)
   (vector-set! v i (* i i)))
   '(0 1 2 3 4))
 v)                 => #(0 1 4 9 16)
```

(**for-each-with-separator** *proc separator-proc list1 list2...*) ⇒ *nil*

This procedure performs in a similar manner to the for-each procedure and its arguments are similar to the arguments to for-each with the exception of the new separator-proc argument. The separator-proc is executed after each element in the list is processed except is does not get executed after the last element is processed. The separator-proc takes a single argument which is the most recently processed element.

```
(for-each-with-separator
 (lambda (i) (display i))
 (lambda (i) (display ", "))
 '(1 2 3 4 5))
  => Displays "1, 2, 3, 4, 5" in the transcript window.
```

(**if** *test consequence [alternate]*) ⇒ *object or nil*

Test, consequent, and alternate may be arbitrary expressions.

An if expression is evaluated as follows: first, test is evaluated. If it yields a true value (any value which is not #f), then consequent is evaluated and its value is returned. Otherwise alternate is evaluated and its value is returned. If test yields a false value and no alternate is specified, then the result of the expression is unspecified.

```
(if (> 3 2) 'yes 'no)=> yes
(if (> 2 3) 'yes 'no)=> no
(if (> 3 2)
   (- 3 2)
   (+ 3 2))       => 1
```

(**in-new-environment-do** *body...*) ⇒ *nil*

Creates a new, parent-less environment in which to evaluate the supplied body expressions.

(**map** *proc list1 list2...*) ⇒ *list*

Map applies proc element-wise to the elements of the lists and returns a list of the results, in order from left to right. The dynamic order in which proc is applied to the elements of the lists is unspecified.

```
(map cadr '((a b) (d e) (g h)))=> (b e h)
(map (lambda (n) (expt n n))
  '(1 2 3 4 5))     => (1 4 27 256 3125)
(map + '(1 2 3) '(4 5 6))=> (5 7 9)
(let ((count 0))
  (map
    (lambda (ignored)
      (set! count (+ count 1))
      count)
    '(a b c)))      => unspecified
```

(**show-progress-begin** *label body...*) ⇒ *object or nil*

Show-progress-begin is just like begin, except that a progress meter is displayed during its execution. The meter displays the message supplied as a string as the first argument and a partially-filled circle. The circle shows the percentage of expressions that have been evaluated. Note that this, at best, only approximates the actual proportions involved, since show-progress-begin has no clues about how long it will take to evaluate each expression..

(**show-progress-for-each** *message proc list1 list2...*) ⇒ *nil*

Show-progress-for-each is just like for-each, except that a progress meter is displayed during its execution. The meter displays the message supplied as a string as the first argument and a partially-filled circle. The circle shows the percentage of top-level list elements that have been processed by the given procedure. Note that this, at best, only approximates the actual proportions involved, since show-progress-for-each has no clues about how long it will take to process each set of elements.

(**^super** *type operation object...*) ⇒ *object*

This is just like (operation object . args) except that the method search begins at type rather than the first argument's type. It is required that type be an immediate supertype of the method's receiver type, although the current implementation does not yet enforce this restriction. ^super is analogous to the Smalltalk-80 mechanism of the same name, except that due to Alter's multiple inheritance it is necessary for the programmer to explicitly state which supertype is to be dispatched to.

Converting 8

(**as-backup** *filename-type*) ⇒ *string*

Returns a string that is derived from the given filename and can be used as a backup for that filename. The behavior of this operation is host-specific.

(**char->integer** *char*) ⇒ *integer*

Given a character, char->integer returns an exact integer representation of the character. This procedure implements an injective order isomorphism between the set of characters under the char<=? ordering and some subset of the integers under the <= ordering. That is, if

```
(char<=? a b)   => #t, and
(<= x y)        => #t
```

and x and y are in the domain of integer->char, then

```
(<= (char->integer a)
   (char->integer b))=> #t
```

(**char-downcase** *char*) ⇒ *character*

This procedure return a character c such that (char-ci=? char c) is true. In addition, if char is alphabetic, then the result of char-downcase is lower case. See also char-upcase.

```
(char-downcase #\A)=> #\a
```

(**char-upcase** *char*) ⇒ *character*

This procedure returns a character c such that (char-ci=? char c) is true. In addition, if char is alphabetic, then the result of char-upcase is upper case. See also char-downcase.

```
(char-upcase #\a)=> #\A
```

(**exact->inexact** *number*) ⇒ *float*

Exact->inexact returns an inexact representation of arg. The value returned is the inexact number that is numerically closest to the argument.

For exact arguments which have no reasonably close inexact equivalent, Alter may signal an error.

This procedure implements the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

(**filename->string** *filename-type*) ⇒ *string*

Converts the argument filename into an Alter string. See also string->filename.

(**inexact->exact** *number*) ⇒ *integer*

Inexact->exact returns an exact representation of arg. The value returned is the exact number that is numerically closest to the argument.

For inexact arguments which have no reasonably close exact equivalent, Alter may signal an error.

This procedure implements the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

(**integer->char** *int*) ⇒ *character*

Given an exact integer that is the image of a character under char->integer, integer->char returns that character. This procedure implements an injective order isomorphism between the set of characters under the char<=? ordering and some subset of the integers under the <= ordering. That is, if

```
(char<=? a b)   => #t, and
(<= x y)        => #t
```

and x and y are in the domain of integer->char, then

```
(char<=? (integer->char x)
     (integer->char y))=> #t
```

(**list->string** *listofchars*) ⇒ *string*

List->string returns a newly allocated string formed from the characters in the list of args. String->list and list->string are inverses so far as equal? is concerned.

```
(list->string (list #\a #\b #\c))=> "abc"
```

(**list->vector** *list*) ⇒ *vector*

List->vector returns a newly created vector initialized to the elements of the list arg. See also vector->list.

```
(list->vector '(dididit dah))=> #(dididit dah)
```

(**number->string** *number [radix]*) ⇒ *string*

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, radix defaults to 10.

The procedure number->string takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number n)
     (radix r))
  (eqv? n
    (string->number
      (number->string n r) r)))
```

is true. It is an error if no possible result makes this expression true.

If n is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true; otherwise the format of the result is unspecified.

The result returned by number->string never contains an explicit radix prefix.

The error case can occur only when n is not a complex number or is a complex number with a non-rational real or imaginary part.

(**object->string** *object*) ⇒ *string*

Returns a string representation of sexpr. The resulting string is the same as what "display" would output.

(object->string 5)=> "5"

(object->string ’(a b c))=> "(a b c)"

(**string->filename** *string [resolve]*) ⇒ *filename*

Converts the argument string into a filename object. This procedure does not require the corresponding file to exist, but an error may result if the syntax of the name is not appropriate for the host operating system. Alternatively, certain host-specific "adjustments" may be made when converting the string to a filename that make it more palatable to the host. If the second argument is supplied, it must be the symbol ’resolve and the first argument should correspond to a relative pathname; Alter will attempt to find the specified filename along the DOME / Alter search path. If it is found, an absolute filename will be returned, otherwise it will return a relative filename. See also construct, filename->string.

(**string->list** *string*) ⇒ *list*

String->list returns a newly allocated list of the characters that make up the String->given string. String->list and list->string are inverses so far as String->equal? is concerned.

(**string->number** *string [radix]*) ⇒ *number*

Returns a number of the maximally precise representation expressed by the given string. Radix must be an exact integer, either 2, 8, 10, or 16. If supplied, radix is a default radix that may be overridden by an explicit radix prefix in string (e.g. "#o177"). If radix is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, then string->number returns #f.

(string->number "100")=> 100

(string->number "100" 16)=> 256

(string->number "1e2")=> 100.0

Although string->number is an essential procedure, an implementation may restrict its domain in the following ways. String->number is permitted to return #f whenever string contains an explicit radix prefix. If all numbers supported by an implementation are real, then string->number is permitted to return #f whenever string uses the polar or rectangular notations for complex numbers. If all numbers are integers, then string->number may return #f whenever the fractional notation is used. If all numbers are exact, then string->number may return #f whenever an exponent marker or explicit exactness prefix is used, or if a # appears in place of a digit. If all inexact numbers are integers, then string->number may return #f whenever a decimal point is used.

(**string->symbol** *string*) ⇒ *symbol*

Returns the symbol whose name is arg. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves. See symbol->string.

The following examples assume that the implementation's standard case is lower case:

(eq? 'mISSISSIppi 'mississippi)=> #t

(string->symbol "mISSISSIppi")=> the symbol with name "mISSISSIppi"

```
(eq? 'bitBlt (string->symbol "bitBlt"))=> #f
(eq? 'JollyWog
  (string->symbol
    (symbol->string 'JollyWog)))=> #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D.")))=> #t
```

(**string-capitalize** *string*) ⇒ *string*

This procedure returns a string of equal length to the argument string such that the first character of the string is now uppercase (char-ci=? is true).

(string-capitalize "abCDe")=> "AbCDe"

(**string-downcase** *string*) ⇒ *string*

This procedure returns a string of equal length to the argument string such that char-ci=? is true for each corresponding character of the argument and the result. In addition, if a given character in the argument string is alphabetic, the corresponding character in the result string is lower case. See also string-upcase.

(string-downcase "abCDe")=> "abcde"

(**string-upcase** *string*) ⇒ *string*

This procedure returns a string of equal length to the argument string such that char-ci=? is true for each corresponding character of the argument and the result. In addition, if a given character in the argument string is alphabetic, the corresponding character in the result string is upper case. See also string-downcase.

(string-upcase "abCDe")=> "ABCDE"

(**symbol->string** *symbol*) ⇒ *string*

Returns the name of symbol as a string. If the symbol was part of an object returned as the value of a literal expression or by a call to the read procedure, and its name contains alphabetic characters, then the string returned will contain characters in the implementation's preferred standard case---some implementations will prefer upper case, others lower case. If the symbol was returned by string->symbol, the case of characters in the string returned will be the same as the case in the string that was passed to string->symbol. It is an error to apply mutation procedures like string-set! to strings returned by this procedure.

The following examples assume that the implementation's standard case is lower case:

(symbol->string 'flying-fish)=> "flying-fish"

(symbol->string 'Martin)=> "martin"

(symbol->string
 (string->symbol "Malvina"))=> "Malvina"

(**type->symbol** *type*) ⇒ *symbol*

Converts a type (e.g., grapething, color-type, procedure-type) into a symbol. This is useful for dealing with DOME objects.

(**vector->list** *vector*) ⇒ *list*

Vector->list returns a newly allocated list of the objects contained in the elements of arg. See also list->vector.

(vector->list '#(dah dah didah))=> (dah dah didah)

Defining 9

(**add-method** *interface body...*) ⇒ *procedure*

Adds the specified procedure as a handler for the method named in the interface argument. Alter's implementation of classes, operations and methods follows the OakLisp style, as presented in the OOPSLA '86 Proceedings.

The detailed form of the arguments to add-method are as follows:

(add-method (operation (receiver-class) . argument-list) . body)

Add-method is a special form because the body and argument-list are not evaluated. However, the operation and receiver-class ARE evaluated. They are typically symbols bound to the objects of interest. This means, of course, that the operation must previously exist before add-method can be used to add to it (see find-operation, make). A special case is when "operation" is an unbound symbol, in that case Alter automatically applies find-operation to that symbol in the current lexical environment.

The argument-list is used to effectively build a procedure that also includes the body. This procedure is returned as the result of add-method.

When the operation is called with an instance of the receiver-class (or a subclass) as the first argument, Alter will forward the call to the procedure.

(**bindings**) ⇒ *list*

Returns a list of pairs that represent the user-defined bindings within the active lexical environment. Each pair has a car that is a symbol, and a cdr that is the value bound to that symbol. Predefined procedures and operations are NOT included in this list (see 'predefined-bindings').

(**copy** *object*) ⇒ *object*

Returns a copy of the argument.

Copying strings returns a string of equal length to the argument string such that char=? is true for each corresponding character of the argument and the result (equivalent to string-copy).

Copying lists returns a list of equal length to the argument list such that eq? returns true for each corresponding element of the argument and the result.

Copying vectors returns a vector of equal length to the argument vector such that eq? returns true for each corresponding element of the argument and the result.

Copying dictionaries returns a dictionary of equal size to the argument dictionary such that eq? returns true for each corresponding key of the argument and the result and eq? returns true for each corresponding value of the argument and the result.

Copying grapethings is dependent on how the particular subclass implements copying itself.

Copying instances of user-defined-types returns an object whose instance variables are set to a copy of the object that the argument's instance variable was set to.

Calling copy on booleans, numbers, characters returns the argument itself.

(**define** *varspec valueorbody body...*) ⇒ *object or procedure*

Definitions are valid in some, but not all, contexts where expressions are allowed. They are valid only at the top level of a program and, in some implementations, at the beginning of a body.

A definition should have one of the following forms:

```
(define variable expression)
(define (variable formals) body)
```

Formals should be either a sequence of zero or more variables, or a sequence of one or more variables followed by a space-delimited period and another variable (as in a lambda expression). This form is equivalent to

```
(define variable
  (lambda (formals) body))
```

(define (variable . formal) body)

Formal should be a single variable. This form is equivalent to

```
(define variable
  (lambda formal body))
```

At the top level of a program, a definition

```
(define variable expression)
```

has essentially the same effect as the assignment expression

```
(set! variable expression)
```

if variable is bound. If variable is not bound, however, then the definition will bind variable to a new location before performing the assignment, whereas it would be an error to perform a set! on an unbound unbound variable.

```
(define add3
 (lambda (x) (+ x 3)))
(add3 3)        => 6
(define (add2 x) (+ x 2))
(add3 3)        => 5
(define (sum . args) (apply + args))
(sum 1 2 3 4 5) => 15
(define first car)
(first '(1 2))      => 1
```

(**find-operation** *name*) ⇒ *operation*

Note: add-method has changed so that calling find-operation is usually no longer necessary.

Find-operation is a special form. The argument is a symbol that is intended to be bound to an operation. Alter first checks the current lexical environment for a binding. If one exists and the value is an operation, Alter returns that value. If one exists and it is not an operation, an error occurs. If a user-defined binding does not exist, but a predefined binding does, Alter binds the symbol in the current lexical environment to a surrogate operation that allows the user to add methods without disrupting the space of predefined symbols; a surrogate handles calls just like a normal operation, except that it can forward calls to the predefined operation if it is given an object that falls outside of its interface range.

If neither a user-defined or predefined binding exists, Alter creates a new operation and binds it to the given symbol. The return value of find-operation is the new or existing operation (or surrogate). See also add-method, make.

```
(find-operation foo)=> a new, normal operation
(find-operation foo)=> the same normal operation
(find-operation length)=> a new, surrogate operation
(find-operation +)=> error; a procedure
```

(**lambda** *formals body...*) ⇒ *procedure*

Formals should be a formal arguments list as described below, and body should be a sequence of one or more expressions.

A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The result of the last expression in the body will be returned as the result of the procedure call.

```
(lambda (x) (+ x x))=> a procedure
((lambda (x) (+ x x)) 4)=> 8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)=> 3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)          => 10
```

Formals should have one of the following forms:

```
(variable . . .)
```

The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored in the bindings of the corresponding variables.

```
variable
```

The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stored in the binding of the variable.

```
variable1 . . . variablen-1 . variablen
```

If a space-delimited period precedes the last variable, then the value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other formal arguments.

It is an error for a variable to appear more than once in formals.

```
((lambda x x) 3 4 5 6)=> (3 4 5 6)
((lambda (x y . z) z)
  3 4 5 6)          => (5 6)
```

Each procedure created as the result of evaluating a lambda expression is tagged with a storage location, in order to make eqv? and eq? work on procedures.

(**let** *bindings body...*) ⇒ *object*

Bindings should have the form

```
((variablei initi) . . .)
```

where each initi is an expression, and body should be a sequence of one or more expressions. It is an error for a variable to appear more than once in the list of variables being bound.

The inits are evaluated in the current environment (in some unspecified order), the variables are bound to fresh locations holding the results, the body is evaluated in the extended environment, and the value of the last expression of body is returned. Each binding of a variable has body as its region.

```
(let ((x 2) (y 3))
  (* x y))          => 6


(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))       => 35
```

(**let*** *bindings body...*) ⇒ *object*

Bindings should have the form

```
((variablei initi) . . .)
```

and body should be a sequence of one or more expressions.

