t

The special symbol t is a constant whose value is itself and represents the value true. However anything non- nil is considered to be true. Generally functions that return values that are to be interpreted as true or false return t for true.

nil

The special symbol nil is a constant whose value is itself. It represents two things -

* The value false. Anything non-nil is considered to be true by functions that accept true or false values as input.
* The empty list.

Special Symbols

nil
t

pi

pi has the value 3.14159265358979..., i.e. the length of the circumference of a circle with diameter 1.

For the purpose of executing trigonometric functions sin cos arctan , all angles are measured in radians, where pi radians = 180 degrees.

Constants

pi

Lists

A list is an ordered sequence of zero or more items.

Written representation

A list is written as follows - round left bracket, the written representations of the items in order separated by whitespace followed by a round right bracket. Whitespace between the brackets and the first and last items is optional.

Internal representation

A non-empty list is represented internally by the cons of its first element and the list of the rest of the elements. An empty list is represented by the symbol nil.

Functions that construct lists

cons list

Functions that act on lists

car cdr append mapcar dolist

Examples:

(a b c)

(2)

()

Characters

A character is a single ASCII character. It is represented by #\ followed by itself. Characters are the constituents of strings.

Functions relating to characters

char-code code-char char

Keywords

A keyword is a special sort of symbol.

They are specified by a colon ":" appearing before their print-name.

Unlike other symbols they are self-evaluating, i.e. they are not bound to a variable.

They are used among other things for specifying keyword arguments in function argument lists.

Symbols

A symbol is a name for something. It is represented by a string containing only symbol constituent characters, provided that they don't represent a valid number. This string is called the print name of the symbol.

Symbols have various properties. They have a function value, which is normally set by defun and is retrieved when a list with that symbol as first element is evaluated. They have a property list which is set and accessed using the function get.

A symbol has a package which may be one of three in Apteryx Lisp - the "current" package, the keyword package or no package.

In any given package there is only one symbol with a given name.

Symbols in the current package are read in and written out using just their names. Keyword symbols are specified by the prefix ":" and symbols with no package have the prefix "#:". Symbols with a package are called interned, symbols without a package are uninterned. Uninterned symbols aren't uniquely named because they don't belong to a package.

Functions that construct symbols

gensym make-symbol

Examples:

```
fred
:tom
+
```

Strings

A string is a sequence of characters. Its input and output representation is the sequence of characters enclosed in two double quotes. The character backslash is used as a quoting character i.e. a backslash followed by a character represents that character. So \" represents a double quote in a string and \\ represents a backslash. The maximum length of a string is 65516 characters.

Functions that construct strings

make-string

Functions that act on strings

char length

Examples:

"Hello world"

""

Conses

A cons is a pair of objects. The most common use of conses is in forming lists. A cons can be explicitly constructed using the function cons. Its components can be accessed using car and cdr.

Example:

```
(3 . 4)
```

Integers

Integers are 32-bit signed integers i.e. any number from -2^32 to 2^32-1. Their output format is normal decimal format. Input format can be in hex binary or octal using prefixes #x, #b and #o respectively. For example 26 = #b11010 = #o32 = #x1a.

Functions relating to integers

+ - * / rem 1+ 1-

Examples:

```
2
9999
-34
```

Floating point numbers

Floating point numbers are approximate representations of real numbers. They are represented internally by a mantissa and exponent. Real numbers are accurate to about 11 decimal places with a decimal exponent in the range -38 to 38.

The written representation is a decimal number consisting of an optional sign followed by optional digits followed by a decimal point followed by optionally more digits followed optionally by an exponent which consists of the letter e followed by a integer. There must be at least one digit before or after the decimal point (a single dot is part of the syntax of dotted lists).

If arithmetic functions take arguments that are a mixture of floating point numbers and integers, the results are always floating point.

The function float converts a number to a floating point number.

Examples:

3.45

2.3e15

-9.876e-1

Structures

Structures are objects belonging to a type defined by the user using defstruct. A structure consists of a finite number of named fields, each field having a value which is a lisp object.

Structures are created, accessed and changed using the functions defined by defstruct.

Example:

```
#S(person name "Fred Jones" age 32 hair-colour red)
```

Arrays

An array is a sequence of N lisp objects, each object being accessable by an index in the range 0 to N-1.

The written representation is the same as for a list, but preceded by a hash character.

Functions that construct arrays

vector make-array

Example:

```
#(a b c d)
```

Builtin functions

A builtin function is one that is predefined by Apteryx Lisp.

Example:

```
<Builtin: cons>
```

Special forms

A special form is the function value of one of let, let*, case, while, function etc. They are not functions because they do not evaluate all their arguments before being evaluated, and they are not defined as macros.

Example:

```
<Special form: while>
```

Builtin macros

A builtin macro is one that is predefined by Apteryx Lisp.

Example:

```
<Macro: <Builtin: setf>>
```

Closures

A closure is the result of applying function to a lambda expression, to produce an object that represents a function. defun also produces a closure.

A closure consists of the function argument list, the expressions that make up the body of the function and any lexical variable bindings in force at the time that the closure was created.

Example:

```
<closure: (x) : #E((* x x))>
```

Macro

A macro represents a user-defined macro defined using defmacro. It is effectively a closure that is applied to the unevaluated arguments of the macro call to produce the result that is evaluated in the lexical scope of the macro call.

Windows constants

A windows constant is an object that represents an integer that is given a particular meaning by Microsoft Windows (TM) in certain circumstances.

The windows constant contains both the integer and the nature of the circumstance in which it is to be used, generally as a certain argument to a certain function or group of functions.

The windows constants are the global variable values of symbols whose names are those of the constants (as given in the Microsoft Windows (TM) API).

Example:

```
#<constant: show_window sw_maximize>
```

Windows

A window represents a program-created window on-screen which contains associated data and a re-paint procedure. Create a window using the function make-window.

Colours

A colour represents a colour in RGB format, i.e. a number from 0 to 255 to describe the intensity of each of red, green and blue.

Colours are created using the function rgb.

Example:

```
#<Color: 0 0 255>
```

Points

A point represents a point on the screen or in a window, with x and y coordinates in pixels. Points are created using the function point and their components are accessed using the functions point-x and point-y. The x and y coordinates must be in the range -32768 to +32767, i.e. signed 16 bit integers.

Example:

```
#<Point: 20 40>
```

Extents

An extent represents the size of a rectangle on the screen or in a window, i.e. the width and height in pixels. Extents are created using the function extent and their components are accessed using the functions extent-x and extent-y. The x and y components must be in the range -32768 to +32767, i.e. signed 16 bit integers.

Example:

```
#<Extent: 100 200>
```

Rectangles

A rectangle represents the size and position of a rectangle on the screen or in a window. A rectangle is created using the function rect.

Example:

```
#<Rect: top-left:(0, 0), bottom-right:(100, 200)>
```

Brushes

A brush is a drawing tool used to color things in - it has colour and hatch type. Brushes are created using the functions create-hatch-brush and create-solid-brush.