Let* is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (variable init) is that part of the let* expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))       => 70
```

(**letrec** *bindings body...*) ⇒ *object*

Bindings should have the form

```
((variablei initi) . . .)
```

and body should be a sequence of one or more expressions. It is an error for a variable to appear more than once in the list of variables being bound.

The variables are bound to fresh locations holding undefined values, the inits are evaluated in the resulting environment (in some unspecified order), each variable is assigned to the result of the corresponding init, the body is evaluated in the resulting environment, and the value of the last expression in body is returned. Each binding of a variable has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
          #t
          (odd? (- n 1)))))
            (odd?
          (lambda (n)
            (if (zero? n)
            #f
             (even? (- n 1))))))
  (even? 88))
          => #t
```

One restriction on letrec is very important: it must be possible to evaluate each init without assigning or referring to the value of any variable. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of letrec, all the inits are lambda expressions and the restriction is satisfied automatically.

(**make** *type*) ⇒ *object*

(**make** *type ivars supertypes*) ⇒ *type*

Creates an instance of the specified type. ’Make’ can be used to create instances of type Operation, but can also be used to create instances of GrapEThing and its subclasses (e.g., DoMENode, NetArc). See also add-method, find-operation. Alter’s implementation of classes, operations and methods follows the OakLisp style, as presented in the OOPSLA ’86 Proceedings.

```
(make domenode)=> #<value: <DoMENode>>
(make operation)=> #<method13369>
(make string-type)=> ""
```

(**methods** *operation*) ⇒ *list*

Returns an alist (see assoc) containing the types and procedures currently defining the operation. Each alist component is of the form (class . procedure).

(**predefined-bindings**) ⇒ *list*

Returns a list of pairs that represent the predefined bindings. Each pair has a car that is a symbol, and a cdr that is the value bound to that symbol. User-defined procedures and operations are NOT included in this list (see ’bindings’).

(**quasiquote** *expression*) ⇒ *object*

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the <template>, the result of evaluating ‘<template> is equivalent to the result of evaluating ’<template>. If a comma appears within the <template>, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the elements of the list are inserted in place of comma at-sign expression sequence.

```
‘(list ,(+ 1 2) 4)=> (list 3 4)
(let ((name ’a)) ‘(list ,name ’,name))=> (list a (quote a))
‘(a ,(+ 1 2) ,@(map abs ’(4 -5 6)) b)=> (a 3 4 5 6 b)
‘((foo ,(- 10 3)) ,@(cdr ’(c)) . ,(car ’(cons)))=> ((foo 7) . cons)
‘#(10 5 ,(sqrt 4) ,@(map sqrt ’(16 9)) 8)=> #(10 5 2 4 3 8)
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
‘(a ‘(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          => (a ‘(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 ’x)
      (name2 ’y))
  ‘(a ‘(b ,,name1 ,’,name2 d) e))=> (a ‘(b ,x ,’y d) e)
```

The notations ‘<template> and (quasiquote <template>) are identical in all respects. ,<expression> is identical to (unquote <expression>), and ,@<expression> is identical to (unquote-splicing <expression>). The external syntax generated by write for two-element list whose car is one of these symbols may vary between implementations.

```
(quasiquote (list (unquote (+ 1 2)) 4))=> (list 3 4)
‘(quasiquote (list (unquote (+ 1 2)) 4))=> ‘(list ,(+ 1 2) 4)
```

(**quote** *object*) ⇒ *object*

Returns the argument. The argument may be any external representation of an Alter object. This notation is used to include literal constants in Alter code.

```
(quote a)          => a
(quote #(a b c))=> #(a b c)
(quote (+ 1 2)) => (+ 1 2)
```

(quote object) may be abbreviated as ’object. The two notations are equivalent in all respects.

```
’a            => a
’#(a b c)     => #(a b c)
’()           => ()
’(+ 1 2)      => (+ 1 2)
’(quote a)    => (quote a)
’’a           => (quote a)
```

Numerical constants, string constants, character constants, and boolean constants evaluate "to themselves"; they need not be quoted.

```
’"abc"        => "abc"
"abc"         => "abc"
’145932       => 145932
145932        => 145932
’#t           => #t
#t            => #t
```

It is an error to alter a constant (i.e the value of a literal expression) using a mutation procedure like set-car! or string-set!.

(**set!** *variable expression*) $\Rightarrow$ *object*

Expression is evaluated, and the resulting value is stored in the location to which variable is bound. Variable must be bound either in some region enclosing the set! expression or at top level. The result of the set! expression is unspecified.

```
(define x 2)
(+ x 1)       => 3
(set! x 4)    => unspecified
(+ x 1)       => 5
```

(**unquote** *expression*) $\Rightarrow$ *object*

See quasiquote.

(**unquote-splicing** *expression*) $\Rightarrow$ *object*

See quasiquote.

Dictionaries 10

(**dictionary->list** *dictionary*) ⇒ *list*

Creates a list that is a projection of the given dictionary. The list is of the form ((k1 . v1) (k2 . v2) ...) where each k is a key from the dictionary and each v is the corresponding value at that key. Note that the result list is in a convenient form for using with assoc, assv and assq.

```
(let ((d (make-dictionary)))
  (dictionary-set! d "A" 1)
  (dictionary-set! d "B" 2)
  (dictionary->list d))=> (("A" . 1) ("B" . 2))
```

(**dictionary-keys** *dictionary*) ⇒ *list*

Returns a list of the keys defined in the given dictionary. The keys appear in the list in no particular order, and that order may change with the addition or removal of a single key. In fact, there is no guarantee that successive calls to dictionary-keys on the same dictionary will produce the same ordering. See also make-dictionary, dictionary-ref, dictionary-set!, dictionary-unset!, dictionary-values.

```
(let ((dict (make-dictionary ’eq?)))
  (dictionary-set! dict ’alpha "one")
  (dictionary-set! dict ’beta "two")
  (dictionary-keys dict))=> ’(alpha beta)
```

(**dictionary-ref** *dictionary key [default]*) ⇒ *object*

If the given dictionary has a value associated with the given key, that value is returned. If the key has no associated value, either the default (if supplied) or nil is returned. See also make-dictionary, dictionary-set!, dictionary-keys, dictionary-unset!.

```
(define foo (make-dictionary ’eq?))
(dictionary-set! foo 10 "ten")
(dictionary-ref foo 10)=> "ten"
(dictionary-ref foo ’x)=> nil
(dictionary-ref foo ’x ’nothing)=> ’nothing
```

(**dictionary-set!** *dictionary key obj*) ⇒ *#t or #f*

Inserts the given object into the dictionary associated with the given key. If the key previously had an associated value in the dictionary, that previous association is broken and the new association is established. Dictionary-set! returns #f if there was no previous association for that key, and returns #t if there was a previous association for that key. See also make-dictionary, dictionary-ref, dictionary-keys, dictionary-unset!.

```
(define table (make-dictionary ’eq?))
(dictionary-set! table ’small ’(helvetica 9 0))=> #f
(dictionary-set! table ’small ’(times 10 0.5))=> #t
```

(**dictionary-unset!** *dictionary key*) ⇒ *#t or #f*

Removes any association that may have previously existed for the given key in the given dictionary. If an association existed for the key at the time of call to the dictionary-unset!, the procedure returns #t, otherwise it returns #f. The comparison used to match the key is either eq? or equal?, depending on how the dictionary was created (see make-dictionary). See also dictionary-ref, dictionary-set!, dictionary-keys.

```
(define sys-table (make-dictionary ’eq?))
(dictionary-unset! sys-table ’foo)=> #f
(dictionary-set! sys-table ’foo 97)
(dictionary-unset! sys-table ’foo)=> #t
```

(**dictionary-values** *dictionary*) ⇒ *list*

Returns a list of the values defined in the given dictionary. The values appear in the list in no particular order, and that order may change with the addition or removal of a single key-value pair. In fact, there is no guarantee that successive calls to dictionary-values on the same dictionary will produce the same ordering. See also make-dictionary, dictionary-ref, dictionary-set!, dictionary-unset!, dictionary-keys.

```
(let ((dict (make-dictionary ’eq?)))
  (dictionary-set! dict ’alpha "one")
  (dictionary-set! dict ’beta "two")
  (dictionary-values dict))=> ’("one" "two")
```

(**make-dictionary** *[comparison [size]]*) ⇒ *dictionary*

Make-dictionary has two optional arguments. The first, currently required to be either ’eq? or ’equal, specifies the kind of comparison used on keys to search for and retrieve values from the dictionary. (’Equal?does not currently work for keys that are lists.) ’Eq? is the default. If the second argument is also given, it must be a positive integer and indicates the allocation size of the dictionary to create. The allocation size is the number of key->value associations that may be made before the dictionary must be automatically (and invisibly) enlarged. Since dictionary enlargement can involve a lot of copying and may significantly "overshoot" in size, you may gain some performance for large dictionaries if you give a reasonably close but conservative estimate in the second argument to make-dictionary.

(define baz (make-dictionary))=> small dictionary using ’eq? for key comparison

(dictionary-set! baz "black" ’(0 0 0))

(define nob (make-dictionary ’equal?))=> small dictionary using ’equal? for key comparison

(dictionary-set! nob "black" ’(0 0 0))

(dictionary-ref bar "black")=> nil

(dictionary-ref nob "black")=> ’(0 0 0)

Document Generation 11

(**bottom-margin** *document-context*) ⇒ *inches*

Returns the size of the bottom margin in inches for a page generated by the document-context.

(**cr** *document-context [integer]*) ⇒ *nil*

Adds a carriage return (newline) to the document-context's stream. If the optional positive integer argument is passed then a number of carriage returns equal to the value of the integer are added to the document-context's stream.

(**dec-indent-level!** *document-context*) ⇒ *nil*

Decrements the document-context's indent level counter.

(**draw-grapething** *ps-context object [translation]*) ⇒ *nil*

(**draw-grapething** *document-context object [translation]*) ⇒ *nil*

Renders the grapething at the specified position on the graphics context.

(**finalize** *ps-context*) ⇒ *nil*

(**finalize** *document-context*) ⇒ *nil*

This is the call that is made to the document-context after the document has been generated. It is used to close files, etc.

(**inc-indent-level!** *document-context*) ⇒ *nil*

Increments the document-context's indent level counter.

(**indent** *document-context*) ⇒ *nil*

Adds a number of spaces equal to the value of the receiver's indent-level multiplied by its indent-size to the receiver's stream.

(**indent-level** *document-context*) ⇒ *nil*

Returns the value of the receiver's indent-level. The indent-level is multiplied by the receiver's indent-size to determine the total number of spaces to use when indenting.

(**indent-size** *document-context*) ⇒ *nil*

Returns the value of the receiver's indent-size. The indent-size is multiplied by the receiver's indent-level to determine the total number of spaces to use when indenting.

(**left-margin** *document-context*) ⇒ *inches*

Returns the size of the left margin in inches for a page generated by the receiver.

(**next-multilevel-tag** *document-context level [separator [key]]*) ⇒ *string*

Bumps the multilevel counter at the indicated level and returns a string consisting of the current state of each level separated by the specified separator string. If separator is not specified, a period "." is used. If the optional key is present, then the multilevel counter associated with that key is affected ('para is the default key). See also reset-multi-level-counter and supports-native-paragraph-numbering?

(**next-put** *document-context string*) ⇒ *nil*

Adds the string argument to the receiver's output stream.

(**open** *ps-context [output]*) ⇒ *nil*

(**open** *document-context [output]*) ⇒ *nil*

Sets the receiver's port to an output-port. If the optional output argument is not provided then a window is open and the port is set to the window. If the optional output argument is passed it must be a string, filename or output-port. If it is a string, the string is converted to a filename, the file is opened and the port is set to the resulting output-port. If it is a filename, the file is opened and the port is set to the resulting output-port. If it is an output-port, the port is set to it. An error results if any other type of argument is passed.

(**page-height** *document-context*) ⇒ *inches*

Returns the size of the height of a page generated by the receiver in inches.

(**page-width** *document-context*) ⇒ *inches*

Returns the size of the width of a page generated by the receiver in inches.

(**port** *document-context*) ⇒ *port*

Returns the receiver's current output port.

(**print-size** *document-context*) ⇒ *number*

Returns the maximum dimension in inches of printed data when drawing graphics.

(**put-string** *ps-context string [font-description]*) ⇒ *nil*

Add the string to the context's current paragraph using font if specified.

(**reset-multilevel-counter** *document-context [key]*) ⇒ *nil*

Resets the multilevel counter. If key is given, the multilevel counter associated with that key is reset (default key is 'para).

(**right-margin** *document-context*) ⇒ *inches*

Returns the size of the right margin in inches for a page generated by the receiver.

(**scale** *document-context*) ⇒ *point*

Returns a point representing the scale factor for all points used when drawing graphics.

(**scale-to!** *document-context rectangle*) ⇒ *nil*

Sets the scale and translation using the rectangle argument. The scale is set to a point whose x and y coordinates are equal to the larger of the print size divided by the rectangle's width or the print size divided by the rectangle's height. The translation is set to the rectangle's upper left corner point negated.

(**set-bottom-margin!** *document-context inches*) ⇒ *nil*

Sets the size of the bottom margin for a page generated by the document-context. The size is in inches.

(**set-face!** *document-context string*) ⇒ *nil*

Sets the name of the font family that the reliever should use when drawing strings with the draw-string operation.

(**set-indent-level!** *document-context indent-level*) ⇒ *nil*

Sets the value of the receiver's indent-level. The indent-level is multiplied by the receiver's indent-size to determine the total number of spaces to use when indenting.

(**set-indent-size!** *document-context indent-level*) ⇒ *nil*

Sets the value of the receiver's indent-size. The indent-size is multiplied by the receiver's indent-level to determine the total number of spaces to use when indenting.

(**set-left-margin!** *document-context inches*) ⇒ *nil*

Sets the size of the left margin for a page generated by the receiver. The size is in inches.

(**set-line-style!** *context symbol*) ⇒ *nil*

(**set-line-style!** *document-context symbol*) ⇒ *nil*

This operation sets the context's line style to the symbol argument. The symbol represents the current dash pattern to be used when drawing lines. The symbol is one of {normal simpledash longdash dot dashdot dashdotdot phantom chain shortdash hidden}.

(**set-line-width!** *document-context width*) ⇒ *nil*

Sets the line width of the context to the integer argument. The value indicates how many pixels wide the pen is for drawing lines.

(**set-page-height!** *document-context inches*) ⇒ *nil*

Sets the width a page generated by the receiver. The size is in inches.

(**set-page-width!** *document-context inches*) ⇒ *nil*

Sets the size of the width of a page generated by the receiver. The size is in inches.

(**set-paint-color!** *document-context color-value*) ⇒ *nil*

Given a graphics context instance, sets the context's current paint color to the color value argument. The argument is a colorvalue instance representing the current pen color for drawing objects. See make-rgb-color, make-hsb-color, and make-cmy-color for making a colorvalue instance.

(**set-paint-style!** *document-context symbol*) ⇒ *nil*

Given a graphicscontext instance, sets the context's current paint style to the symbol argument. The argument is a symbol representing the current pen drawing style ('solid or 'gray). See also set-paint-color!.

(**set-port!** *document-context port*) ⇒ *nil*

Sets the receiver's current output port to the argument.

(**set-print-size!** *document-context inches*) ⇒ *nil*

Sets the maximum dimension in inches of printed data when drawing graphics.

(**set-relative-scale!** *document-context factor*) ⇒ *nil*

Sets a value like 1.0 or 1.5 (150%), etc that can scale fonts or other values.

(**set-right-margin!** *document-context inches*) ⇒ *nil*

Sets the size of the right margin for a page generated by the receiver. The size is in inches.

(**set-scale!** *document-context point*) ⇒ *nil*

Sets the point used by the receiver to scale points when drawing graphic objects.

(**set-top-margin!** *document-context inches*) ⇒ *nil*

Sets the size of the top margin for a page generated by the document-context. The size is in inches.

(**set-translation!** *document-context point*) ⇒ *nil*

Sets the point used by the receiver to translate points when drawing graphic objects.

(**start-para** *ps-context [style]*) ⇒ *nil*

(**start-para** *document-context [style]*) ⇒ *nil*

Starts a new paragraph. If the optional style parameter is passed, a new paragraph with that style is created, otherwise the receiver's current style is used.

(**supports-native-paragraph-numbering?** *document-context*) ⇒ *boolean*

Answers #t if the given document context represents a format that has native support for multi-level paragraph numbering (e.g., Maker Interchange Format). Otherwise returns #f. See also next-multilevel-tag and reset-multilevel-counter.

(**top-margin** *document-context*) ⇒ *inches*

Returns the size of the top margin in inches for a page generated by the document-context.

(**translation** *document-context*) ⇒ *point*

Returns the point used by the receiver to translate points when drawing graphic objects.

(**write-postamble** *document-context*) ⇒ *nil*

Writes a postamble on the receiver’s stream.

(**write-preamble** *document-context*) ⇒ *nil*

Writes a preamble on the receiver’s stream.

Enumerating 12

(**assoc** *obj list*) ⇒ *pair or #f*

This procedure finds the first pair in list whose car field is obj, and returns that pair. If no pair in list has obj as its car, then #f (not the empty list) is returned. Assoc uses equal? to compare arg1 with the car fields of the pairs in list.

See also assq, assv.

```
(assoc (list 'a) '(((a)) ((b)) ((c))))=> ((a))
```

(**assq** *obj list*) ⇒ *pair or #f*

This procedure finds the first pair in list whose car field is obj, and returns that pair. If no pair in list has obj as its car, then #f (not the empty list) is returned. Assq uses eq? to compare arg1 with the car fields of the pairs in list.

See also assv, assoc.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)      => (a 1)
(assq 'b e)      => (b 2)
(assq 'd e)      => #f
(assq (list 'a) '(((a)) ((b)) ((c))))=> #f
(assq 5 '((2 3) (5 7) (11 13)))=> unspecified
```

(**assv** *obj list*) ⇒ *pair or #f*

This procedure finds the first pair in list whose car field is obj, and returns that pair. If no pair in list has obj as its car, then #f (not the empty list) is returned. Assv uses eqv? to compare arg1 with the car fields of the pairs in list.

See also assq, assoc.

```
(assv 5 '((2 3) (5 7) (11 13)))=> (5 7)
```

(**detect** *list predicate [none]*) ⇒ *object*

Detect is similar to select except that it returns the first object in the list that satisfies the predicate. If no objects satisfy the predicate then the optional none argument is returned. If the none argument is not specified then nil is returned.

(**member** *obj list*) ⇒ *list or #f*

This procedure returns the first sublist of list whose car is obj, where the sublists of list are the non-empty lists returned by (list-tail arg2 k) for k less than the length of list. If obj does not occur in list, then #f (not the empty list) is returned. Member uses equal? to compare arg with the elements of list.

See also memq, memv.

(member (list 'a) '(b (a) c))=> ((a) c)

(**memq** *obj list*) ⇒ *list or #f*

This procedure returns the first sublist of list whose car is obj, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If obj does not occur in list, then #f (not the empty list) is returned. Memq uses eq? to compare obj with the elements of list.

See also memv, member.

(memq 'a '(a b c))=> (a b c)

(memq 'b '(a b c))=> (b c)

(memq 'a '(b c d))=> #f

(memq (list 'a) '(b (a) c))=> #f

(memq 101 '(100 101 102))=> unspecified

(**memv** *obj list*) ⇒ *list or #f*

This procedure returns the first sublist of list whose car is obj, where the sublists of list are the non-empty lists returned by (list-tail list k) for k less than the length of list. If obj does not occur in list, then #f (not the empty list) is returned. Memv uses eqv? to compare obj with the elements of list.

See also memq, member.

(memv 101 '(100 101 102))=> (101 102)

(**select** *list predicate*) ⇒ *list*

Select returns a new list containing references to the top-level members of the given list that cause the specified one-argument procedure to answer a true value (a true value in Alter is any value that is not eq? with #f).

```
(select '("alpha" "beta" "gamma")
  (lambda (s) (string<? s "c")))=> '("alpha" "beta")
```

File:Modifying 13

(**copy-to** *filename destination*) ⇒ *nil*

Copies filename to destination (another filename). Destination may be a completely different pathname. It is an error if destination cannot be opened for writing or filename cannot be opened for reading.

(**delete** *filename-type*) ⇒ *filename*

Deletes the file represented by the given filename.

(**make-directory** *filename-type*) ⇒ *filename*

Causes the host to create the directory represented by filename. The behavior of this function is host-specific. Everything but the last component of filename must already exist before make-directory is called; it is an error otherwise.

(**move-to** *filename destination*) ⇒ *nil*

Renames filename to destination (another filename). Destination may be a completely different pathname, but the behavior of move-to is host-specific if filename and destination are on different devices. It is an error if destination cannot be opened for writing or filename cannot be opened for reading.

File:Naming 14

(**construct** *filename rest...*) ⇒ *filename*

Returns a new filename instance whose head is the given filename, and whose tail consists of the given string. The precise behavior of this operation is host-specific.

(**current-directory**) ⇒ *filename*

Returns a filename instance that represents the directory in which DOME was started. The behavior of this procedure is host- and installation-specific. For Macintosh installations, (current-directory) is the same as (dome-home).

(**dome-home**) ⇒ *filename*

Returns a filename instance that represents the directory containing DOME's library and support files. The behavior of this procedure is host- and installation-specific. On Unix and Windows NT systems, the pathname comes from the environment variable "DOMEHOME"; if it is not set then the start up directory is used. On DOS machines, the start up directory is used. On Macintosh systems, Alter uses the pathname of the directory containing the DOME application.

(**head** *filename-type*) ⇒ *string*

Returns a string representing the given filename with the last component removed. If the filename represents a file, the directory path is returned. If the filename represents a directory, the parent directory path is returned.

(**load-path**) ⇒ *filename*

Returns the current path used by dome to resolve filenames. This path is the concatenation of any user specified path (the value of *dome-load-path*) with the default load path.

(**resolve** *filename*) ⇒ *filename*

Returns the full path of a file if it exists along the dome-load-path, otherwise returns the filename unaltered.

(**tail** *filename-type*) ⇒ *string*

Returns a string representing the given filename with all but the last component removed.

(**temporary-filename**) ⇒ *filename*

Returns a filename that can be opened for writing to save temporary data, such as a file to print. The composition of the filename may be host-specific. The filename is guaranteed to not exist.

(**user-home**) ⇒ *filename*

Returns a filename instance that represents the home directory of the user. The behavior of this procedure is host- and installation-specific. Note: Macintosh and Windows hosts do not ordinarily support the concept of user home directories. For Macintosh installations, (user-home) is the same as (dome-home). In Windows and Unix installations, if the HOME environment variable is set, that is used to form the filename, otherwise "C:\" is used for Windows, and "/" is used for Unix.

Font Descriptions 15

(**bold** *font-description*) ⇒ *boolean*

Returns a boolean indicating whether the font is bold or not.

(**default-font-description**) ⇒ *font-description*

Returns the system's default font-description.

(**family** *font-description*) ⇒ *string*

Returns a string or a vector representing the font's family. If a vector is returned, it is a vector of strings representing alternative names for the font family.

(**fixed-width** *font-description*) ⇒ *boolean*

Returns a boolean that indicates whether all characters in the font can be expected to be the same width.

(**italic** *font-description*) ⇒ *boolean*

Returns a boolean indicating whether the font is italic or not.

(**serif** *font-description*) ⇒ *boolean*

Returns a boolean indicating whether the font is serif or not.

(**set-bold!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's bold property to the boolean argument.

(**set-family!** *font-description string*) ⇒ *nil*

Sets the receiver's family to the string argument.

(**set-fixed-width!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's fixed-width property to the boolean argument. The font's set-width property indicates whether all characters in the font can be expected to be the same width.

(**set-italic!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's italic property to the boolean argument.

(**set-serif!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's serif property to the boolean argument.

(**set-strikeout!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's strikeout property to the boolean argument.

(**set-underline!** *font-description boolean*) ⇒ *nil*

Sets the value of the font's underline property to the boolean argument.

(**size** *font-description*) ⇒ *points*

Returns the size of the font in pixels.

(**strikeout** *font-description*) ⇒ *boolean*

Returns a boolean indicating whether the font is represented with a strikeout.

(**underline** *font-description*) ⇒ *boolean*

Returns a boolean indicating whether the font is represented with an underline.

Graphics 16

(**draw-arc** *context bounding-rectangle start-angle sweep-angle [fill]*) ⇒ *nil*

(**draw-arc** *document-context bounding-rectangle start-angle sweep-angle [fill]*) ⇒ *nil*

Draws an arc on the given graphics context. If the fill-flag is true, the arc will be filled in a pie-wedge fashion. The arc is drawn to inscribe the specified rectangle. The start and sweep angles are in degrees. Zero degrees is rightward along the x axis, and the sweep angle proceeds clockwise.

(**draw-line** *context from-point to-point*) ⇒ *nil*

(**draw-line** *document-context from-point to-point*) ⇒ *nil*

Draws a line on the given graphics context between the two supplied points (see "point?"). This primitive is used mainly in user-defined DOME Tool Specification / ProtoDOME methods.

(**draw-polyline** *context point-list [fill]*) ⇒ *nil*

(**draw-polyline** *document-context point-list [fill]*) ⇒ *nil*

Draws a polyline on the given graphics context between the supplied points (see "point?"). This primitive is used mainly in user-defined DOME Tool Specification / ProtoDOME methods.

(**draw-rectangle** *context rectangle [fill]*) ⇒ *nil*

(**draw-rectangle** *document-context rectangle [fill]*) ⇒ *nil*

Draws a rectangle on the given graphics context. If the fill-flag is true, the rectangle will be filled; otherwise only the border will be drawn.

(**draw-string** *context string point*) ⇒ *nil*

(**draw-string** *document-context string alignment loc extent*) ⇒ *nil*

Renders the string at the specified position on the graphics context. The position specifies the location of the left side and baseline of the first character.

I/O 17

(**char-ready?** *[port]*) ⇒ *#t or #f*

Returns #t if a character is ready on the input port and returns #f otherwise. If char-ready returns #t then the next read-char operation on the given port is guaranteed not to hang. If the port is at end of file then char-ready? returns #t. Port may be omitted, in which case it defaults to the value returned by current-input-port.

(**close** *port*) ⇒ *nil*

Closes the file associated with port, rendering the port incapable of generating characters. This routine has no effect if the port has already been closed. The value returned is unspecified.

(**close-input-port** *port*) ⇒ *nil*

Closes the file associated with port, rendering the port incapable of generating characters. This routine has no effect if the port has already been closed. The value returned is unspecified.

(**close-output-port** *port*) ⇒ *nil*

Closes the file associated with port, rendering the port incapable of generating characters. This routine has no effect if the port has already been closed. The value returned is unspecified.

(**commit** *port*) ⇒ *nil*

Force all buffered output to be committed so no output is left buffered.

(**contents** *port*) ⇒ *string*

(**contents** *filename*) ⇒ *string*

Returns the contents of the specified filename. If the file does not exist, Alter will raise an error.

(**current-input-port**) ⇒ *input-port or nil*

Returns the current default input port. Alter does not always have a default input port, so the return value may be nil. See with-input-from-file.

(**current-output-port**) ⇒ *output-port*

Returns the current default input or output port. See with-output-to-file.

(**display** *object [port]*) ⇒ *nil*

Writes a representation of sexpr to the given port. Strings that appear in the written representation are not enclosed in doublequotes, and no characters are escaped within those strings. Character objects appear in the representation as if written by write-char instead of by write. Display returns an unspecified value. The port argument may be omitted, in which case it defaults to the value returned by current-output-port.

(**edit** *graphmodel*) ⇒ *unspecified*

(**edit** *filenameorstring*) ⇒ *nil*

Given a GraphModel instance, the edit operation forces the given graphmodel instance to be assigned an editor window (if it doesn't already have one), and then forces that window to be popped to the front. If the graph does not already have a window, the behavior of the edit operation follows what is described in the DOME User's Manual for the preferences setting 'Use Same Editor'.

Edit will accept either a string or a filename as an argument. Given a string, the edit operation treats the string as the name of a file. It attempts to open an editor on the file's contents. If the file contains a DOME model, the operation is the same as (edit (load string)). If the file contains Alter text (and the first character in the file is a semicolon), DOME will open an Alter evaluator window on the file's contents. If neither of the above is true, DOME gives the user the option of opening the file into a text editing window.

(**flush** *port*) ⇒ *nil*

Flushes any output that may have been buffered for the specified output port but not yet written to the device (either a file or window). The port is not closed. It is an error to apply flush to an output port that has already been closed.

(**load** *filenameorstring [loudly [modified]]*) ⇒ *object or grapething*

The load procedure reads expressions and definitions from the file and evaluates them sequentially. It is unspecified whether the results of the expressions are printed. The load procedure does not affect the values returned by current-input-port and current-output-port. Load returns an unspecified value. The argument may be either a string or a filename instance (see string->filename).

If the file name is a relative pathname, DOME searches a sequence of directories. The sequence is determined by the contents of the global symbol *dome-load-path* plus the default directories obtained through the expression

```
        (construct (construct (construct dome-home "tools")
"*") "lib")
```

The directories given in *dome-load-path*, if any, are searched first in order, followed by the default directories. You can set *dome-load-path* in your DOME initialization file (see the DOME User's Manual).

(**name** *namednode*) ⇒ *string*

(**name** *graphmodel*) ⇒ *string*

(**name** *user-defined-type*) ⇒ *string*

(**name** *package*) ⇒ *string*

(**name** *graphobjectattribute*) ⇒ *string*

(**name** *port*) ⇒ *string*

(**name** *nameddirarc*) ⇒ *string*

Returns a string representing the name of the given grapething instance. The name operation is defined on classes NamedNode, GraphObjectAttribute, NamedDirArc, GraphModel, and their subclasses. See also set-name!.

When given a package it returns the name of the package.

(**newline** *[port]*) ⇒ *nil*

Writes an end of line to port. Exactly how this is done differs from one operating system to another. Returns an unspecified value. The port argument may be omitted, in which case it defaults to the value returned by current-output-port.

(**open-input-file** *string*) ⇒ *input-port*

(**open-input-file** *filename*) ⇒ *input-port*

Takes a string or filename representing an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened, an error is signalled.

(**open-input-string** *string*) ⇒ *input-port*

Opens an input port on the string argument.

(**open-output-file** *string*) ⇒ *output-port*

(**open-output-file** *filename*) ⇒ *output-port*

Takes a string or filename representing an output file to be created and returns an output port capable of writing characters to a new file by that name. If the file cannot be opened, an error is signalled. If a file with the given name already exists, the effect is unspecified. If the string "" is passed as the argument then a transcript window is opened and its port is returned.

(**open-output-string**) ⇒ *output-port*

Opens an output port on the string argument.

(**peek-char** *[port]*) ⇒ *character*

Returns the next character available from the input port, without updating the port to point to the following character. If no more characters are available, an end of file object is returned. Port may be omitted, in which case it defaults to the value returned by current-input-port.

The value returned by a call to peek-char is the same as the value that would have been returned by a call to read-char with the same port. The only difference is that the very next call to read-char or peek-char on that port will return the value returned by the preceding call to peek-char. In particular, a call to peek-char on an interactive port will hang waiting for input whenever a call to read-char would have hung.

(**read** *[port]*) ⇒ *expression*

Read converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the nonterminal datum. Read returns the next object parsable from the given input arg, updating arg to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end of file object is returned.

The port argument may be omitted, in which case it defaults to the value returned by current-input-port. It is an error to read from a closed port.

(**read-char** *[port]*) ⇒ *character*

Returns the next character available from the input port, updating the port to point to the following character. If no more characters are available, an end of file object is returned. Port may be omitted, in which case it defaults to the value returned by current-input-port.

(**read-through-char** *char [port]*) ⇒ *string*

Return a string from the current position of the port to the first occurrence of char inclusive. If char is not encountered then return the entire string from port.

(**reset** *port*) ⇒ *nil*

Reset the position to the beginning.

(**with-input-from-file** *string proc*) ⇒ *object*

The file is opened for input, an input port connected to it is made the default value returned by current-input-port, and the thunk (procedure or operation) is called with no arguments. When the thunk returns, the port is closed and the previous default is restored. With-input-from-file returns the value yielded by thunk.

If an escape procedure is used to escape from the continuation of this procedure, the port is closed.

(**with-output-to-file** *string proc*) ⇒ *object*

The file is opened for output, an output port connected to it is made the default value returned by current-output-port, and the thunk (procedure or operation) is called with no arguments. When the thunk returns, the port is closed and the previous default is restored. With-output-to-file returns the value yielded by thunk.

If an escape procedure is used to escape from the continuation of these procedures, the port is closed.

If the first argument is an empty string, output is directed to a newly created transcript window.

(**with-output-to-selected** *prompt-string ok-proc [candidate-file [cancel-proc]]*) ⇒ *nil*

The user is prompted (with prompt-string) to choose where output should be directed, whether to a file or a transcript window. If the user chooses "File", then a host-specific file dialog prompts for a filename. If the optional candidate-file argument is supplied, the file dialog is initialized with that filename and directory location.

An output stream corresponding to the user's choice is created and made the default; the zero-argument ok-proc procedure is then called. When ok-proc terminates for any reason, the stream is closed, the environment's default stream is returned to its state prior to the call to with-output-to-selected.

If the user cancels the choice, the optional zero-argument cancel-proc procedure is called.

The result of with-output-to-selected is either the return value of ok-proc, the return value of cancel-proc or nil, depending on which is executed.