Pens

A pen is a drawing tool used to draw lines or curves - it has colour and thickness. Pens are created by the function create-pen.

Fonts

A font is a graphic representation of the ANSI character set used to draw letters. Fonts are created by the function create-font.

Streams

A stream represents a file opened for input or output. Create a stream using the function open.

Whitespace

Whitespace refers to the following characters - space, newline, carriage return, page break. Whitespace is used to separate items in written representations of lisp objects.

Symbol constituents

Characters that can be used to make up a symbol name. They are - letters a-z, A-Z, digits 0-9 and the characters +-^_*&@.[]%/<>=

Types

Syntax

Palettes

A device context has associated with it a palette of that device context. If a colour that is not on the palette is used for drawing graphics on the device context then either the nearest available colour or a mixture of available colours will be used to approximate the requested colour.

Device context

A device context is something that graphics commands operate on.Apteryx Lisp uses a concept of a current device context. The control structure with-dc creates a device context for the specified window, and all graphics commands executed inside that control structure will cause graphics to be drawn in that window. When with-dc is finished, then that created device context ceases to exist, and whatever was the previous current device context becomes the current device context again.

There is an initial null device context which simply ignores any graphics commands.

A current device context is also automatically created when part of a window needs to be repainted and its painter function is called.

As well as commands that draw on a device context there are others that create drawing tools such as a font, brush or pen, which can be selected using with-select, and others that change characteristics of the current device context.

All commands that operate on the current device context are listed underGraphics functions.

Garbage collection

When any sort of data is created in lisp, memory must be allocated to store that data. When that data is no longer accessible by any means, that memory can be recycled. Periodically, a garbage collection process occurs where lost memory is reclaimed.Apteryx Lisp uses a copy and compact scheme to do garbage collection. When garbage collection occurs an icon of a garbage bag with the letter G on it is displayed.

Functions

A function is something that takes some number of arguments as input values and returns a value. This is a standard mathematical concept independent of any particular computer programming language.

In Lisp a function may also have some side effects, i.e. cause something to occur in addition to just returning a value. In cases where a function was applied primarily for its side-effects and the value returned was unimportant, such a function would be called a procedure in other programming languages.

Functions arise from two sources - they are provided by Apteryx Lisp or they are created using a lambda expression.

They can be specified in basically two ways - by referring to them by name, or by taking the function value of a lambda expression. Functions that are referred to by name are either built-in functions or they have been defined using defun.

Lambda expressions

A lambda expression is a description of a function in terms of its argument list and its body. For example -

(lambda (x y) (print x) (+ x y))

The keyword lambda indicates that this is a lambda expression. The (x y) is the argument list, in this case two positional arguments called x and y, and the other two elements (print x) and (+ x y) are the body of the lambda expression, which are evaluated when a function derived from the lambda expression is applied to some arguments.

Use the function special form to create a closure of the lambda expression, which may then be passed as an argument to functions such as apply funcall mapcar etc.

Argument lists

The argument list is a specification of what arguments may be passed to a function or macro and how they will be interpreted when passed.

Such a list is given in a defun or defmacro after the name of the function or macro, and also when specifying a lambda expression.

An argument list is given as a list, with the elements specifying the arguments in the list.

First come the positional arguments which are specified by a symbol which is the name of the local variable whose value will be the value of the positional argument passed when the function (or macro) being defined is called. When the function is applied to a list of arguments, if the argument list has n positional arguments then there must be at least n arguments given and these will be bound in order to the variables named as positional arguments in the argument list.

For example - (defun my-fun (x y z) (* x y z)), the arguments x, y and z are all positional arguments. When calling my-fun, e.g (my-fun 3 4 5) the local variable x will bound to 3, y to 4 and z to 5, so that the value returned will be (* 3 4 5) = 60.

Other types of arguments in an argument list are specified by preceding them with special argument list keywords. These keywords are not actually keyword symbols as such, but symbols with names starting with "&".

Optional arguments - these come after the keyword &optional, and are specified as a symbol being the variable name, or a two or three element list containing the variable name, default value expression and (optionally) the "suppliedp" variable name.

When the function is passed arguments, any after the positional arguments are first treated as optional arguments in the order given. If no value is given for an optional argument then the default value expression is used to calculate its value, and if there is no default value expression then nil is the value. The suppliedp variable, if given in the argument list, is bound to t or nil, according to whether an actual value was given to the optional argument.

For example - (defun my-fun (x &optional y (z (+ 5 6)) (q nil q-supplied))), the argument x is positional and y, z and q are optional arguments. If the function is called with (my-fun 3 4), then x will have the value 3, y the value 4, z the value 11, q the value nil and q-supplied the value nil.

Keyword arguments are given after the &key argument list keyword. They are similiar to optional arguments in that they may be given or not, and they can have a default value expression and suppliedp variable. However their presence or absence is specified by preceding their value with a keyword. When specifying the variable name, either a name can be given, in which case the keyword is the keyword symbol with the same print name, or a list of 2 elements can be given, consisting of the keyword and its variable name. If the keyword and variable name are given as a two element list, then this must be a 1, 2 or 3 element list (containing optional default value expression and suppliedp variable) to avoid ambiguity.

For example - (defun my-fun (x &key y (z (+ 5 6)) ((:q queue) nil q-supplied))) Here x is positional, y, z and queue are keyword arguments. A function call might be (my-fun 39 :q 'jim :z 13), in which case the values supplied would be x = 39, queue = jim, z = 13 and y = nil. Note that keyword arguments can be given in any order.

A single argument may be specified after a &rest keyword. This is the variable that will be

bound to the list of all arguments passed after the positional and optional arguments. If both &key and &rest are used, all the arguments after the positional and optional arguments have to be in the form of keyword-value pair. If the the &allow-other-key argument list keyword is given, then keywords other than those specified may be passed as arguments.

For example - (defun my-list (&rest x) x). When called as (my-list 3 4 (+ 2 5)) the result returned will be (3 4 7).

Variables

A variable is a place that holds a value, and is usually associated with a symbol which is the name of that variable. The variable is said to be bound to the symbol. There are two types of variables - special or dynamic or global variables and lexical or static or local variables.

Lexical variables are ones that are bound by some syntactical construct and may only be referred to within that construct. Examples of constructs that create lexical variables are let let* do do* defun defmacro , and also lambda expressions.

The evaluation rule for symbols is that they evaluate to the innermost lexical variable bound to that symbol, or if there is none, to the special variable for that symbol. When one variable binding takes precedence over a second, the first is said to shadow the second. The same shadowing rules apply to variable references in setq and setf.

However, the functions set and symbol-value always refer to the special variable for a symbol.

Lexical variables are always given a value when they are created, but an attempt to reference a global variable that has not previously had a value assigned to it using setq or setf will result in a undefined variable error.