```
        (with-output-to-selected "Where should the output
go?"
                        (lambda () (display "hello, world")
(newline))
                        "C:\\Temp\\testout.txt"
                        (lambda () (warn "Never mind.")))
```

(**write** *object [port]*) ⇒ *nil*

Writes a written representation of sexpr to the given port. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Write returns an unspecified value. The port argument may be omitted, in which case it defaults to the value returned by current-output-port.

(**write-char** *char [port]*) ⇒ *nil*

Writes the character (not an external representation of the character) to the given port and returns an unspecified value. The port argument may be omitted, in which case it defaults to the value returned by current-output-port.

Lists 18

(car *list***)** ⇒ *object*

Returns the contents of the car field of the pair. Note that it is an error to take the car of the empty list.

```
(car '(a b c))      => a
(car '((a) b c d))=> (a)
(car '(1 . 2))      => 1
(car '())           => error
```

(cdr *list***)** ⇒ *object*

Returns the contents of the cdr field of the pair. Note that it is an error to take the cdr of the empty list.

```
(cdr '((a) b c d))=> (b c d)
(cdr '(1 . 2))      => 2
(cdr '())           => error
```

(cons *obj1 obj2***)** ⇒ *pair*

Returns a newly allocated pair whose car is obj1 and whose cdr is obj2. The pair is guaranteed to be different (in the sense of eqv?) from every existing object.

```
(cons 'a '())       => (a)
(cons '(a) '(b c d))=> ((a) b c d)
(cons "a" '(b c))=> ("a" b c)
(cons 'a 3)         => (a . 3)
(cons '(a b) 'c)  => ((a b) . c)
```

(list *items...***)** ⇒ *list*

Returns a newly allocated list of its arguments.

```
(list 'a (+ 3 4) 'c)=> (a 7 c)
(list)              => ()
```

(list-head *list index***)** ⇒ *list*

Returns a copy of the list including only the first k elements. List-tail could be defined by

```
(define list-head
  (lambda (x k)
    (if (or (zero? k) (null? x))
      (list)
      (set-cdr! (list (car x)) (list-head (cdr x) (- k 1))))))
```

(list-last *list***)** ⇒ *sexpr*

Returns the last element of list. (This is the same as (list-ref list (- (length list) 1)).)

```
(list-ref '(a b c d)) => d
(list-ref '()) => nil
```

(list-ref *list index*) ⇒ *sexpr*

Returns the kth element of list. (This is the same as the car of (list-tail list k).)

```
(list-ref '(a b c d) 2)=> c
(list-ref '(a b c d)
  (inexact->exact (round 1.8)))=> c
```

(list-tail *list index*) ⇒ *list*

Returns the sublist of list obtained by omitting the first k elements. List-tail could be defined by

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
      x
      (list-tail (cdr x) (- k 1)))))
```

(remove-from-list! *list object*) ⇒ *list*

Returns the argument list with all top-level references to object removed (using eq? test). Note that the return value will not be the same as the list passed in if the first element is eq? to object. This operation is destructive to the argument list, so any references to list may also be affected. The structure and values of the resulting list are the same as defined for copy-without.

```
(define foo '(a b c))
(remove-from-list! foo 'b)=> (a c)
foo                 => (a c)

(define foo '(a b a c))
(remove-from-list! foo 'a)=> (b c)
foo                 => (a b c)

(remove-from-list! '(a b c b) 'b)=> (a c)
(remove-from-list! '(b) 'b)=> ()
(remove-from-list! '(a b . c) 'b)=> (a . c)
(remove-from-list! '(b . c) 'b)=> (nil . c)
(remove-from-list! '(nil nil nil . c) 'nil)=> (nil . c)
(remove-from-list! '("a" "b" "c") "b")=> ("a" "b" "c")
(remove-from-list! '(a (b c)) 'b)=> (a (b c))
```

(**reverse** *list*) ⇒ *list*

Returns a newly allocated list consisting of the elements of list in reverse order.

```
(reverse '(a b c))=> (c b a)
(reverse '(a (b c) d (e (f))))=> ((e (f)) d (b c) a)
```

(**set-car!** *list obj*) ⇒ *object*

Stores arg in the car field of the pair. The value returned by set-car! is unspecified.

```
(define f (list 'not-a-constant-list))
(define g '(constant-list))
(set-car! f 3)    => unspecified
(set-car! g 3)    => error
```

(**set-cdr!** *list obj*) ⇒ *object*

Stores arg in the cdr field of pair. The value returned by set-cdr! is unspecified.

(**sort** *list proc*) ⇒ *list*

Return a new list that contains the same elements but ordered by the comparison procedure. The comparison procedure should accept two parameters and return #t if the first parameter should be ordered before the second, #f otherwise.

```
(sort '(12 56 2 100 8) (lambda (a b) (<= a b)))
=> (2 8 12 56 100)
(sort '(("foo" . 56) ("bar" . 100) ("zip" . 8))
    (lambda (a b) (<= (cdr a) (cdr b))))
=> (("zip" . 8) ("foo" . 56) ("bar" . 100))
(sort '(("foo" . 56) ("bar" . 100) ("zip" . 8))
    (lambda (a b) (string<=? (car a) (car b))))
=> (("bar" . 100) ("foo" . 56) ("zip" . 8))
```

Sort will fail to terminate if it is given a circular list. The sort procedure uses a quicksort algorithm.

Logic 19

(**and** *args...*) ⇒ *object or #f*

The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

```
(and (= 2 2) (> 2 1))=> #t
(and (= 2 2) (< 2 1))=> #t
(and 1 2 'c '(f g))=> #t
(and)           => #t
```

(**not** *object*) ⇒ *#t or #f*

Not returns #t if obj is false, and returns #f otherwise.

```
(not #t)        => #f
(not 3)         => #f
(not (list 3))  => #f
(not #f)        => #t
(not '())       => #f
(not (list))    => #f
(not 'nil)      => #f
```

(**or** *expressions...*) ⇒ *object or #f*

The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

```
(or (= 2 2) (> 2 1))=> #t
(or (= 2 2) (< 2 1))=> #t
(or #f #f #f)   => #f
(or (memq 'b '(a b c))
    (/ 3 0))    => (b c)
```

Math Functions 20

(**acos** *number*) ⇒ *number*

This procedure computes the usual transcendental function.

(**asin** *number*) ⇒ *number*

This procedure computes the usual transcendental function.

(**atan** *y [x]*) ⇒ *number*

This procedure computes the usual transcendental function.

(**cos** *number*) ⇒ *number*

This procedure computes the usual transcendental function.

(**exp** *number*) ⇒ *number*

Exp computes e^number. The result is a real number.

(**expt** *base exponent*) ⇒ *number*

Returns base raised to the power exponent: number^exponent = e^(exponent log base).

0^0 is defined to be equal to 1.

(**log** *number*) ⇒ *number*

Log computes the natural logarithm of arg (not the base ten logarithm).

(**sin** *number*) ⇒ *number*

This procedure computes the usual transcendental function.

(**tan** *number*) ⇒ *number*

This procedure computes the usual transcendental function.

Miscellaneous 21

(**category** *thunk [category]*) ⇒ *string*

Given a procedure or operation, returns its category as a string. This category is where the procedure or operation can be found in the various browsers that are part of the Projector / Alter programming environment. If a second optional argument is given, it must be a string, and it is used to set the category of the procedure or operation.

(**date-today**) ⇒ *vector*

Returns a list containing a representation of the current system date. The list is interpreted as follows: (year day-of-year).

(**dates** *filename*) ⇒ *dictionary*

Returns a dictionary with the file's access dates.

(**dome-version**) ⇒ *string*

Returns a string that represents the version of DOME being used.

(**error** *error-message*) ⇒ *string*

This procedure stops execution and displays the Alter Environment Browser (Activation Stack) with the argument as the text in the message box.

(**interface** *proc*) ⇒ *list*

Given a procedure, interface returns a string representing its interface. Given an operation, interface returns a list of the classes that the operation is defined on.

(**message-to-user** *message*) ⇒ *string*

Writes the specified string to the DOME message pane in the DOME Launcher window, followed by a newline.

(**platform**) ⇒ *association*

Return an association where the association is made up of a symbol representing the kind of operating system of the platform ('win32, 'unix, 'mac) and a string that provides more detail about the platform such as version or detailed operating system name.

(**random** *[low high]*) ⇒ *number*

Returns a random number. With no arguments, the returned value is a floating point number evenly distributed between 0.0 and 1.0. With two integer arguments, the returned value is an integer evenly distributed between the two given values (inclusive). The first argument must be less than the second argument.

(**reset-environment**) ⇒

Resets the registry, default input port, default output port and execution stack of the current environment to their default values.

(**return-spec** *proc*) ⇒ *string*

Returns a string representing the type of object typically returned by the given procedure.

(**sleep** *milliseconds*) ⇒ *nil*

Causes the DOME process to go inactive for the specified number of milliseconds. This is useful for animation and other related purposes.

(**time-now**) ⇒ *vector*

Returns a time object containing a representation of the current system time. See time->list and time.

Model:Accessing 22

(**accessories** *graphobject*) ⇒ *list*

Answer a list of nodes that are accessories of the node. Accessories are nodes that are usually attached to the boundary of the node.

(**add-binding-named** *graphmodel configuration graphobject*) ⇒ *modelbinding*

(**add-binding-named** *graphobject configuration graphmodel*) ⇒ *modelbinding*

For a graphobject add a binding with the given name from the graphobject to the specified graphmodel.

For a graphmodel add a binding with the given name from the specified graphobject to the graphmodel.

(**add-child** *graphobject graphmodel*) ⇒ *graphmodel*

Checks to see if aGraphModel is a subdiagram of aGraph-Object. If it is NOT a subdiagram it adds aGraphModel to aGraphObject's subdiagrams and returns aGraphModel. Otherwise returns nil.

(**archetype** *domenode*) ⇒ *node*

(**archetype** *graphobject*) ⇒ *graphobject*

Given an instance of a node, the archetype operation returns either nil (if the node has no archetype), or the node instance that serves as the argument's archetype. An archetype is a node that resides on the shelf (see the DOME User's Manual). By definition, the archetype of an archetype is itself.

(**archetype-shelf** *graphmodel*) ⇒ *graphmodel*

Answer the archetype page associated with the graphmodel.

(**archetype?** *graphmodel*) ⇒ *#t or #f*

(**archetype?** *remotegraphobject*) ⇒ *#t or #f*

(**archetype?** *graphobject*) ⇒ *#t or #f*

Is the object an archetype?

(**archetypifiable?** *graphobject*) ⇒ *#t or #f*

Answer #t if instances of the grapething's class can be archetypes.

(**arcs** *graphmodel*) ⇒ *list*

Given a GraphModel instance, the arcs operation returns a list of the arcs present in the graph. The list includes only the arcs at the top level within the given graph; it does NOT contain any arcs from subdiagrams. See also nodes.

(**baseline** *graphmodel*) ⇒ *integer*

(**baseline** *graphobject*) ⇒ *integer*

Answer the distance from the top of the line to the bottom of most of the characters (by convention, bottom of A) in the current style used by the model to render text.

(**binding-named** *grapething string*) ⇒ *modelbinding*

Answer the binding whose configuration is as specified.

(**border-bounds** *domenode*) ⇒ *rectangle*

Returns a rectangle (list of the form ((x1 . y1) . (x2 . y2)) that is used as a starting point for drawing the object's outline. Units are pixels.

(**bounds** *grapething*) ⇒ *rectangle*

Returns a rectangle (see "rectangle?") that represents the node's outer dimensions, including the name, if any. Units are pixels.

(**color** *font-description*) ⇒ *color-value*

(**color** *graphobject*) ⇒ *color-value*

Returns a color-value representing the color of the font or graph object.

(**components** *grapething*) ⇒ *list*

Given a GrapeThing instance, the components operation returns a list of the components contained within the instance. For GraphModels, the list will contain nodes and arcs. For nodes, the list will contain nodes. For arcs, the list may contain nodes and a GrapEArcName.

(**configurations** *graphmodel*) ⇒ *list*

(**configurations** *grapething*) ⇒ *list*

Answer the configurations that are available to be used throughout the model.

(**container** *graphobject*) ⇒ *visualgrapething*

The container operation is an inverse to the components method. Given a GrapEThing instance, it returns the object that contains it. The returned object may be a DoMENode, NetArc or GraphModel instance. See components.

(**current-binding** *graphmodel*) ⇒ *modelbinding*

Answer the modelbinding that matches the graphmodel's current state.

(**current-configuration** *graphmodel*) ⇒ *string*

Answer the configuration string that the graphmodel has active.

(**default-child-type** *graphobject*) ⇒ *graphmodel-class*

Returns the default type of graphmodel created when a new subdiagram is added.

(**description** *grapething*) ⇒ *string*

(**description** *proc [string]*) ⇒ *string*

If the first argument is an instance of GrapEThing, description returns the value (a string) of the description property of the object. If the first argument is a procedure or operation, and the optional second argument is missing, description returns a string describing it. Normally the description text is retrieved from the file (construct (construct (dome-home) 'lib') 'alterdsc.txt'), but if the optional argument is supplied, it becomes the description for the procedure or operation.

(**destination** *netarc*) ⇒ *node*

(**destination** *graphobjectreference*) ⇒ *node*

Given an arc, the destination operation returns the node object that is at the destination end of the arc. See also origin.

Given a GraphObjectReference, destination returns the node for which the argument is a surrogate.

(**direction** *graphboundary*) ⇒ *symbol*

Returns a symbol representing the semantic direction of the given boundary node in a graph. The possible values are 'in, 'out and 'inout.

(**do-over-model** *grapething thunk*) ⇒ *nil*

Traverse the grapething's model and apply the thunk to each object contained in the model.

(**elements** *graphobject*) ⇒ *list*

Answer a list of nodes that are elements of the node. Elements are nodes that are contained within the node.

(**filename** *graphmodel*) ⇒ *filename-type*

Given a graph model instance, the filename operation returns an instance of filename-type that represents the file the model was last saved in.

(**find-vacant-position** *graphmodel width height*) ⇒ *point*

Answer a position (point) in the graph that can contain an object with the given width and width.

(**frozen-color** *graphobject*) ⇒ *#t or #f*

Answer #t if the color of the graphobject should not change through indirect means.

(**get-property** *object propname*) ⇒ *object*

(**get-property** *propname object*) ⇒ *object*

Gets the value of the named property from the given object. The object is usually an instance of grapething or one of its subclasses, but get-property will also work on instances of other GrapE classes. It is an error if the property name is not valid for the given class of object. The DOME Tool Specifications define the properties on the various DOME objects. The property name which IS case sensitive can either by a string or a symbol.

(**get-property-definition** *grapething-class propertyname*) ⇒ *propertydefintion or nil*

(**get-property-definition** *grapething propertyname*) ⇒ *propertydefinition or nil*

Return the property definition for the named property.

(**get-property-definitions** *grapething-class*) ⇒ *dictionary*

(**get-property-definitions** *grapething [categoryname]*) ⇒ *dictionary*

Return all (or those that have a category of categoryname) of the property definitions for the specified object.

(**graph** *grapething*) ⇒ *graphmodel*

Returns the graphmodel containing the object, if any. Returns nil if the object cannot be traced back to any graph. The graph of a graphmodel is itself.

(**graph-label** *graphmodel*) ⇒ *node*

Answer the graphlabel of the graphmodel.

(**has-archetype?** *graphobject*) ⇒ *#t or #f*

Answer #t if the graphobject has an archetype.

(**has-binding-named?** *grapething string*) ⇒ *#t or #f*

Answer #t if the grapething has a modelbinding with a configuration as specified.

(**has-binding?** *grapething*) ⇒ *#t or #f*

Answer #t if the grapething has at least one modelbinding.

(**has-child?** *graphobject*) ⇒ *#t or #f*

Answer #t if the graphobject has subdiagrams.

(**has-parent-connection?** *grapething*) ⇒ *#t or #f*

Answer #t if the grapething has a parent connection.

(**has-parent?** *grapething*) ⇒ *#t or #f*

Answer #t if the grapething has a parent

(**has-property-set?** *grapething propname*) ⇒ *boolean*

(**has-property-set?** *propname grapething*) ⇒ *boolean*

Returns #t if the given grapething object has a value set for the named property. It is an error if the property name is not valid for the given class of object. The DOME Tool Specifications define the properties on the various DOME objects. The property name which IS case sensitive can either by a string or a symbol.

(**identifier** *grapething*) ⇒ *number*

Answer a number that uniquely identifies the grapething in the model.

(**implementations** *graphobject*) ⇒ *list*

If aGraphObject has an archetype then this returns the archetypes's implementations. Otherwise it returns aGraphObject's subdiagrams.

(**incoming-arcs** *domenode*) ⇒ *list*

Given a node instance, the incoming-arcs operation returns a list of the arcs entering the node. See also outgoing-arcs.

(**instances** *domenode*) ⇒ *list*

The argument to the instances operation must be an archetype, that is, a node that resides on the shelf (see the DOME User's Manual). The instances operation will return a list of all of the archetype's instances. See also archetype.

(**logical-top-graph** *graphmodel*) ⇒ *graphmodel*

Answer the logical top graphmodel. The logical top graphmodel corresponds to the topmost graphmodel in a hierarchy whose subtree of graphmodels forms one 'model', that is, a cohesive set of (interrelated) graphmodels that are all from the same notation family.

(**master** *grapething*) ⇒ *grapething*

Returns the instance that serves as the master for the given object. The following expression will always be true:

```
(memq self (views (master self)))
```

where self is a graphobject instance and (not (nil? (master self))).

(**name-emphasis** *graphobject*) ⇒ *symbol*

Returns a symbol describing the current style of the object's title ('normal, 'italic, 'bold, 'underline or 'strikeout). See also set-name-emphasis!

(**name-set!** *user-defined-type string*) ⇒ *string*

(**name-set!** *graphmodel string*) ⇒ *string*

(**name-set!** *namednode string*) ⇒ *string*

(**name-set!** *graphobjectattribute string*) ⇒ *string*

(**name-set!** *nameddirarc string*) ⇒ *string*

Obsolete. Use set-name! instead.

(**name-source** *grapething*) ⇒ *grapething*

Answer the grapething from which the name is really coming from.

(**nodes** *graphmodel*) ⇒ *list*

The nodes operation retrieves the list of nodes contained in the given graphmodel instance. Only the nodes at the top level within the given graphmodel are returned; the list does NOT contain any nodes from subdiagrams. See also arcs.

(**open-models**) ⇒ *list*

This procedure returns a list of graphmodel instances that are currently open (but not necessarily displayed) within DOME.

(**origin** *netarc*) ⇒ *node*

Given an arc, the origin operation returns the node object that is at the origin end of the arc. See also destination.

(**outgoing-arcs** *domenode*) ⇒ *list*

Given a node instance, the outgoing-arcs operation returns a list of the arcs emanating from the node. See also incoming-arcs.

(**parent** *graphmodel*) ⇒ *graphobject*

Returns the parent of the given graph, if any. If the argument has no parent, nil is returned.

(**parent-connection** *graphobject*) ⇒ *graphobject*

Parent-connection is an operation defined on nodes and arcs that returns the corresponding object in the parent diagram (if any). For example, the parent-connection of a boundary point on a DFD graph is an arc whose destination is the process node being refined. In some cases, the parent-connection may be part of an archetype.

(**position** *graphobject*) ⇒ *point*

Returns the absolute position (in pixels) of the given object as a point (x . y).

(**property-schema-files** *grapething*) ⇒ *list*

Answer the property schema files property which is a list of filenames that represent UDP models.

(**rationale** *grapething*) ⇒ *string*

Returns the value of the rationale property of the given grapething instance. The returned value is a string (possibly empty).

(**relative-position** *domenode*) ⇒ *point*

Returns the relative position (in pixels) of the given node as a point (x . y). The returned value is relative to the object's container's position. If the container is a GraphModel instance, the container's position is taken to be (0 . 0).

(**remove** *grapething*) ⇒ *nil*

Deletes the specified grapething from its container and removes all other references to it.

(**remove-child** *graphobject graphmodel*) ⇒ *graphmodel*

Checks to see if aGraphModel is a subdiagram of aGraph-Object. If it is a subdiagram it removes aGraphModel from aGraphObject's subdiagrams and returns aGraphModel. Otherwise returns nil.

(**resolve-identity** *remotegraphobject*) ⇒ *grapething*

Attempts to resolve the reference to the remote DOME object. If successful, the operation answers the remote object. If unsuccessful, DOME raises an error dialog and the operation returns nil. During the attempt to resolve the identity, DOME may cause the file containing the object to be loaded into memory.

(**route** *netarc*) ⇒ *list*

Returns a list of points of the form (x . y) that represent where the given arc bends. The endpoints (where the arc attaches to the origin and destination nodes) are not included in the list, therefore it may be the case that route returns an empty list. Units are pixels.

(**selected-components** *grapething*) ⇒ *list*

Answer a list of graphobjects that are currently selected in the graphmodel.

(**set-border-bounds!** *domenode rectangle*) ⇒ *rectangle*

Sets the rectangular border bounds of the given node to be the specified rectangle. See "rectangle?" for a specification of the representation of a rectangle.

(**set-color!** *font-description color-value*) ⇒ *nil*

(**set-color!** *graphobject color*) ⇒ *unspecified*

Sets the color of the receiver to the color-value argument.

(**set-configurations!** *graphmodel list*) ⇒ *unspecified*

Set the configurations that are available to be used throughout the model.

(**set-container!** *grapething new-container*) ⇒ *#t or #f*

Make grapething a component of new-container. See components.

(**set-current-binding!** *graphmodel modelbinding*) ⇒ *unspecified*

Set the graphmodel's current modelbinding to the specified modelbinding.

(**set-current-configuration!** *graphmodel string*) ⇒ *unspecified*

Set the configuration string that the graphmodel has active.

(**set-description!** *grapething string*) ⇒ *unspecified*

Set the description of the object.

(**set-destination!** *netarc node*) ⇒ *unspecified*

Set the destination of the netarc.

(**set-direction!** *graphboundary symbol*) ⇒ *unspecified*

Set the direction of the graphboundary.

(**set-frozen-color!** *graphobject boolean*) ⇒ *unspecified*

Set the frozen color property of the graphobject. The frozen color property prevents the color of the graphobject from changing through indirect means.

(**set-master!** *grapething grapething*) ⇒ *unspecified*

The specified grapething should serve as the master of the grapething.

(**set-name!** *graphmodel string*) ⇒ *unspecified*
(**set-name!** *namednode string*) ⇒ *unspecified*
(**set-name!** *graphobjectattribute string*) ⇒ *unspecified*
(**set-name!** *nameddirarc string*) ⇒ *unspecified*

Set the name of the object.

(**set-name-emphasis!** *graphobject symbol*) ⇒ *unspecified*

Set the textual emphasis of the object's title string (which may actually be more than the name). The detailed appearance depends on the fonts available from the platform. Valid symbols are 'normal, 'italic, 'bold, 'underline and 'strikeout. If given an invalid symbol, the text will be displayed as with 'normal. See also name-emphasis.

(**set-origin!** *netarc node*) ⇒ *unspecified*

Set the origin of the netarc.

(**set-parent-connection!** *graphobject parent*) ⇒ *child-object*

Set the parent connection of the node or arc to the corresponding object in the parent diagram.

(**set-position!** *domenode point*) ⇒ *point*

Sets the absolution position of the node to be the point, if possible. Units are in pixels, and the point is a pair of the form (x . y). This operation can also be applied to arc name tags and other accessories. It is an error to apply this operation to instances of GraphObjectAttribute, since their positions are determined completely automatically, and are strictly relative to their containers. See also relative-position-set! and synchronize-display.

(**set-property!** *object propname value*) ⇒ *nil*

(**set-property!** *propname object value*) ⇒ *nil*

Sets the value of the propnamestring property on the grapething object to the specified value. Set-property may work on non-GrapEThing objects, but its behavior is unspecified in those cases. It is an error if the property name is not valid for the given class of object. The DOME Tool Specifications define the properties on the various DOME objects. The property name which IS case sensitive can either by a string or a symbol.

(**set-property-schema-files!** *grapething list*) ⇒ *unspecified*

Set the property schema files property to the specified list of filenames.

(**set-rationale!** *grapething string*) ⇒ *unspecified*

Set the rationale of the object.

(**set-relative-position!** *domenode point*) ⇒ *point*

Sets the relative position of the node to be the point, if possible. Units are in pixels and the point is a pair of the form (x . y). The position given is relative to the node's container (possibly the graph, whose position is considered to be (0 . 0). Some subclasses of DoMENode may generate an error if set-relative-position! is applied to them. In particular, subclasses of GraphObjectAttribute typically resist attempts to manually adjust their relative positions. See also set-position! and synchronize-display.

(**set-route!** *netarc list*) ⇒ *unspecified*

Set the route of the netarc. The route is a list of points.

(**set-size!** *font-description number*) ⇒ *nil*

(**set-size!** *domenode point*) ⇒ *point*

Given a font-description and an integer, sets the size of the font in pixels.

Given a node and a point, sets the width and height of the node using the x and y coordinates of the point. The bounds of the node are recomputed to be centered around its current position.

(**subdiagrams** *graphobject*) ⇒ *list*

Returns a list of graphmodel instances that are children of the given graphobject instance. The returned list may be empty. See also parent-connection.

(**text-line-height** *graphobject*) ⇒ *integer*

Return the space between lines.

(**top-model** *grapething*) ⇒ *grapething*

Answer the graphmodel that is the root of all other graphmodels in the model.

(**unset-property!** *grapething propname [value]*) ⇒ *nil*

(**unset-property!** *propname grapething [value]*) ⇒ *nil*

Removes any value bound to the specified property for the given object. If the optional third argument is supplied, the value is unbound only if the current binding is eq? to the third argument. It is an error if the property name is not valid for the given class of object. The DOME Tool Specifications define the properties on the various DOME objects. The property name which IS case sensitive can either by a string or a symbol.

(**views** *grapething*) ⇒ *list*

Returns a list of views (at the next level only) of the given grapething instance. The elements of the returned list will be of the same type as the argument supplied to the views operation. If there are no views, an empty list will be returned. See also master. The following express will always be true:

```
(memq self (views (master self)))
```

where self is some graphobject instance and (not (nil? (master self)))

(**what-are-you** *grapething*) ⇒ *string*

Given a grapething instance, the what-are-you operation returns a string describing the instance (e.g., a user-sensible rendering of its class name). This is mainly useful for interacting with the user.

Model:Creation 23

(**merge-file** *graphmodel file*) ⇒ *nil*

Merge the model contained in the file (String or Filename) into the graphmodel.

(**new-child-model** *graphmodel-class graph-object*) ⇒ *graphmodel*

(**new-child-model** *metadomegraph graphobject*) ⇒ *graphmodel*

(**new-child-model** *protodomespecwrapper graph-object*) ⇒ *graphmodel*

Creates a new instance of the given graphmodel class as a child of the specified graphobject (node or arc). New-child-model does NOT open an editor window on the resulting instance. The 'edit' operation will do this. See also new-top-model, new-in, nodes and arcs.

(**new-in** *metadomenodespec graph [position]*) ⇒ *node*

(**new-in** *domenode-class graph [position]*) ⇒ *node*

(**new-in** *metadomearcspec graph originnode destinationnode [route]*) ⇒ *arc*

(**new-in** *netarc-class graph originnode destinationnode [route]*) ⇒ *arc*

New-in is an operation defined on node classes and arc classes, and is used for creating new instances of nodes and arcs. Both methods require the second argument to be a graphmodel instance that is to contain the new object.

The node class method accepts a third (optional) argument that specifies the pixel coordinates for the new node; if it is omitted, GrapE will use a default position that depends on the type of node. The position coordinate must be supplied as a pair whose car is the x coordinate and whose cdr is the y coordinate (y=0 is the top of the window). Units are pixels.

The third and fourth arguments to the arc class method must be node instances. The third argument specifies the node that will serve as the new arc's origin; the fourth argument specifies the node that will serve as the new arc's destination. It is legal for the origin and destination to be the same object. The fifth (optional) argument must be a list of points (pairs) that serve as the intermediate route (bend) points for the arc, proceeding from the origin side to the destination side. If no route it supplied, the arc will be a straight line automatically clipped to the boundaries of the origin and destination node. The form of each point is a pair whose car is the x coordinate and whose cdr is the y coordinate. Units are pixels, and y=0 is the top of the editing area.

(define g (new-top-model DFDGraph))

(define n1 (new-in DFDProcess g '(100 . 100)))=> a process node (define n2 (new-in DFDProcess g '(300 . 100)))=> a process node

(new-in DFDDataflow g n1 n2 '((200 . 50)))=> a dataflow arc with one bend point

(edit g)

(**new-top-model** *graphmodel-class*) ⇒ *graphmodel*

Creates a new instance of the given graphmodel class and initializes it as a new top-level model (i.e., it has no parent). New-top-model does NOT open an editor window on the resulting instance. The edit operation will do this. See also new-child-model, nodes, arcs, new-in.

(**write-bitmap** *graphmodel filename format*) ⇒ *nil*

Writes a bitmap of the graph into the specified filename using the indicated format. Supported formats are currently 'gif, 'mif (Maker Interchange Format; 'frame and 'framemaker are aliases), 'xwd and 'eps (Encapsulated PostScript; 'epsf is an alias).

Model:Dependencies 24

(**add-interest** *grapething aspect lambda interestedobject [role]*) ⇒ *nil*

Create a dependency on the first argument so that an action is triggered when it changes. Specifically, the given lambda expression is executed every time the specified "aspect" of the object changes. The lambda expression is passed the changed object, the aspect that changed (e.g., name) and the previous value (e.g., "foo") as parameters.

Two additional parameters are given to add-interest to make it easier to remove the dependencies. The first of these, the fourth argument to add-interest, is the dependent. In typical usage, the lambda expression will execute some behavior on the dependent, such as updating a property or re-propagating the change in another direction. If the dependent goes away for some reason, DOME will automatically remove the dependencies registered against it.

If a dependent needs to be able to selectively remove dependencies, the optional sixth argument may be used to create such dependencies. This same symbol can be given to remove-interest.

The second (aspect) and optional fifth (role) arguments may be either strings or symbols.

Example: Display a message to the user when the description changes:

```
        (add-interest self 'description
                        (lambda (changed-object changed-
aspect old-value)
                        (message-to-user
                        (append old-value " -> "
                        (description changed-object))))
                        self)
```

See also remove-interest.

(**remove-interest** *grapething aspect interestedobject [role]*) ⇒ *nil*

Remove a previously created dependency on the first argument. The second argument indicates the aspect or property of interest; all dependencies observing that aspect and whose dependent matches the third argument are removed. If the optional fourth argument is supplied, the dependencies must also have been created with that role.

If none of the dependencies match the pattern derived from the arguments, no

dependencies are removed.

Example: Remove the interest expressed earlier (see add-interest)

```
(remove-interest self ’description self)
```

See also add-interest.

Model:Generating 25

(**document-generator-named** *grapething string*) ⇒ *generator*

Return the first document generators applicable to the receiver object whose name matches the string argument.

See also document-generators

(**document-generators** *grapething*) ⇒ *list*

Return the document generators applicable to the receiver object (GrapEThing).

See also document-generator-named

(**generate** *domegeneratorspec subject settings*) ⇒ *unspecified*

Run the named document generator on the subject object passed and given settings contained in the settings object.

Below is an simple example which outlines how this might be used:

```
(define test-generate
  (lambda (obj)
    (let ( (generator nil) (settings nil) )
      (set! generator (car (document-generators obj)))
      (set! settings (new-settings generator))
      (set-property!  "outputType" settings ’window)
      (set-property!  "outputFormat" settings "Text")
      (generate generator obj settings)
    )
  )
)
```

See also document-generators and new-settings.