A new lexical variable is created every time that the construct that defines it is executed. The binding of the variable to its name finishes when the construct is executed, but the variable itself may live on. This is mainly relevant when a closure is created using function on a lambda expression within the construct, the lambda expression evaluates a lexical variable defined by the construct, and the resulting closure is passed out of the construct.

For example - (defun make-adder (x) #'(lambda (y) (+ x y))). The lexical variable in question is x. The result of say (make-adder 3) will be a function which accepts a single argument and adds 3 to it.

Lexical scope

Lexical scope is when the interpretation of a name depends on where it is lexically, i.e. what syntactical construct it occurs in. An alternative is dynamic scope, where the interpretation of a name changes with time.

Function value

The function value is the interpretation of either a symbol or a lambda expression as a function. One of the properties of a symbol is its function value, this may be set using defun, defmacro or using setf and symbol-function. If the function value is a macro (defined by defmacro or predefined), or a special form then it is not therefore a function as such.

The function value of a lambda expression is the "closure" formed by binding lexical variable references in the lambda expression to the variables that are bound to those references at the time the closure is made (using function).

Special form

The function value of a symbol that has its own special rules for evaluation (see evaluation).

Argument

A value that is passed as input to a function. A function is said to be applied to its arguments. (see evaluation).

Evaluation

Two things happen when a Lisp program is executed - evaluation and application.

When Lisp:Eval is called, the selected expression is evaluated. This means that the rules of evaluation are used to decide how to evaluate the expression and return a result.

The rule of application depends on the type of the data described by the expression. Most data types in lisp are self-evaluating, this means that the result of evaluating the expression is to return the same expression as the result, and nothing else happens.

The two data types that aren't self-evaluating are lists and non-keyword symbols.

A non-keyword symbol is evaluated by determining which variable it is bound to, and returning the value of the variable.

If the symbol is in the lexical scope of a local variable declaration, as in let let* do do* or as an argument in a function definition, then the variable in the innermost scope that the symbol is in is the one whose value is returned.

If no local variable is associated with the symbol, then the global variable associated with that symbol is the one used. The global variable may not have been set to a value, in which case an error occurs.

A symbol may be defined as a constant using defconstant, in which case the value of that constant will be returned, and in fact such a symbol may not be bound to a local variable. t and nil are predefined to be constants with values equal to themselves (so they are effectively self-evaluating).

How a non-empty list is evaluated depends on the first member of the list. First the function value of the first member is determined. This will be either a function, macro or special form.

Generally this first member will be a symbol, and its function value is a function if it has been defined using defun, a macro if it has been defined using defmacro. There are also predefined functions, macro's and special forms.

The members of the list after the first member (and there can be any number of them from zero upwards) are called the arguments.

If the function value of the first member is a function, then the arguments are themselves first evaluated, and then the function value of the first member is applied to the evaluated arguments.

If the function value of the first member is a macro, then the function associated with that macro is applied to the unevaluated arguments of the list, and the result is evaluated in the original lexical scope of the list.

If the function value of the first member is a special form, then that form has its own particular rules for evaluation which can be looked up in the help for that special form.

Special forms include case cond catch dotimes dolist while progn prog1 prog2 if when unless let let* unwind-protect defsetf defconstant setq psetq quote backquote and or defstruct with-struct function defun defmacro with-continuous-gc with-dc with-select .

These are all the forms that are not defined as functions or macros in Apteryx Lisp. However in official Common Lisp terminology a special form is one of a special set of basic forms from which all other non-function forms could in principle be defined as macros.

Application

A function is said to be applied to its arguments. Most application occurs when a list whose first member's function value is a function isevaluated.

Errors

An error occurs whenever you ask Apteryx Lisp to do something that doesn't make sense.

There are two basic kinds of error - syntax errors, where it tries to read a lisp expression that is not syntactically correct, and evaluation errors when an error occurs evaluating an expression.

The response to a syntax error in a editor window is to display a message box with a short message about the error, and in most cases to move to the position in the buffer where the error occurs. This position is the position where the lisp parser decides it can no longer make any sense of what it is reading. If a syntax error occurs during evaluation of read or load, then the position of the error will be shown by showing the line, line number and name of file where the error occurred. In such a case, the error will also be an evaluation error.

An evaluation error occurs when either an attempt is made to evaluate an invalid expression, or an attempt to apply a function to its arguments fails. A message box appears with a short message about the error, possibly with a description of the object that caused the error, followed by a stack dump.

A stack dump is a display of all the things that Apteryx Lisp was currently doing when the error occurred. Stack dumps are output to the output window. Each line of a stack dump is one of an evaluation, a compilation or an application. An evaluation is shown simply as the expression being evaluated. An application is shown as the function being applied followed by a list of its arguments enclosed in brackets and separated by commas. A compilation is shown using the word "compiling" followed by the expression being compiled. By perusing the stack dump you will often be able to determine the cause of an error.

When you define your own functions you may wish them to fail under some circumstances. The function error can be used to signal an error.

Compilation

Compilation is a process where Lisp expressions are first converted to a form that can be more efficiently evaluated before they are evaluated. Automatic compilation is turned on by setting the special variable *auto-compile* to a non-nil value (e.g. t) and off again by setting it to nil.

When compilation is on, if the the-compiler function has been setf-ed to a function value, then any expression being evaluated is compiled using that function before being evaluated. If (the-compiler) has the value nil, then expressions are compiled by expanding all the macro calls.

Loading the file "compile.lsc" defines a value for (the-compiler) which compiles expressions into compiled pseudo-code objects, similiar in concept to p-code for Pascal. The compiled code objects should run faster than purely interpreted code, and also be less likely to run out of execution stack. The function compile-and-load is also defined in the file "compile.lsc", and it can be used to compile whole files of Lisp expressions. The combination of compiling and loading is done to make the process completely transparent. For example, functions in a file may be defined to be used in macro definitions which in turn occur in later function definitions. Simply compiling all expressions without evaluating them would lead to undefined function errors in such a case. "compile.lsp" is the source file for "compile.lsc". Be warned that alteration of "compile.lsc" could result in crashing Apteryx Lisp (due to the attempted execution of invalid code objects).

Generally it is appropriate to set *auto-compile* to t before loading defuns, defmacros and other definitions that have been debugged and are going to be used heavily.

For compilation to work properly, any macros used in an expression must be defined before the expression is evaluated. If a defun calls a macro, and you redefine the macro (by evaluating its defmacro) then if *auto-compile* was on when you evaluated the defun you will have to re-evaluate the defun to get the benefit of the redefined macro. Compilation is therefore probably inappropriate for code that is in a state of development.

WARNING: Apteryx Lisp doesn't currently support packages, and since the compiler is written in Lisp, there is a possibility of user code conflicting with the compiler code. To avoid this, all functions and symbols that are local to the compiler start with an underscore character. So avoid using symbol names starting with underscores in your own code.

Important concepts

Application
Argument
Argument lists
Compilation
Device context
Errors
Evaluation
Function value
Functions
Garbage collection
Lambda expressions
Lexical scope
Palettes
Special form
Variables

Editor overview

Apteryx Lisp consists of a multi-document file editor tightly integrated into the lisp system.