(**new-settings** *domegeneratorspec*) ⇒ *settings-object*

Create and return a new settings object, the properties of which determine how and where a generator will operate. Common properties of the settings object include:

’filename

’outputFormat (’Text ’MIF)

’outputType (’window ’file ’directory)

See also generate

Model:Testing 26

(**any-changes?** *graphmodel*) ⇒ *#t or #f*

Answer #t if the graphmodel or any graphmodel in the model has any changes.

(**has-name?** *grapething*) ⇒ *#t or #f*

Answer #t if the grapething has a name.

(**input-and-output?** *graphboundary*) ⇒ *#t or #f*

Answer #t if the graphboundary can be used as both an input and an output.

(**input-only?** *graphboundary*) ⇒ *#t or #f*

Answer #t if the graphboundary can be used only as an input.

(**input?** *graphboundary*) ⇒ *#t or #f*

Answer #t if the graphboundary can be used as an input.

(**is-kind-of?** *grapething class*) ⇒ *#t or #f*

Given an object and a class, is-kind-of? returns #t if the object is an instance of the class, and returns #f if it is not. Both the object must be an instance of some non-basic class (a basic class is one of the Scheme predefined types, e.g., integer, symbol, string). Is-kind-of? is not defined on instances of basic classes.

(**logical-top-graph?** *graphmodel*) ⇒ *#t or #f*

Answer #t if the graphmodel is the logical top graph. See logical-top-graph.

(**moveable?** *graphobject*) ⇒ *#t or #f*

Answer #t if the user may interactive move the graphobject.

(**output-only?** *graphboundary*) ⇒ *#t or #f*

Answer #t if the graphboundary can be used only as an output.

(**output?** *graphboundary*) ⇒ *#t or #f*

Answer #t if the graphboundary can be used as an output.

(**resizable?** *graphobject*) ⇒ *#t or #f*

Answer #t if the node may be resized by the user.

(**selected?** *graphobject*) ⇒ *#t or #f*

Answer #t if the graphobject is selected.

(**top-model?** *graphmodel*) ⇒ *#t or #f*

Answer #t if the graphmodel is the top model.

(**visible?** *graphobject*) ⇒ *#t or #f*

Answer #t if the graphobject is visible.

Model:User-Interface 27

(**bring-to-focus** *grapevisualthing*) ⇒ *unspecified*

Raise / open the graphmodel that has the grapething and then select the grapething.

(**clear-selection** *graphobject*) ⇒ *unspecified*

De-select the graphobject.

(**close-model** *graphmodel*) ⇒ *unspecified*

Confirm any potential loss of data with the user and then close all editor windows open on any portion of the model.

(**data-dictionary-edit** *grapething*) ⇒ *unspecified*

Edit the grapething using the data dictionary editor.

(**deletion-request?** *grapething*) ⇒ *#t or #f*

Answer #t if the user should be allowed to delete this grapething.

(**deselect-all** *graphmodel*) ⇒ *unspecified*

De-select all graphobjects in the graphmodel.

(**display-errors** *list*) ⇒ *nil*

The display-errors procedure displays the given list of errors in a scrollable window so that the object with a problem can be directly inspected or focussed on. The list is an association list where the car of each association is a GrapEVisualThing or a list of GrapEVisualThings, and the cdr is an error message (a string).

```
(define g (new-top-model DFDGraph))
(display-errors (list (list g "Name Required")))
```

(**edit-name** *grapevisualthing*) ⇒ *unspecified*

Open a dialog to allow the user to rename the object.

(**inspector-edit** *grapevisualthing*) ⇒ *unspecified*

Edit the grapething using the inspector editor.

(**move-to-back** *graphobject*) ⇒ *unspecified*

Move the graphobject behind all other graphobjects.

(**move-to-front** *graphobject*) ⇒ *unspecified*

Move the graphobject in front of all other graphobjects.

(**print** *graphmodel*) ⇒ *unspecified*

Open a dialog to allow the user to print the graphmodel.

(**refresh-display** *grapevisualthing*) ⇒ *unspecified*

Redraw the given object.

(**save** *graphmodel*) $\Rightarrow$ *unspecified*

Save the graphmodel. If the graphmodel has never been saved then open a dialog to allow the user to save the graphmodel.

(**save-as** *graphmodel*) $\Rightarrow$ *unspecified*

Open a dialog to allow the user to save the graphmodel.

(**set-selection** *graphobject*) $\Rightarrow$ *unspecified*

Select the graphobject.

(**square-route** *netarc*) $\Rightarrow$ *unspecified*

Square the route of the netarc.

(**synchronize-display** *graphmodel*) $\Rightarrow$ *nil*

If the given graphmodel has an editor window open, synchronize-display brings it up to date by forcing all pending graphic operations to complete. The precise behavior of this operation is host-specific.

Modules 28

(**loaded-modules**) ⇒ *list*

Returns a list of the modules currently loaded in the system.

(**provide** *module*) ⇒ *boolean*

Each module has a unique name (a string). The provide and require functions accept either a string or a symbol as the module-name argument. If a symbol is provided, its print-name is used as the module name.

The provide procedure adds a new module name to the list of modules, thereby indicating that the module in question has been loaded.

(**require** *module*) ⇒ *boolean*

Each module has a unique name (a string). The provide and require functions accept either a string or a symbol as the module-name argument. If a symbol is provided, its print-name is used as the module name.

The require function tests whether a module is already present; if the module is not present, require proceeds to load the appropriate file or set of files. The pathname argument, if present, is a single filename that is to be loaded. If the pathname argument is not provided, the system will attempt to determine which files to load by searching along the DOME load-path as follows:

- first, it will look for a file named module-name,
- next, it will look for a file named module-name.alt,
- last, it will look for a file named module-name.lib.

Alter also records the modified timestamp of a module's file when it loads it. If the module has already been loaded, Alter will check the current timestamp of the file. If it is later than the stored timestamp then Alter will re-load the module.

OS Interface 29

(**shell** *command args...*) ⇒ *integer*

Synchronously executes the command with the string args and returns its exit status as an integer. Zero usually means success; non-zero means failure.

Packages 30

(**all-packages**) ⇒ *list*

Returns a list of all packages currently defined in the system.

(**current-package**) ⇒ *package*

Answer the package that symbols are searched for and created in by default.

(**delete-package** *package-name*) ⇒ *package*

The package argument may be either a package or the name of a package. If the package specified for deletion is currently used by other packages, unuse-package is performed on all such packages so as to remove their dependency on the specified package.

(**export** *symbol [package]*) ⇒ *package*

The symbols argument should be a list of symbols, or possibly a single symbol. These symbols become accessible as external symbols in package.

(**exported-symbols** *package-name*) ⇒ *list*

Returns a list of all symbols exported by package.

(**find-package** *package-name*) ⇒ *package*

The package argument should be a string or symbol. The package with argument as a name is returned; if no such package exists an error is signaled. If the argument is a package object then the argument is returned.

(**import** *symbols exporting-package [importing-package]*) ⇒ *package*

The symbols argument should be a list of symbols, or possibly a single symbol. The exporting-package argument is the package that the symbols are being imported from. These symbols become internal symbols in importing-package and can therefore be referred to without having to use qualified-name (colon) syntax.

(**in-package** *package-name*) ⇒ *package*

This procedure currently does nothing except yield a nil return value. It is included for compatibility with emacs lisp modes.

(**make-package** *package-name*) ⇒ *package*

This creates and returns a new package with the specified package name. The argument may be either a string or a symbol.

(**package-name** *package*) ⇒ *string*

Answer the name of the package.

(**package-use-list** *package*) ⇒ *package*

Returns a list of packages used by package.

(**package-used-by-list** *package*) ⇒ *package*

Returns a list of packages the use package.

(**rename-package** *package new-package-name*) ⇒ *package*

The new argument should be a string or symbol. The package argument is a string, symbol or package object. The old name is eliminated and replaced by new.

(**unexport** *symbol-or-list [package]*) ⇒ *package*

The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in package.

(**unuse-package** *package-to-unuse [unusing-package]*) ⇒ *package*

The packages-to-unuse argument should be a list of packages or packages names, or possibly a single package or package name. These packages are removed from the use-list of package.

(**use-package** *package-to-use [using-package]*) ⇒ *used-package*

The packages-to-use argument should be a list of packages or package names, or possibly a single package or package name. These packages are added to the use-list of package if they are not there already. All external symbols in the packages to use become accessible in package as internal symbols.

Points 31

(**point*** *point1 point2...*) ⇒ *point*

This procedure returns the product of its argument points. See "point?" for an explanation of how points are represented in Alter. This procedure is obsolete; use * which now accepts both numbers and points.

(point* ’(5 . 2) ’(3 . 4))=> ’(15 . 8)

(point* #(5 2) #(3 4))=> #(15 8)

(point* ’(5 . 2) 3)=> ’(15 . 6)

(point* #(5 2) 4 ’(1 . 2))=> #(20 16)

(**point+** *point1 point2...*) ⇒ *point*

This procedure returns the sum of its arguments, where the first argument is a point and the rest of the arguments are points or numbers. See "point?" for an explanation of how points are represented in Alter. This procedure is obsolete; use + which now accepts both numbers and points.

(point+ ’(1 . 2) ’(3 . 4))=> ’(4 . 6)

(point+ #(1 2) #(3 4))=> #(4 6)

(point+ ’(1 . 2) 3 4)=> ’(8 . 9)

(point+ #(1 2) 3 4)=> #(8 9)

(**point-** *point1 point2...*) ⇒ *point*

This procedure returns the difference of its arguments, where the first argument is a point and the rest of the arguments are points or numbers. See "point?" for an explanation of how points are represented in Alter. This procedure is obsolete; use - which now accepts both numbers and points.

(point+ ’(5 . 2) ’(3 . 4))=> ’(2 . -2)

(point- #(5 2) #(3 4))=> #(2 -2)

(point- ’(5 . 2) 3 4)=> ’(-2 . -5)

(point- #(5 2) 3 4)=> #(-2 -5)

(**point-max** *point rest...*) ⇒ *point*

Returns the lower right corner of the rectangle that contains all of the points passed as arguments. This function is obsolete; use max which now accepts both numbers and points.

(point-min ’(2 . 2) ’(1 . 3))=> ’(2 . 3)

(point-min #(3 2) #(3 1) #(2 4))=> ’(3 . 4)

(**point-min** *point rest...*) ⇒ *point*

Returns the upper left corner of the rectangle that contains all of the points passed as arguments. This procedure is obsolete; use min which now accepts both numbers and points.

(point-min '(2 . 2) '(1 . 3))=> '(1 . 2)

(point-min #(3 2) #(3 1) #(2 4))=> '(2 . 1)

(**point/** *point1 point2...*) ⇒ *point*

This procedure returns the quotient of its argument points. See "point?" for an explanation of how points are represented in Alter. This procedure is obsolete; use / which now accepts both numbers and points.

(point/ '(6 . 2) '(3 . 2))=> '(2 . 1)

(point/ #(6 2) #(3 2))=> #(2 1)

(point/ '(16 . 32) 4)=> '(4 . 8)

(point/ #(40 40) 4 '(1 . 2))=> #(10 5)

(**point<** *pt1 pt2 pt3...*) ⇒ *#t or #f*

Returns #t if both the x and y coordinates of the first point are less than the x and y coordinates of the second point (see "point?"). Otherwise returns #f. This procedure is obsolete; use < which now accepts both numbers and points.

(point< '(3 . 2) '(4 . 5))=> #t

(point< '(3 . 2) '(2 . 2))=> #f

(**point<=** *pt1 pt2 pt3...*) ⇒ *#t or #f*

Returns #t if both the x and y coordinates of the first point are less than or equal to the x and y coordinates of the second point (see "point?"). Otherwise returns #f. This procedure is obsolete; use <= which now accepts both numbers and points.

(point<= '(3 . 2) '(4 . 2))=> #t

(point<= '(3 . 2) '(3 . 1))=> #f

(**point>** *pt1 pt2 pt3...*) ⇒ *#t or #f*

Returns #t if both the x and y coordinates of the first point are greater than the x and y coordinates of the second point (see "point?"). Otherwise returns #f. This procedure is obsolete; use > which now accepts both numbers and points.

(point> '(3 . 2) '(2 . 2))=> #f

(point> '(3 . 2) '(2 . 1))=> #t

(**point>=** *pt1 pt2 pt3...*) ⇒ *#t or #f*

Returns #t if both the x and y coordinates of the first point are greater than or equal to the x and y coordinates of the second point (see "point?"). Otherwise returns #f. This procedure is obsolete; use >= which now accepts both numbers and points.

```
(point>= '(3 . 2) '(2 . 2))=> #t
(point>= '(3 . 2) '(2 . 3))=> #f
```

(**radius** *point*) ⇒ *number*

Answer the polar coordinate system radius of the given point.

(**theta** *point*) ⇒ *number*

Returns the angular component (in radians) of the ray represented by the given point (see "point?").

```
(theta '(2 . 0))   => 0.0
(theta '(5 . 5))   => 0.785398
(theta '(-1 . 3))  => 1.89255
(theta '(2 . -2))  => 5.49779
```

(**x** *point*) ⇒ *number*

Returns the x-coordinate of a point (see "point?").

```
(x '(3 . 9))       => 3
(x #(2.5 7))       => 2.5
```

(**y** *point*) ⇒ *number*

Returns the y-coordinate of a point (see "point?").

```
(y '(3 . 9))       => 9
(y #(2.5 7))       => 7
```

Printer Driver 32

(**background-color** *alter-graphics-context*) ⇒ *colorvalue*

Given a graphics context instance, the background-color operation returns a color value object that represent the current background color used by the context.

(**face** *alter-graphics-context*) ⇒ *string*

(**face** *grapething*) ⇒ *string*

(**face** *document-context*) ⇒ *string*

Given a graphics context instance, the face operation returns a string representing the current typeface for displaying text objects. This operation is useful primarily in user-defined DOME printer drivers. (See appendix.) The possible return values are: "Helvetica", "Times", "Courier" and "Palatino".

(**landscape** *alter-graphics-context*) ⇒ *#t or #f*

Given a graphics context instance, the landscape operation returns a boolean value indicating whether or not the user selected landscape orientation in the print dialog. This operation is useful primarily in user-defined DOME printer drivers. (See appendix.)

(**line-style** *alter-graphics-context*) ⇒ *symbol*

(**line-style** *context*) ⇒ *symbol*

This operation returns a symbol representing the current dash pattern to be used when drawing lines. The symbol returned is one of {normal simpledash longdash dot dashdot dashdotdot phantom chain shortdash hidden}.

(**line-width** *alter-graphics-context*) ⇒ *integer*

Given a graphics context instance, the line-width operation returns an integer value indicating how many pixels wide the pen is for drawing lines. This operation is useful primarily in user-defined DOME printer drivers. (See appendix.)

(**paint** *alter-graphics-context*) ⇒ *array*

Given a graphics context instance, the paint operation returns a two-element vector; the first value is a colorvalue instance representing the current pen color for drawing objects and the second value is a symbol representing the pen stroke style ('solid or 'gray). This operation is useful primarily in user-defined DOME printer drivers. (See appendix.) See also paintcolor, paintstyle. The result of the paint operation is the same as the following (where gc is the graphicscontext instance):

```
(let ((p (make-vector 2)))
(vector-set! p 1 (paintcolor gc))
(vector-set! p 2 (paintstyle gc))
p)
```

(**paint-color** *alter-graphics-context*) ⇒ *colorvalue*

Given a graphics context instance, the paint-color operation returns a colorvalue instance representing the current pen color for drawing objects. This operation is useful primarily in user-defined DOME printer drivers. (See appendix.) See red, green, blue, hue, saturation, brightness, cyan, magenta and yellow operations for extracting color components from a colorvalue instance.

(**paint-style** *alter-graphics-context*) ⇒ *symbol*

Given a graphicscontext instance, returns a symbol representing the current pen drawing style (’solid or ’gray). This operation is useful primarily for writing user-defined DOME printer drivers (see appendix). See also paintcolor, paint.

(**relative-scale** *graphmodel*) ⇒ *number*

(**relative-scale** *alter-graphics-context*) ⇒ *number*

Returns a value like 1.0 or 1.5 (150%), etc that can scale fonts or other values.

Rectangles 33

(**center** *rectangle*) ⇒ *point*

Returns the point that lines at the center of the supplied rectangle. See "rectangle?" for a specification of how rectangles are represented.