There are lisp-aware editor commands for auto-indentation, expression selection and navigation and expression evaluation.

Output from evaluation goes to the special Output Window.

There are some lisp commands for manipulating the editor.

Configuration file

The configuration file is the file that is specified as the parameter for Apteryx Lisp when it is started up. If no parameter is given it defaults to "apteryx.lcf" (in the current directory).

It stores the names of the files that are being edited in a given session and the size of the main window when not maximized.

It also stores the cursor color.

The configuration is actually stored as lisp commands that are executed when Apteryx Lisp starts up.

If problems occur starting up, it may be that the configuration file has been corrupted, in which case you can delete it before trying to restart Apteryx Lisp.

Color Scheme

The color scheme of Apteryx Lisp is largely inherited from the Windows system color scheme, which can be set by the user in the Control Panel.

One minor exception is the color of the text cursor. This defaults to red. It can be changed by using the functions set-cursor-color and rgb. The cursor color gets stored as part of the configuration file, so only needs to be changed once.

File Windows

File windows are created using the File:New or File:Open commands.

Output Window

The output window is the window labeled LispOutput which is always present, and which appears near the bottom of the the Apteryx Lisp window when the editor windows are Window:Cascaded.

Output to the output window comes from printing to *standard-output*, showing results of evaluations, error messages and stack dumps.

Expressions in the output window cannot be evaluated - if you wish to evaluate one, copy and paste it into a file window first.

Auto-indentation

Apteryx Lisp provides commands that allow you to automatically indent lisp expressions. This means that when a lisp expression is multi-line, the number of spaces at the beginning of each line depends on how deeply nested the expression is at that point.

The rule used is fairly simple - indent 2 spaces for each level of nesting.

The commands that auto-indent are Special:Indent Line, Special:New Indented Line and Lisp:Indent. Normally you would use the keyboard shortcuts for these which are tab, Ctrl-Enter and F5 respectively.

Generally, use Ctrl-Enter instead of Return when starting a new line, use tab to correct indentation of the current line (e.g. you forgot to use Ctrl-Enter instead of Return) and use F5 to correct the indentation of a whole top-level expression.

The parser actually requires that lines that are not the beginning of a top-level expression be indented. This requirement prevents the parser going all the way to the bottom of a file when a right bracket has been omitted.

Parsing lisp expressions

Lisp expressions are read in from a file or file window by the parser, which parses them, i.e. recognises the components of an expression.

Parsing occurs when an expression is evaluated, when the functions read and load are called, and when the various lisp-aware editor commands are executed, such as select expression, find expression forward or backwards and the auto-indentation commands.

The parser attempts to read in a correct lisp expression. However if something is read in that cannot be interpreted as a valid lisp expression, then a syntax error will occur. An error will generally be reported by a message box, with further information written to the output window. When the parser is reading a file window the cursor will be sitting in front of the first un-interpretable character.

When the expression being parsed is not to be evaluated, the parser doesn't need to construct the object being parsed and is less fussy about what it accepts, for example it doesn't check the components of a structure expression against the structure definition.

Top-level expressions

A top-level expression is a lisp expression which is not contained in any other expression. In general a file of lisp commands consists of a sequence of top-level expressions.

The various commands that operate on lisp expressions, e.g. Lisp:Eval, Lisp:Select Expr and Lisp:Indent all operate on a top-level expression that either contains or precedes the text cursor in a file window, unless there is a current selection, in which case the command acts on whatever has been selected.

Special:Find Expr Bwd also has to find a top-level expression, even though in general it picks out the sub-expression that precedes the cursor. Bracket flashing also involves first finding the beginning of a top-level expression, so that typing a right bracket can result in a syntax error.

The beginning of a top-level expression is found using the assumption that any line without leading spaces is the beginning of a top-level expression. Then the parser reads forward until it finds a top-level expression that the cursor is in or which is the last top-level expression that comes before the cursor.

Note that this does not require that the top-level expression in question start at the beginning of a line, but if it doesn't, then a syntax error in a previous top-level expression can cause the command to abort with that syntax error.

Bracket flashing

When you type a right bracket in a file window, Apteryx Lisp determines if the bracket is the end of a lisp expression, and if it is, it "flashes" the whole expression thus terminated.

What this means is that it selects the whole expression while the ")" key is held down, and unselects it when it is released. If the beginning of the expression is not visible in the file window, then the window will temporarily scroll to show you the beginning of the selected expression while the bracket key is held down.

It is possible that a syntax error will be detected when trying to find the expression that the bracket terminates. When this happens a message box for the error appears, but the cursor remains after the bracket just typed. To find the position of the error, select Lisp:Select Expr (F2).

Editing

Menu Commands

File
- New
- Open
- Save
- Save As
- Save All
- Printer Setup
- Exit

Edit
- Undo
- Redo
- Cut
- Paste
- Copy
- Duplicate
- Select All
- Show Undos

Search
- Find
- Find Next
- Find and Replace
- Browse
- Browse Headings

Special
- Indent Line
- New Indented Line
- Find Expr Fwd
- Find Expr Bwd
- File Statistics
- Tabs to Spaces
- Save Config

Lisp
- Eval
- Load Buffer
- Load Rest
- Select Expr
- Indent

Window
- Tile
- Cascade
- Tile with Output
- Arrange Icons

Help
- Index
- Help on Help
- About Apteryx Lisp
- About Registration

File:New (menu option)

Create a new file window called (Untitled) which is not associated with a file until you do a File:Save As command.

File:Open (menu option)

Brings up a "File Open" dialog box. To select a file click on a file name in the list box or enter the filename in the filename field. To open the selected file, double click on its name in the list box or hit the OK button or press Enter.

A new file window will be created for the specified file.

However if the file is already opened, a message to that effect will come up and its existing window will be brought to the front.

This means that Apteryx Lisp doesn't allow you to open two editing windows for the same file.

File:Save (menu option)

Save changes made to the file in the current editing window.

This option is greyed if no changes have been made since the last save.

If the current file window is (Untitled) (i.e. has been created using File:New) then a "Save As" dialog box will come up and you must supply the name of the file that you want the window contents saved to.

File:Save As (menu option)

Brings up a "Save As" dialog box to specify a new file name for the file that the current editor window is to be saved to.

File:Save All (menu option)

Save all editor windows that have had changes made since the last save.

File:Printer Setup (menu option)

Setup default printer for all printing operations

File:Exit (menu option)

Exit from Apteryx Lisp. If there are editor windows that have unsaved changes, then you will be prompted as to whether you want to save those changes before exiting, or if you want to cancel the exit command.

Edit:Undo (menu option)

Undo last editing operation on current file window. Undo's can be repeated and mixed with Edit:Redo provided that no other editing operation intervenes.

Edit:Redo (menu option)

Undo previous Edit:Undo operation.

Edit:Cut (menu option)

Delete selected text from current window, and put it into the Windows clipboard.

Edit:Paste (menu option)

Insert contents of Windows clipboard at current cursor position in the current editing window, replacing the currently selected text if any.

Edit:Copy (menu option)

Copy the selected text into the Windows clipboard (without deleting it).

Edit:Duplicate (menu option)

Duplicate the currently selected text by copying it into the Windows clipboard and then copying it into current editing window.

Edit:Select All (menu option)

Select the whole of the current editing window.

Edit:Show Undos (menu option)

Output into the output window the list of past editing operations stored in the undo stack of the current file window.

Search:Find (menu option)

Bring up a dialog which lets you perform an incremental search in the current editing window. The search starts as you type characters into the search string field. You can choose case-sensitive or case-insensitive. Choose OK to move to the search string found, or choose Cancel to return to your original position. Choose Next to find the next copy of the string currently being searched for and choose To Top to go to the top of the file.

Search:Find Next (menu option)

Bring up search dialog searching for the last string searched for using Search:Find.

Search:Find and Replace (menu option)

Bring up a search and replace dialog box. It works similiarly to the plain search dialog brought up by Search:Find but adds a field for entering replacement text, and a Replace and Find Next button. The button replaces any currently found text with the text in the Replacement Text field, and then looks for the next match to the Search Text if any.

Search:Browse (menu option)

Bring up a BROWSER dialog. It shows in a list box all lines in the current file window starting with either "(" or ";;;". Lines starting with ;;; represent headings and the rest represent the beginnings of top-level expressions.

Click on a line in the list box to move to the corresponding line in the file window. Choose OK to return to the file window or Cancel to go back to your original position in the file window. Set the Headings Only check box to see only the lines starting with ;;;.

The general idea is that you should create one line comments starting with ;;; at the beginning of major sections of your source code file. You can then use Browse with and without Headings Only checked to quickly navigate around a source file.

Search:Browse Headings (menu option)

As for Search:Browse but with the Headings Only option set.

Special:Indent Line (menu option)

Indent the current line according to the auto-indentation scheme.

Special:New Indented Line (menu option)

Start a new line which is indented according to the auto-indentation scheme.

Special:Find Expr Fwd (menu option)

Find and select the expression after the current cursor or selection position. Failure to find a syntactically valid expression will result in an error message.

Special:Find Expr Bwd (menu option)

Find and select the expression before the current cursor or selection position. Failure to find a syntactically valid expression will result in an error message. Parsing starts from the top of an assumed top-level expression.

Special:File Statistics (menu option)

Display statistics on size of current file window.

Special:Tabs to Spaces (menu option)

Convert all tab characters (which Apteryx Lisp does not show properly) to equivalent spaces, assuming the usual tab-stops every 8 characters.

Special:Save Config (menu option)

Save the current configuration file, which contains a list of files being edited and the current un-maximised size of the main window.

Lisp:Eval (menu option)

If there is no selection then first find a top-level expression surrounding or before the current cursor position (as in Lisp:Select Expr).

Read in the found or selected expression and evaluate it, showing the results in the output window.

Lisp:Load Buffer (menu option)

Read in and evaluate all top-level expressions in the current file window.

Lisp:Load Rest (menu option)

Read in and evaluate all top-level expressions in the current file window, starting at the current cursor position.

Lisp:Select Expr (menu option)

Find a top-level expression that starts before the current cursor position (and which may finish before or after the current cursor position).

The assumption is made that a line starting with (, #, ' or ` is a top-level expression.

Lisp:Indent (menu option)

If there is a selection indent it according to the auto-indentation scheme, or else find a top-level expression that starts before the current cursor expression and indent that.

Window:Tile (menu option)

Tile the file windows and the output window.

Window:Cascade (menu option)

Cascade the file windows with the output window at the bottom.

Window:Tile with Output (menu option)

Put all file windows in top 70% of screen and LispOutput window in bottom 30% of screen, with current file window (if any) on top.

Window:Arrange Icons (menu option)

Arrange icons in a neat row.

Help:Index (menu option)

Get help on Apteryx Lisp.

Help:Help on Help (menu option)

Get help on Microsoft Windows (TM) help.

Help:About Apteryx Lisp (menu option)

Show credits and copyright notice for Apteryx Lisp.

Help:About Registration (menu option)

Bring up a read only window with information about registering as a user of Apteryx Lisp.

do (&rest VARIABLE-DECLARATION) (TEST &optional RESULT) &rest EXPRS

A general looping macro.

Local variables are created from the VARIABLE-DECLARATIONS just as in let and the scope of the resultant variable bindings is the rest of the do expression. However each variable declaration can include a third element (in addition to the variable name and initial value) which is the update value for the variable and is assigned to that variable each time around the loop. The update expressions are calculated in parallel so that any references to other variables in an update expression always refer to the old value of those variables.

TEST-RESULT consists of a list containing a test expression and an optional result expression (whose default value if not supplied is nil). The test expression is evaluated at the start of each loop (after variable initialisation or update) and if the test evaluates to nil i.e. false, then the loop terminates otherwise execution continues.

Each time the TEST expression is non-nil i.e. true, the EXPRs are evaluated in order, and then the update expressions (if any) are evaluated and assigned to their respective variables.

When the TEST expression evaluates to nil and the loop terminates, the RESULT expression is evaluated to give the result to be returned from do.

do* **(&rest VARIABLE-DECLARATION) (TEST &optional RESULT) &rest EXPRS**

A general looping macro.

Local variables are created from the VARIABLE-DECLARATIONS just as in let* and the scope of the resultant variable bindings is the rest of the do* expression. However each variable declaration can include a third element (in addition to the variable name and initial value) which is the update value for the variable and is assigned to that variable each time around the loop. The update expressions are calculated sequentially so that any references to other variables in an update expression refer to the new values of preceding variables and the old values of following variables.

TEST-RESULT consists of a list containing a test expression and an optional result expression (whose default value if not supplied is nil). The test expression is evaluated at the start of each loop (after variable initialisation or update) and if the test evaluates to nil i.e. false, then the loop terminates otherwise execution continues.

Each time the TEST expression is non-nil i.e. true, the EXPRs are evaluated in order, and then the update expressions (if any) are evaluated and assigned to their respective variables.

When the TEST expression evaluates to nil and the loop terminates, the RESULT expression is evaluated to give the result to be returned from do*.

Looping macros

do
do*

constant-value **CONSTANT**

Returns the integer value of the windows constant CONSTANT.