(center '((3 . 8) . (5 . 12)))=> '(4 . 10)

(center #(#(3 8) #(5 12)))=> '(4 . 10)

(**expand-rectangle** *rectangle expansion*) ⇒ *rectangle*

Returns a new rectangle which represents the given rectangle expanded by the specified expansion. The expansion may be specified as a number, a point or a rectangle. If it is a number, all sides of the rectangle are moved outward by the specified amount. If it is a point, then the left and right sides are expanded outward by the amount specified in the x component of the point, and the top and bottom sides are expanded outward by the amount specified in the y component of the point. If the expansion is a rectangle, then the left, top, right and bottom sides of the first argument are expanded by the amounts specified for the left, top, right and bottom of the expansion.

(expand-rectangle '((3 . 8) . (5 . 12)) 1)=> '((2 . 7) . (6 . 13))

(expand-rectangle '((3 . 8) . (5 . 12)) '(1 . 2))=> '((2 . 6) . (6 . 14))

(expand-rectangle '((3 . 8) . (5 . 12)) '((1 . 2) . (3 . 4))=> '((2 . 6) . (8 . 16))

(**extent** *rectangle*) ⇒ *number*

Returns the extent (a point whose x represents the width and whose y represents the height) of the supplied rectangle (see "rectangle?").

(extent '((3 . 8) . (5 . 12)))=> '(2 . 4)

(**height** *rectangle*) ⇒ *number*

Returns the extent of the image in the y (vertical) direction.

(**lower-left** *rectangle*) ⇒ *number*

Returns the lower left corner of the rectangle (see "rectangle?").

(lower-left '((3 . 8) . (5 . 12)))=> '(3 . 12)

(lower-left #(#(3 8) #(5 12)))=> '(3 . 12)

(**lower-right** *rectangle*) ⇒ *number*

Returns the lower right corner of the rectangle (see "rectangle?").

(lower-right '((3 . 8) . (5 . 12)))=> '(5 . 12)

(lower-right #(#(3 8) #(5 12)))=> #(5 12)

(**scale-rectangle** *rectangle scale*) ⇒ *rectangle*

Returns a new rectangle which represents the given rectangle scaled by the specified amount. The scale is specified as a point or number. (See "rectangle?").

(scale-rectangle '((3 . 8) . (5 . 12)) '(1 . 2))=> '((3 . 16) . (5 . 24))

(scale-rectangle '((3 . 8) . (5 . 12)) 5)=> '((15 . 40) . (25 . 60))

(**translate-rectangle** *rectangle translation*) ⇒ *rectangle*

Returns a new rectangle which represents the given rectangle translated by the specified amount. The translation is specified as a point. (See "rectangle?"). The height and width of the new rectangle are the same as the argument.

(translate-rectangle '((3 . 8) . (5 . 12)) '(1 . 2))=> '((4 . 10) . (6 . 14))

(**upper-left** *rectangle*) ⇒ *number*

Returns the upper left corner of the rectangle (see "rectangle?").

(upper-left '((3 . 8) . (5 . 12)))=> '(3 . 8)

(upper-left #(#(3 8) #(5 12)))=> #(3 8)

(**upper-right** *rectangle*) ⇒ *number*

Returns the upper right corner of the rectangle (see "rectangle?").

(upper-right '((3 . 8) . (5 . 12)))=> '(5 . 8)

(upper-right #(#(3 8) #(5 12)))=> '(5 . 8)

(**width** *rectangle*) ⇒ *number*

If given a rectangle, returns the width (in the x dimension) of the supplied rectangle (see "rectangle?"). If given a string and a graphics context, returns the width of the string in pixels based on the font currently installed on the graphics context.

(width '((3 . 8) . (5 . 12)))=> 2

(width #(#(3 8) #(5 12)))=> 2

(width “A” context)=> depends on current font

Registry 34

(**identity** *identifier*) ⇒ *object*

Return the object from the registry that has the given identifier.

(**register** *object*) ⇒ *identifier*

Add the object to the registry and return an identifier that may be used later for lookup.

(**unregister** *identifier*) ⇒ *nil*

Remove the identifier and its associated object from the registry.

Rpc 35

(**open-client-socket** *hostname port*) ⇒ *input-output-port*

Open a connection to the given host on the given port.

Smalltalk 36

(**smalltalk** *method object arguments...*) ⇒ *list*

Provides a means of invoking an arbitrary Smalltalk method. This is mainly used when prototyping DOME tools using a VisualWorks / Smalltalk environment. It can also serve as a means of accessing some DOME infrastructure features that are not yet revealed via Alter, though this is not recommended. The first argument to the method is the receiver object. The second argument is a string encoding the method handle, e.g., "name:". The remaining arguments are supplied as arguments to the Smalltalk method. The result of the Smalltalk method is returned as the result of this Alter primitive.

Alter checks the method handle to make sure there are the correct number of supplemental arguments. For example, "name:" requires a single argument, whereas "in:at:" requires two, and "class" requires none.

Strings 37

(**make-string** *length [char]*) ⇒ *string*

Make-string returns a newly allocated string of the specified length. If char is given, then all elements of the string are initialized to char, otherwise the contents of the string are unspecified.

(**string** *char rest...*) ⇒ *string*

Returns a newly allocated string composed of the arguments.

(**string-append** *string1 string2...*) ⇒ *string*

Returns a newly allocated string whose characters form the concatenation of the given strings.

```
(string-append "a" "bc" "de")=> "abcde"
```

(**string-copy** *string*) ⇒ *string*

Returns a newly allocated copy of the given string.

(**string-fill!** *string char*) ⇒ *string*

Stores arg2 in every element of the given arg1 and returns an unspecified value. See also make-string.

(**string-length** *string*) ⇒ *integer*

Returns the number of characters in the given string. See also length.

(**string-ref** *string index*) ⇒ *character*

String-ref returns the indexth character of string using zero-origin indexing.

(**string-set!** *string index char*) ⇒ *string*

String-set! stores char in the indexth position of string and returns an unspecified value. String-set! uses zero-origin indexing.

```
(define (f) (make-string 3 \#*))
(define (g) "***")
(string-set! (f) 0 #\?)=> unspecified
(string-set! (g) 0 #\?)=> error
(string-set! (symbol->string 'immutable) 0 #\?)=>
error
```

(**substring** *string start end*) ⇒ *string*

Substring returns a newly allocated string formed from the characters of string beginning with index start (inclusive) and ending with index end (exclusive).

```
(substring "abcdefg" 2 4)=> "cd"
```

(substring "abc" 2 4)=> error

(substring "abc" 0 3)=> "abc"

(**substring-index** *string substring [start]*) ⇒ *number-or-nil*

Returns the zero-based index of the first occurrence of the specified substring in the argument string. If start is specified, the search begins at that index rather than at the beginning. Nil is returned if the substring does not occur in the searched range, or if the specified start index is past the end of the string.

(substring-index "abcdefg" "cd")=> 2

(substring-index "abcdabcd" "cd" 4)=> 6

(substring-index "qrs" "cd")=> nil

Testing 38

(**=** *num1 num2 num3...*) ⇒ *#t or #f*

This procedures returns #t if its arguments are equal. = is transitive.

The traditional implementations of = in Lisp-like languages are not transitive.

While it is not an error to compare inexact numbers using =, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of =.

(**boolean?** *object*) ⇒ *#t or #f*

Boolean? returns #t if sexpr is either #t or #f and returns #f otherwise.

```
(boolean? #f)   => #t
(boolean? 0)    => #f
(boolean? '())  => #f
```

(**bound?** *symbol*) ⇒ *#t or #f*

Returns #t if the symbol is bound to a value in the current lexical environment, otherwise returns #f.

(**char-alphabetic?** *char*) ⇒ *#t or #f*

This procedure returns #t if its argument is alphabetic, otherwise it returns #f. The alphabetic characters are the 52 upper and lower case letters. See also char-numeric?, char-whitespace?, char-upper-case? and char-lower-case?.

(**char-ci<=?** *arg1 arg2*) ⇒ *#t or #f*

This procedure is similar to char<=?, but treats upper case and lower case letters as the same. For example, (char-ci<=? #\a #\B) returns #t, whereas (char<=? #\a #\B) returns #f.

(**char-ci<?** *arg1 arg2*) ⇒ *#t or #f*

This procedure is similar to char<?, but treats upper case and lower case letters as the same. For example, (char-ci<? #\a #\B) returns #t, whereas (char<? #\a #\B) returns #f.

(**char-ci=?** *arg1 arg2*) ⇒ *#t or #f*

This procedure is similar to char=?, but treats upper case and lower case letters as the same. For example, (char-ci=? #\A #\a) returns #t.

(**char-ci>=?** *arg1 arg2*) ⇒ *#t or #f*

This procedure is similar to char>=?, but treats upper case and lower case letters as the same. For example, (char-ci>=? #\a #\B) returns #t, whereas (char>=? #\a #\B) returns #f.

(**char-ci>?** *arg1 arg2*) ⇒ *#t or #f*

This procedure is similar to char>?, but treats upper case and lower case letters as the same. For example, (char-ci>? #\A #\b) returns #t, whereas (char>? #\A #\b) returns #f.

(**char-lower-case?** *char*) ⇒ *#t or #f*

The procedures return #t if its argument is a lower case character, otherwise it returns #f. The alphabetic characters are the 52 upper and lower case letters. See also char-whitespace?, char-numeric?, char-whitespace? and char-upper-case?.

(**char-numeric?** *char*) ⇒ *#t or #f*

This procedure returns #t if its argument is numeric, otherwise it returns #f. The numeric characters are the ten decimal digits. See also char-alphabetic?, char-whitespace?, char-upper-case? and char-lower-case?.

(**char-upper-case?** *char*) ⇒ *#t or #f*

This procedure returns #t if its argument is upper case, otherwise it returns #f. The alphabetic characters are the 52 upper and lower case letters. See also char-whitespace?, char-numeric?, char-whitespace? and char-lower-case?

(**char-whitespace?** *char*) ⇒ *#t or #f*

This procedure returns #t if its argument is whitespace, otherwise it returns #f. The whitespace characters are space, tab, line feed, form feed and carriage return. See also char-alphabetic?, char-numeric?, char-upper-case? and char-lower-case?

(**char<=?** *arg1 arg2*) ⇒ *#t or #f*

Returns #t if the character represented by the first argument is the same as or precedes the character represented by the second argument, assuming a total ordering on the set of characters.

(**char<?** *arg1 arg2*) ⇒ *#t or #f*

Returns #t if the character represented by the first argument precedes the character represented by the second argument, assuming a total ordering on the set of characters.

(**char=?** *arg1 arg2*) ⇒ *#t or #f*

Returns #t if the two arguments represent the same character.

(**char>=?** *arg1 arg2*) ⇒ *#t or #f*

Returns #t if the character represented by the first argument is the same as or follows the character represented by the second argument, assuming a total ordering on the set of characters.

(**char>?** *arg1 arg2*) ⇒ *#t or #f*

Returns #t if the character represented by the first argument follows the character represented by the second argument, assuming a total ordering on the set of characters.

(**char?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a character, otherwise returns #f.

(**class?** *object*) ⇒ *#t or #f*

Returns true if the given object represents a grapething-class or one of

its subclasses; returns false otherwise.

```
(class? 5)          => #f
(class? integer-type)=> #t
(class? grapething)=> #t
(class? 'node)   => #f
(class? (make type '() '()))=> #f
```

(**color?** *object*) ⇒ *#t or #f*

Returns true if the given object is an instance of color-type; returns false otherwise. See palette, colors.

(**dictionary?** *object*) ⇒ *#t or #f*

Returns true if the given object is a dictionary instance (see make-dictionary); returns false otherwise.

(**directory?** *filename-type*) ⇒ *#t or #f*

Returns #t if the filename represents a directory on the host system. Returns #f otherwise.

(**eof-object?** *object*) ⇒ *#t or #f*

Returns #t if arg is an end of file object, otherwise returns #f. The precise set of end of file objects will vary among implementations, but in any case no end of file object will ever be an object that can be read in using read.

(**eq?** *arg1 arg2*) ⇒ *#t or #f*

Eq? is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?.

Eq? and eqv? are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, and non-empty strings and vectors. Eq?'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when eqv? would also return true. Eq? may also behave differently from eqv? on empty vectors and empty strings.

```
(eq? 'a 'a)         => #t
(eq? '(a) '(a))    => unspecified
```

```
(eq? (list 'a)
    (list 'a))      => #f
(eq? "a" "a")       => unspecified
(eq? "" "")         => unspecified
(eq? '() '())       => #t
(eq? 2 2)           => unspecified
(eq? #\A #\A)       => unspecified
(eq? car car)       => #t
(let ((n (+ 2 3)))
  (eq? n n))        => unspecified
(let ((x '(a)))
  (eq? x x))        => #t
(let ((x '#()))
  (eq? x x))        => #t
(let ((p (lambda (x) x)))
  (eq? p p))        => #t
```

(**equal?** *arg1 arg2*) ⇒ *#t or #f*

Equal? recursively compares the contents of pairs, vectors, and strings, applying eqv? on other objects such as numbers and symbols. A rule of thumb is that objects are generally equal? if they print the same. Equal? may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)      => #t
(equal? '(a) '(a))=> #t
(equal? '(a (b) c)
      '(a (b) c))   => #t
(equal? "abc" "abc")=> #t
(equal? 2 2)        => #t
(equal? (make-vector 5 'a)
      (make-vector 5 'a))=> #t
(equal? (lambda (x) x)
      (lambda (y) y))=> unspecified
```

(**eqv?** *arg1 arg2*) ⇒ *#t or #f*

The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if arg1 and arg2 should normally be regarded as the same object. This relation is left slightly open to interpretation, but the following partial specification of eqv? holds for all implementations of Scheme.

The eqv? procedure returns #t if:

arg1 and arg2 are both #t or both #f.

arg1 and arg2 are both symbols and

```
(string=?
  (symbol->string arg1)
  (symbol->string arg2))
        => #t
```

Note: This assumes that neither arg1 nor arg2 is an ‘‘uninterned symbol”.

arg1 and arg2 are both numbers, are numerically equal (see =), and are either both exact or both inexact.

arg1 and arg2 are both characters and are the same character according to the char=? procedure both arg1 and arg2 are the empty list.

arg1 and arg2 are pairs, vectors, or strings that denote the same locations in the store.

arg1 and arg2 are procedures whose location tags are equal

The eqv? procedure returns #f if:

arg1 and arg2 are of different types

one of arg1 and arg2 is #t but the other is #f

arg1 and arg2 are symbols but

```
(string=?
  (symbol->string arg1)
  (symbol->string arg2))
        => #f
```

one of arg1 and arg2 is an exact number but the other is an inexact number.

arg1 and arg2 are numbers for which the = procedure returns #f.

arg1 and arg2 are characters for which the char=? procedure returns #f.

one of arg1 and arg2 is the empty list but the other is not.

arg1 and arg2 are pairs, vectors, or strings that denote distinct locations.

arg1 and arg2 are procedures that would behave differently (return a different value or have different side effects) for some arguments.

```
(eqv? ’a ’a)        => #t
(eqv? ’a ’b)        => #f
(eqv? 2 2)          => #t
(eqv? ’() ’())      => #t
(eqv? 100000000 100000000)=> #t
(eqv? (cons 1 2) (cons 1 2))=> #f
```

```
(eqv? (lambda () 1)
      (lambda () 2))=> #f
(eqv? #f 'nil)    => #f
(let ((p (lambda (x) x)))
     (eqv? p p))  => #t
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of eqv?. All that can be said about such cases is that the value returned by eqv? must be a boolean.

```
(eqv? "" "")      => unspecified
(eqv? '#() '#())  => unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))=> unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))=> unspecified
```

The next set of examples shows the use of eqv? with procedures that have local state. Gen-counter must return a distinct procedure every time, since each procedure has its own internal counter. Gen-loser, however, returns equivalent procedures each time, since the local state does not affect the value or side effects of the procedures.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))=> #t
(eqv? (gen-counter) (gen-counter))=> #f

(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser))) (eqv? g g))=> #t
(eqv? (gen-loser) (gen-loser))=> unspecified

(letrec
  ((f (lambda () (if (eqv? f g) 'both 'f)))
   (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))=> unspecified
```

```
(letrec
  ((f (lambda () (if (eqv? f g) 'f 'both)))
   (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))=> #f
```

Since it is an error to modify constant objects (those returned by literal expressions), implementations are permitted, though not required, to share structure between constants where appropriate. Thus the value of eqv? on constants is sometimes implementation-dependent.