Example:

```
(constant-value sw_maximize)
 => 3
```

show_window constants

How to display a new window

sw_Hide	Hide the window
sw_Maximize	Show the window full screen
sw_Minimize	Show the window as an icon
sw_Normal	Show as an active normal window

message_box_default constants

Which button in a message box is the default button

mb_DefButton1 First button is the default
mb_DefButton2 Second button is the default
mb_DefButton3 Third button is the default

message_box_icon constants

Which icon to show in a message box

mb_IconExclamation	An exclamation mark
mb_IconInformation	An i for information
mb_IconQuestion	A question mark
mb_IconHand	A hand signalling stop
mb_IconStop	A hand signalling stop

message_box_mode constants

Message box mode

mb_ApplModal	Stop parent window until message box is answered
mb_SystemModal	Stop Windows until message box is answered
mb_TaskModal	Stop program until message box is answered

message_box_type constants

Type of message box

mb_AbortRetryIgnore	Has Abort (default), Retry and Ignore buttons
mb_Ok	Has OK button
mb_OkCancel	Has OK (default) and Cancel buttons
mb_RetryCancel	Has Retry (default) and Cancel buttons
mb_YesNo	Has Yes (default) and No buttons
mb_YesNoCancel	Has Yes (default), No and Cancel buttons

command_id constants

Which message box button the user chose

id_Abort	The Abort button
id_Cancel	The Cancel button
id_Ignore	The Ignore button
id_No	The No button
id_Ok	The OK button
id_Retry	The Retry button
id_Yes	The Yes button

hatch_style constants

Hatch style (for a brush)

hs_BDiagonal	Backwards leaning diagonal lines
hs_Cross	Horizontal and vertical lines
hs_DiagCross	Diagonal crossing lines
hs_FDiagonal	Forwards leaning diagonal lines
hs_Horizontal	Horizontal lines
hs_Vertical	Vertical lines

stock_brush constants

Stock (i.e. pre-supplied) brushes

Black_Brush	Black brush
DkGray_Brush	Dark gray brush
Gray_Brush	Gray brush
Null_Brush	Brush that paints nothing
LtGray_Brush	Light gray brush
White_Brush	White brush

stock_pen constants

Stock (i.e. pre-supplied) pens

Black_Pen	Black pen (one pixel thick)
Null_Pen	Pen that draws nothing
White_Pen	White pen (one pixel thick)

stock_font constants

Stock (i.e. pre-supplied) fonts

ANSI_Fixed_Font ANSI fixed width font
ANSI_Var_Font ANSI variable width font
Device_Default_Font Default font for graphics device
System_Fixed_Font Old system font
System_Font Windows 3 system font

stock_palette constants

Stock (i.e. pre-supplied) palettes

Default_Palette Default colour palette

system_color constants

Colours used by Windows itself in various situations

color_ActiveBorder Colour of border of active window
color_ActiveCaption Colour of caption of active window
color_AppWorkSpace Colour of application work space
color_Background Colour of background
color_BtnFace Colour of button face
color_BtnShadow Colour of button shadow
color_BtnText Colour of button text
color_CaptionText Colour of window caption text
color_GrayText Colour of grayed text
color_Highlight Colour of highlighted text
color_HighlightText Colour of highlighted text
color_InactiveBorder Colour of border of inactive window
color_InactiveCaption Colour of caption of inactive window
color_Menu Colour of menu
color_MenuText Colour of menu text
color_Scrollbar Colour of scroll bar
color_Window Colour of window
color_WindowFrame Colour of window frame
color_WindowText Colour of window text

pen_style constants

Pen style

ps_Solid	Solid line
ps_Dash	Dashed line
ps_Dot	Dotted line
ps_DashDot	Dash and dot line
ps_DashDotDot	Dash and dot dot line
ps_Null	No line
ps_InsideFrame	Pen used to draw inside of a frame

binary_raster_op constants

Binary raster operation type for adding new color to existing when drawing painting etc.

r2_Black	black
r2_CopyPen	new color
r2_MaskNotPen	existing color ANDed with NOT of new
r2_MaskPen	existing color ANDed with new
r2_MaskPenNot	not of existing color ORed with new
r2_MergeNotPen	existing color ORed with NOT of new
r2_MergePen	existing color ORed with new
r2_MergePenNot	existing color ORed with NOT of new
r2_Nop	existing color (i.e. draw nothing)
r2_Not	NOT of existing color
r2_NotCopyPen	NOT of new color
r2_NotMaskPen	NOT of existing color ANDed with new
r2_NotMergePen	NOT of existing color ORed with new
r2_NotXorPen	NOT of existing color XORed with new
r2_White	white
r2_XorPen	existing color XORed with new (this is self-erasing)

Windows constants - values

Windows constant functions

constant-value

data-size OBJECT

Return size of data associated with object.

Examples:

```
(data-size 34)
 => 4
(data-size '(jim fred))
 => 8
(data-size #(0 1 2 3 4 5 6 7 8 9))
 => 44
```

type-of **OBJECT**

Returns a symbol describing the type of OBJECT

Examples:

```
(type-of 2)
 => integer
(type-of 'fred)
 => symbol
```

Type functions

data-size
type-of

set-trace **FLAG**

Use (set-trace t) to set tracing on, and (set-trace nil) to turn it off.

If tracing is on, then every expression being evaluated and its result is printed to the output window.

baktrace

Prints out a dump of current state of stack to output window.

Debugging

baktrace
set-trace

length **SEQUENCE**

Returns as an integer the length of an array, string or list.

Examples:

```
(length '(a b c))
 => 3
(length "Hello")
 => 5
(length #(a b))
 => 2
```

Sequence functions

length

version

Returns string giving version number of Apteryx Lisp.

Example:

```
(version)
 => "1.04"
```

message-box **MESSAGE :caption CAPTION :type TYPE :default DEFAULT :icon ICON**

Puts up a task modal message box showing message.

Default style has just an OK button.

Task modal means that nothing else can be done to Apteryx Lisp until the message box is replied to.

CAPTION is the caption that is displayed on the top of the window.

TYPE is one of the message box type windows constant.s mb_AbortRetry, mb_Ignore, mb_Ok (default), mb_OkCancel, mb_RetryCancel, mb_YesNo, mb_YesNoCancel. It determines which buttons appear on the message box for the user to select.

DEFAULT is one of the message box default windows constant.s mb_DefButton1 (default), mb_DefButton2, mb_DefButton3. It determines which button on the message box is the default button.

ICON is one of the message box icon windows constant.s mb_IconAsterisk, mb_IconExclamation, mb_IconHand, mb_IconInformation, mb_IconQuestion, mb_IconStop. It determines which icon if any appears on the message box.

Returns one of the symbols id_abort, id_cancel, id_ignore, id_no, id_ok, id_retry or id_yes depending on which button was clicked.

pascalize SYMBOL

Convert a symbol into one that only contains alphabetic letters, digits, underscores and which doesn't start with a digit, while retaining uniqueness.

Examples:

```
(pascalize 'myfunction)
 => myfunction
(pascalize '1+)
 => __1__plus_
(pascalize 'my-function)
 => my_function
(pascalize 'my_function)
 => my___function
(pascalize 'let*)
 => let__star_
```

Miscellaneous

message-box
pascalize
version

+ **&rest NUMBERS**

Returns the sum of NUMBERS. If all the NUMBERS are integers the result is an integer otherwise it is a floating number.

Examples:

```
(+ 0 1 2 3 4 5 6 7 8 9 10)
 => 55
(+)
 => 0
(+ 3.1 42)
 => 45.1
```

* **&rest NUMBERS**

Returns the product of NUMBERS. If all the NUMBERS are integers the result is an integer otherwise it is a floating number.

Examples:

```
(* 1.5 2.0)
 => 3.0
(* 3 4)
 => 12
```

rem **INTEGER1 INTEGER2**

Returns the remainder of INTEGER1 divided by INTEGER2. The result has the same sign as INTEGER1.

Examples:

```
(rem 52 10)
 => 2
(rem 52 -10)
 => 2
(rem -52 10)
 => -2
(rem -52 -10)
 => -2
```

mod **INTEGER1 INTEGER2**

Returns the INTEGER1 modulo INTEGER2. The result has the same sign as INTEGER2.

Examples:

```
(mod 52 10)
 => 2
(mod 52 -10)
 => -8
(mod -52 10)
 => 8
(mod -52 -10)
 => -2
```

/ NUMBER1 &optional NUMBER2

Returns NUMBER1 divided by NUMBER2. If both numbers are integers the result is an integer, rounded down towards zero, otherwise the answer is a floating number.

If only NUMBER1 is given, it returns the reciprocal of NUMBER1.

Examples:

```
(/ 34.0 3.0)
 => 11.3333333333333
(/ 34 3)
 => 11
(/ 34.0 3)
 => 11.3333333333333
(/ 3.0)
 => 3.33333333333333e-1
```

- **NUMBER1 &optional NUMBER2**

Returns NUMBER1 minus by NUMBER2. If both numbers are integers the result is an integer otherwise the answer is a floating number.

If only NUMBER1 is given it returns -NUMBER1.

Examples:

```
(- 45)
 => -45
(- 45 32)
 => 13
```

max **NUMBER1 &rest NUMBERS**

Returns the max of NUMBER1 and NUMBERS. If all the NUMBERS are integers the result is an integer otherwise a floating number is returned.

Examples:

```
(max 1 2 3 4 5)
 => 5
(max 1 2 3.0 4 5)
 => 5.0
```

min **NUMBER1 &rest NUMBERS**

Returns the min of NUMBER1 and NUMBERS. If all the NUMBERS are integers the result is an integer otherwise a floating number is returned.

Examples:

```
(min 1 2 3 4 5)
 => 1
(min 1 2 3.0 4 5)
 => 1.0
```

1- INTEGER

Returns INTEGER minus one.

Example:

(1- 45)

=> 44

1+ INTEGER

Returns INTEGER plus one.

Example:

```
(1+ 45)
 => 46
```

Arithmetic

*
=
±
-
=
/
1+
1-
max
min
mod
rem

sin **ANGLE**

return sine of ANGLE (given in radians).

Example:

```
(sin (/ pi 6))
 => 5.0e-1
```

COS ANGLE

return cosine of ANGLE (given in radians).

Example:

```
(cos (/ pi 6))
 => 8.66025403784439e-1
```

arctan TAN

return arctan of TAN in radians

Example:

```
(* (arctan 1.0) 4)
 => 3.14159265358979
```

exp **EXPONENT**

return e (2.718281828459...) to the power of EXPONENT. Opposite of ln.

Examples:

```
(exp 1.0)
 => 2.71828182845905
(exp (ln 2.9))
 => 2.9
```

sqrt x

return square root of X.

Examples:

```
(sqrt 2)
 => 1.4142135623731
(sqrt 1024)
 => 32.0
```

ln x

return the natural log of X (to base e = 2.718281828459...). Opposite of exp.

Examples:

```
(/ (ln 1000) (ln 10))
 => 3.0
(ln (exp 2.9))
 => 2.9
```

round x

return floating point number X rounded to the nearest integer.

Examples:

```
(round 2.3)
 => 2
(round -2.5)
 => -2
(round 1.9)
 => 2
```

trunc x

return floating point number X truncated zero-wards to an integer.

Examples:

```
(trunc 2.3)
 => 2
(trunc -2.5)
 => -2
(trunc 1.9)
 => 1
```

float **NUMBER**

Convert an integer or floating number to a floating number.

Examples:

```
(float 2)
 => 2.0
(float 3.5)
 => 3.5
```

Math functions

arctan
cos
exp
float
ln
round
sin
sqrt
trunc

prin1-length **OBJECT &optional MAX-LENGTH**

Returns number of characters printed out if OBJECT was printed using prin1 or prin1-to-string, with a maximum of MAX-LENGTH (with default 32767). The maximum prevents an infinite loop if the OBJECT contains circular pointers.

Examples:

```
(prin1-length "Hello")
 => 7
(prin1-length '(+ 2 34))
 => 8
```

prin1-to-string OBJECT

Returns the string of characters if OBJECT was printed out by prin1.

Example:

```
(prin1-to-string '(a b (23 4)))
 => "(a b (23 4))"
```

Printing to strings

prin1-length
prin1-to-string

case **KEY-EXPR &rest CASE-CLAUSES**

Use case to select one of a group of actions or values based on the value of an expression

The KEY-EXPR is evaluated. Each of the CASE-CLAUSEs is of the form (key | [([key] ...)] [expr] ...). The first CASE-CLAUSE such that the value of KEY-EXPR is eql to the KEY or one of the KEYs in the list at the head of the clause has its EXPR's evaluated in order. If the KEY at the head of a clause is the symbol t then this counts as being eql to the value of KEY-EXPR. (The KEYs are not evaluated.) The value returned is the last value returned by executing an expr from the clause list evaluated if any, otherwise nil.

Examples:

```
(case (+ 1 1) (1 (+ 2 3)) (2 (+ 3 4)) (t 'fred))
 => 7
(case (+ 1 2) (1 (+ 2 3)) (2 (+ 3 4)) (t 'fred))
 => fred
```

cond **&rest COND-CLAUSES**

Use cond to select one of a group of actions or values based on satisfying a condition

Each of the COND-CLAUSEs is of the form (condition [expr] ...). The first COND-CLAUSE such that CONDITION evaluates to true has its EXPRs evaluated in order. The value returned is the last value returned by executing an EXPR from the clause list evaluated if any, otherwise the value returned by the CONDITION of the successful COND-CLAUSE.

Example:

```
(cond ((= 1 2) 'a) ((= 1 1) 'b) ((= 2 2) 'c))

 => b
```

throw **TAG EXPR**

Use throw to perform an exit with dynamic scope to a corresponding catch.

TAG and EXPR are both evaluated. If there is a catch currently executing with the same TAG, then the execution of expressions in the catch is aborted, and the catch returns the value of EXPR. If no catch with the same TAG is executing, then an error is signaled. TAGs are compared using eq.

Example:

```
(catch 'jim (throw 'jim 34) 36)

=> 34
```

catch **TAG &rest EXPR**

Use catch to trap a dynamic exit initiated by a corresponding throw.