```
(eqv? '(a) '(a))  => unspecified
(eqv? "a" "a")   => unspecified
(eqv? '(b) (cdr '(a b)))=> unspecified
(let ((x '(a))) (eqv? x x))=> #t
```

The above definition of eqv? allows implementations latitude in their treatment of procedures and literals: implementations are free either to detect or to fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

(**even?** *object*) ⇒ *#t or #f*

This procedure returns #t if its argument is even (a multiple of 2). The result for inexact numbers may be unreliable because of small inaccuracies.

(**exact?** *object*) ⇒ *#t or #f*

Exact? provides a test for the exactness of a quantity. For any Alter number, either exact? or inexact? is true, but not both.

(**exists** *filename-type*) ⇒ *#t or #f*

Returns #t if the specified file exists, otherwise returns #f.

(**exists?** *filename-type*) ⇒ *#t or #f*

Returns #t if the specified file exists, otherwise returns #f.

(**grape?** *object*) ⇒ *#t or #f*

Returns true if the given object is an instance of grapething or one of its many subclasses. Returns false otherwise.

(**inexact?** *object*) ⇒ *#t or #f*

Exact? provides a test for the exactness of a quantity. For any Alter number, either exact? or inexact? is true, but not both.

(**input-port?** *object*) ⇒ *#t or #f*

Returns #t if the given port is an input port, otherwise returns #f. See open-input-file.

(**integer?** *object*) ⇒ *#t or #f*

Integer? can be applied to any kind of argument, including non-numbers. It returns #t if the object is an integer, and otherwise it returns #f.

If integer? is true of a number then all higher type predicates are also true of that number. Consequently, if integer? is false of a number, then all lower type predicates are also false of that number.

If x is an inexact real number, then (integer? x) is true if and only if (= x (round x)).

```
(integer? 3.0)   => #t
(integer? 8/4)  => #t
```

The behavior of integer? on inexact numbers is unreliable, since any inaccuracy may affect the result.

See also number?, complex?, real?, and rational?

(**is-a?** *object type*) ⇒ *#t or #f*

Returns #t if object is an instance of type or one of its sub-types. (is-a? object object-type) is always true.

```
(is-a? object-type type)=> #t
(is-a? type object-type)=> #t
(is-a? operation type)=> #t
(is-a? is-a? procedure-type)=> #t
(is-a? is-a? type)=> #f
(is-a? (make object-type) type)=> #f
```

(**list?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a list, otherwise returns #f. By definition, all lists have finite length and are terminated by the empty list. For this reason, it may take list? a fair amount of time to determine its answer if given a very long list.

```
(list? '(a b c))    => #t
(list? '())         => #t
(list? '(a . b))    => #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))        => #f
```

(**negative?** *object*) ⇒ *#t or #f*

This procedure returns #t if its argument is less than zero. The result for inexact numbers may be unreliable because of small inaccuracies.

(**nil?** *object*) ⇒ *#t or #f*

Returns true if the given object is eq? to the nil object (there is only one in Alter). Returns false otherwise.

(**null?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is the empty list, otherwise returns #f.

(**number?** *object*) ⇒ *#t or #f*

Number? can be applied to any kind of argument, including non-numbers. It returns #t if the object is a number, and otherwise it returns #f.

```
(number? 'a)    => #f
(number? 3)     => #t
(number? 3.1415)=> #t
```

(**object?** *object*) ⇒ *#t or #f*

Returns true for any Alter object.

(**odd?** *object*) ⇒ *#t or #f*

This procedure returns #t if its argument is odd (exactly halfway between two adjacent multiples of 2). The result for inexact numbers may be unreliable because of small inaccuracies.

(**operation?** *object*) ⇒ *#t or #f*

Returns true if the given object represents an operation, as defined by make or find-operation; returns false otherwise.

```
(operation? new-in)=> #t
(operation? car)=> #f
```

(**output-port?** *object*) ⇒ *#t or #f*

Returns #t if the given port is an output port, otherwise returns #f. See open-output-file and terminal.

(**pair?** *object*) ⇒ *#t or #f*

Pair? returns #t if sexpr is a pair, and otherwise returns #f. If (list? foo) is true, then (pair? foo) will be true.

```
(pair? '(a . b))  => #t
(pair? '(a b c))  => #t
(pair? '())       => #f
(pair? '#(a b))   => #f
```

(**point?** *object*) ⇒ *boolean*

Answers #t if the argument is a point, #f otherwise. A point is either a two-element vector of the form #(x y), or a pair of the form (x . y), where x and y are both numbers.

(**port?** *object*) ⇒ *#t or #f*

Returns #t if the given object is either an input port or an output port. Otherwise returns #f. See open-input-file, open-output-file and terminal.

(**positive?** *object*) ⇒ *#t or #f*

This procedure returns #t if its argument is greater than zero. The result for inexact numbers may be unreliable because of small inaccuracies.

(**procedure?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a procedure, otherwise returns #f.

```
(procedure? car)=> #t
(procedure? 'car)=> #f
(procedure? (lambda (x) (* x x)))=> #t
(procedure? '(lambda (x) (* x x)))=> #f
(call-with-current-continuation procedure?)=> #t
```

(**rational?** *object*) ⇒ *#t or #f*

Rational? can be applied to any kind of argument, including non-numbers. It returns #t if the object is a rational, and otherwise it returns #f.

If rational? is true of a number then all higher type predicates are also true of that number. Consequently, if rational? is false of a number, then all lower type predicates are also false of that number.

```
(rational? (/ 6 10))=> #t
(rational? (/ 6 3))=> #t
```

The behavior of rational? on inexact numbers is unreliable, since any inaccuracy may affect the result.

See also number?, complex?, real? and integer?.

(**real?** *object*) ⇒ *#t or #f*

Real? can be applied to any kind of argument, including non-numbers. It returns #t if the object is a real number, and otherwise it returns #f.

If real? is true of a number then all higher type predicates are also true of that number. Consequently, if real? is false of a number, then all lower type predicates are also false of that number.

If z is an inexact complex number, then (real? z) is true if and only if (zero? (imag-part z)) is true. If x is an inexact real number, then (integer? x) is true if and only if (= x (round x)).

```
(real? 3)          => #t
(real? -2.5+0.0i)=> #t
(real? #e1e10)  => #t
```

See also number?, complex?, rational? and integer?.

(**rectangle?** *object*) ⇒ *boolean*

Answers #t if the argument is a rectangle, #f otherwise. A rectangle is either a two-element vector whose first element is the upper-left point and second element is the lower-right point (i.e., #(#(<upper-left-x> <upper-left-y>) #(<lower-right-x> <lower-right-y>))), or a pair whose car is the upper left point and whose cdr is the lower right point (i.e., ((<upper-left-x> . <upper-left-y>) . (<lower-right-x> <lower-right-y>))). Alter uses the convention that the x coordinates increase rightward, whereas y coordinates increase downward (as do screen coordinates).

(**string-ci<=?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String-ci<=? is the lexicographic ordering on strings induced by the ordering char-ci<=? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

(**string-ci<?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String-ci<? is the lexicographic ordering on strings induced by the ordering char-ci<? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

(**string-ci=?** *string1 string2*) ⇒ *#t or #f*

Returns #t if the two strings are the same length and contain the same characters in the same positions, otherwise returns #f. String-ci=? treats upper and lower case letters as though they were the same character. See string=?.

```
(string-ci=? "abc" (string #\a #\b #\c))=> #t
(string-ci=? "ABC" "abc")=> #t
(string-ci=? "ab" "AbC")=> #f
```

(**string-ci>=?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String-ci>=? is the lexicographic ordering on strings induced by the ordering char-ci>=? on characters. If two strings differ in length but are the same up to the length of the shorter string, the longer string is considered to be lexicographically greater than the longer string.

(**string-ci>?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String-ci>? is the lexicographic ordering on strings induced by the ordering char-ci>? on characters. If two strings differ in length but are the same up to the length of the shorter string, the longer string is considered to be lexicographically greater than the longer string.

(**string-empty?** *string1*) ⇒ *#t or #f*

Returns #t if the string is the empty string.

(string-empty? "abc")=> #f

(string-empty? "")=> #t

(**string<=?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String<=? is the lexicographic ordering on strings induced by the ordering char<=? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

(**string<?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String<? is the lexicographic ordering on strings induced by the ordering char<? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

(**string=?** *string1 string2*) ⇒ *#t or #f*

Returns #t if the two strings are the same length and contain the same characters in the same positions, otherwise returns #f. String=? treats upper and lower case as distinct characters. See also string-ci=?.

(string=? "abc" (string #\a #\b #\c))=> #t

(string=? "ABC" "abc")=> #f

(**string>=?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String>=? is the lexicographic ordering on strings induced by the ordering char>=? on characters. If two strings differ in length but are the same up to the length of the shorter string, the longer string is considered to be lexicographically greater than the longer string.

(**string>?** *string1 string2*) ⇒ *#t or #f*

This procedure is a lexicographic extension to strings of the corresponding ordering on characters. String>? is the lexicographic ordering on strings induced by the ordering char>? on characters. If two strings differ in length but are the same up to the length of the shorter string, the longer string is considered to be lexicographically greater than the longer string.

(**string?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a string, otherwise returns #f.

(**subtype?** *object type*) ⇒ *#t or #f*

Returns #t if object is type or a subtype of type. In Alter types are subtypes of themselves.

```
(subtype? type type)=> #t
(subtype? object-type type)=> #f
(subtype? type object-type)=> #t
(subtype operation object-type)=> #t
```

(**symbol?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a symbol, otherwise returns #f.

```
(symbol? ’foo) => #t
(symbol? (car ’(a b)))=> #t
(symbol? "bar")=> #f
(symbol? ’nil)  => #t
(symbol? ’())   => #f
(symbol? #f)    => #f
```

(**thunk?** *object*) ⇒ *#t or #f*

Returns #t if the object is a procedure or operation, otherwise it returns #f.

```
(thunk? car)    => #t
(thunk? length)=> #t
(procedure? length)=> #f
(operation? length)=> #t
```

(**type?** *object*) ⇒ *#t or #f*

Returns true if the argument is a type.

```
(type? type)    => #t
(type? object-type)=> #t
(type? (make object-type))=> #f
(type? type?)   => #f
(type? operation)=> #t
(type? (make type ’() ’(object-type)))=> #t
```

(**vector?** *object*) ⇒ *#t or #f*

Returns #t if sexpr is a vector, otherwise returns #f.

(**writable?** *filename-type*) ⇒ *#t or #f*

Returns #t if the filename represents a file than can be opened for writing. Otherwise returns #f. The file must exist, or Alter will raise an error.

(**zero?** *object*) $\Rightarrow$ *#t or #f*

This procedure returns #t if its argument is exactly equal to zero. The result for inexact numbers may be unreliable because of small inaccuracies.

Types 39

(**all-ivars** *type*) ⇒ *list*

Returns a list of symbols that represent the instance variables for the argument and all of its supertypes. Equivalent to appending the results of calling ivars on the result of (cons type (all-supertypes type)).

(**all-subtypes** *type*) ⇒ *list*

Returns a list containing the argument, each of the argument's immediate subtypes and the elements contained in the result of applying all-subtypes to each of the argument's immediate subtypes with duplicates removed.

(**all-supertypes** *type*) ⇒ *list*

Returns a list containing each of the argument's immediate supertypes and the elements contained in the result of applying all-supertypes to each of the argument's immediate supertypes with duplicates removed.

(**get-type** *object*) ⇒ *type*

Returns the type of the argument.

```
(get-type object-type)=> type
(get-type (make object-type))=> object-type
(get-type (make type '() '(object-type)))=> type
```

(**initialize** *type*) ⇒ *type*

(**initialize** *object*) ⇒ *object*

Performs any necessary initialization on the object and returns the initialized object. Initialize is called automatically by the operation make.

(**ivars** *type*) ⇒ *list*

Returns a list of symbols that represent the instance variables for the argument.

(**subtypes** *type*) ⇒ *list*

Returns a list containing the argument and each of its immediate subtypes.

(**supertypes** *type*) ⇒ *list*

Returns a list containing the argument's immediate supertypes.

User Requests 40

(**confirm** *message [initialanswer [yesstring [nostring]]]*) ⇒ *#t or #f*

Pops up a dialog window with two buttons. The window is labeled with the given string. The second and remaining arguments are optional. If the second argument is given, it must be a boolean indicating which button is to be the default (the one considered "pressed" if the user hits the Return key on the keyboard). The third argument, if present, is a string that is used as the label for the true-valued button. The fourth argument, if present, is a string that is used as the label for the false-valued button. Confirm returns a boolean value corresponding to the button that was chosen by the user.

(**request-directory-name** *[filename]*) ⇒ *filename*

Pops up a dialog window prompting the user to designate an existing directory to use in a subsequent activity. Returns a filename, unless the user cancels the operation, in which case it returns nil.

(**request-file-name** *[filename]*) ⇒ *filename*

Pops up a dialog window prompting the user to designate an existing file to use in a subsequent activity. Returns a filename, unless the user cancels the operation, in which case it returns nil.

(**request-new-file-name** *[filename]*) ⇒ *filename*

Pops up a dialog window prompting the user to designate a new file to use in a subsequent activity. Returns a filename, unless the user cancels the operation, in which case it returns nil.

(**user-choose** *message labels values default [equalize]*) ⇒ *symbol*

Pops up a dialog window prompting the user to choose one element from a list, or to cancel the decision. The first argument is the string to serve as the message to prompt the user. The second argument is a list of strings used to label the various buttons that will be created. The third argument is a list of symbols, one for each button. One of these symbols will be returned, indicating which button was pressed. The fourth argument is a symbol that must match one of the symbols in the list given in the third argument, and specifies which button will be the default (the one considered "pressed" if the user hits the Return key). The fifth (optional) argument is a boolean value specifying whether or not to make the buttons all the same width, or to vary their widths depending on the string labels (default is to equalize the widths).

(**user-choose-from-list** *message labels values default [cancel [max-lines [buttons button-values]]]*) $\Rightarrow$ *object*

Pops up a dialog window prompting the user to choose one element from a list, or to cancel the decision. The first argument is the string to serve as the message to prompt the user. The second argument is a list of strings used to label the items in the list. The third argument is a list of objects, one for each list item. One of these objects will be returned, indicating which item was selected. The fourth argument is an object that must match one of the objects in the list given in the third argument, and specifies which list item will be the default (the one initially selected when the dialog opens). The fifth argument is an object that is returned if the cancel button is pressed. The sixth (optional) argument is an integer value specifying the number of lines in the list to show at once. The seventh (optional) argument is another list of strings which serves as the labels for a set of additional buttons. The eighth (optional) argument must be included if the seventh argument is included. This argument is a list objects, one for each string in the seventh argument, that are returned when one of the additional buttons are pressed.

(**warn** *message [text]*) $\Rightarrow$ *nil*

Pops up a modal (dialog) window with the given message and a single button (labeled "OK") that the user must press (or, equivalently, hit the Return key) in order to continue with further DOME activity.

Vectors 41

(**make-vector** *size [fillval]*) ⇒ *vector*

Returns a newly allocated vector of size elements. If a second argument is given, then each element is initialized to fillval. Otherwise the initial contents of each element is unspecified.

(**vector** *items...*) ⇒ *vector*

Returns a newly allocated vector whose elements contain the given arguments. Analogous to list.

```
(vector 'a 'b 'c) => #(a b c)
```

(**vector-fill!** *vector obj*) ⇒ *vector*

Stores arg2 in every element of arg1. The value returned by vector-fill! is unspecified.

(**vector-length** *vector*) ⇒ *integer*

Returns the number of elements in vector.

```
(vector-length #(1 2 3))=> 3
```

(**vector-ref** *vector k*) ⇒ *object*

Vector-ref returns the kth element of vector.

```
(vector-ref '#(1 1 2 3 5 8 13 21)  5)=> 8
(vector-ref '#(1 1 2 3 5 8 13 21)
  (inexact->exact (round (* 2 (acos -1)))))
=> 13
```

(**vector-set!** *vector k obj*) ⇒ *object*

Vector-set! stores obj in element k of vector. The value returned by vector-set! is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)              => #(0 ("Sue" "Sue") "Anna")
(vector-set! '#(0 1 2) 1 "doe")=> error  ; constant vec-
tor
```

Index