Catch evaluates the TAG and then evaluates the EXPRs in turn. If a throw with the same TAG occurs during this evaluation then the value returned is the value passed by the throw, otherwise it is the value of the last EXPR evaluated.

Examples:

```
(catch 'jim (throw 'jim 34) 36)
 => 34
(catch 'jim 36)
 => 36
```

dotimes **(COUNTER LIMIT &optional RESULT) &rest EXPR**

Use dotimes to perform a fixed series of actions a certain number of times.

A local variable called COUNTER is created and its value starts at 0, incrementing by 1 each iteration until reaching LIMIT - 1 on the last iteration. For each iteration the EXPRs are evaluated in order. The returned value is the evaluation of RESULT after the last iteration, if RESULT is supplied, otherwise nil.

Example:

```
(let ((a ())) (dotimes (i 3 a) (setq a (cons i a))))
 => (2 1 0)
```

dolist **(ELEMENT LIST &optional RESULT) &rest EXPR**

Use dolist to iterate over the elements of a list.

A local variable called ELEMENT is created and its value is set to each element of LIST in turn. For each iteration the EXPRs are evaluated in order. The returned value is the evaluation of RESULT after the last iteration, if RESULT is supplied, otherwise nil.

Example:

```
(let ((a 0)) (dolist (x '(1 20 300) a) (setq a (+ x a))))

=> 321
```

while **TEST &rest EXPR**

Use while to iterate while a certain condition holds true.

TEST is evaluated at the beginning of each iteration. If it evaluates to nil the iteration terminates, otherwise each of the EXPRs is evaluated in turn. The value returned is the result from the last EXPR evaluated if any, otherwise nil.

Example:

```
(let ((a 0)) (while (< a 10) (setq a (+ a 3))) a)

=> 12
```

progn **&rest EXPR**

Use progn to group a series of expressions together.

Each of the EXPRs is evaluated in order. The value returned is the value returned from the last EXPR if any, otherwise nil.

Example:

```
(progn (setq a 23) (setq a (+ a 100)) (+ a a))
 => 246
```

prog1 **EXPR1 &rest EXPR**

Use prog1 to evaluate a series of expressions, returning the result of the 1st one.

EXPR1 is evaluated, and its value becomes the return value of the prog1, then the other EXPRs are evaluated in order.

Example:

```
(prog1 (setq a 23) (setq a (+ a 100)) (+ a a))
 => 23
```

prog2 **EXPR1 EXPR2 &rest EXPR**

Use prog2 to evaluate a series of expressions in order, and return the result of the 2nd one.

EXPR1 is evaluated, followed by EXPR2, whose return value becomes the return value of the prog2, followed by the other EXPRs in order. Typically EXPR1 is some sort of startup code, and the EXPRs are some sort of cleanup code.

Example:

```
(prog2 (setq a 23) (setq a (+ a 100)) (+ a a))
 => 123
```

if **CONDITION EXPR1 &optional EXPR2**

Use if to make a choice based on a condition being true or false.

CONDITION is evaluated. If the value returned is non-nil then EXPR1 is evaluated and its value returned, otherwise if EXPR2 is given then it is evaluated and its value returned, otherwise nil is returned.

Examples:

```
(if (= 1 1) 'fred 'jim)
 => fred
(if (= 1 2) 'fred 'jim)
 => jim
```

when **TEST &rest EXPRS**

Use when to evaluate a series of expressions if a certain condition holds true.

TEST is evaluated. If its return value is nil then the when returns nil, otherwise each EXPR is evaluated in turn and the value of the last one if any is the return value, otherwise nil is returned.

Examples:

```
(when (= 1 1) (setq a 100) (+ a 25))
 => 125
(when (= 1 2) (setq a 100) (+ a 25))
 => ()
```

unless **TEST &rest EXPRS**

Use unless to evaluate a series of expressions if a certain condition doesn't hold true.

TEST is evaluated. If its return value is nil then each EXPR is evaluated in turn and the value of the last one if any is the return value, otherwise nil is returned.

Examples:

```
(unless (= 1 1) (setq a 100) (+ a 25))
 => ()
(unless (= 1 2) (setq a 100) (+ a 25))
 => 125
```

let **(&rest VARIABLE-DECLARATION) &rest EXPR**

Use let to define a series of local lexical variables, where the initialisation expressions of the variables are not in the scope of any of them.

Each variable-declaration is of the form variable or (variable) or (variable initialisation-expression). If the initialisation expression is not provided it defaults to nil. The initialisation expression of each variable is evaluated outside of the scope of the let, i.e. no initialisation expression can make reference to any of the variables in the same let. Each local variable is then created and given the value returned from its initialisation expression. The EXPRs are then evaluated in order with these variables defined.

Any variable (local or global) with the same name as one of the let variables that exists before the let is 'shadowed' during the let, i.e. the most newly defined variable takes precedence. But the existing variables continue to exist, and can be referred to again once the let has terminated.

Example:

```
(let ((a 10)) (setq a (* a a)) (+ a (let ((a 9)) a)))

=> 109
```

let* **(&rest VARIABLE-DECLARATION) &rest EXPR**

Use let* to define a series of local lexical variables, where the initialisation expressions of the variables can refer to preceding variables in the let*.

Each variable-declaration is of the form variable or (variable) or (variable initialisation-expression). If the initialisation expression is not provided it defaults to nil. For each variable the initialisation expression is evaluated and then the variable is created with that initial value. This means that the initialisation expression for each variable can refer to variables already defined in the same let*, but not the variable currently being initialised or ones yet to com. The EXPRs are then evaluated in order with these variables defined.

Any variable (local or global) with the same name as one of the let* variables that exists before the let* is 'shadowed' during the let*, i.e. the most newly defined variable takes precedence. But the existing variables continue to exist, and can be referred to again once the let* has terminated.

Example:

```
(let* ((a 100) (b (+ a 2))) (* a b))
 => 10200
```

unwind-protect **EXPR &rest CLEANUP-EXPR**

Use unwind-protect to guarantee evaluation of cleanup code.

EXPR is evaluated, and then the CLEANUP-EXPRs are evaluated. If a non-returning exit occurs during evaluation of EXPR, the CLEANUP-EXPRs are still evaluated before completion of the exit. A non-returning exit could be caused by a throw or a signaled error. If a non-returning exit occurs during evaluation of the CLEANUP-EXPRs then the original exit will be abandoned.

If no non-returning exit occurs then the result of evaluating EXPR is returned.

Control structures

case
catch
cond
dolist
dotimes
if
let
let*
prog1
prog2
progn
throw
unless
unwind-protect
when
while

setf **PLACE VALUE**

Use setf to set the value of a generalised variable.

The PLACE argument is an expression that is a macro or function call that extracts data from some position. Setf effectively inserts a new VALUE into PLACE. This is done using a setf method which must be defined for any function or macro that setf is used with. Setf methods are predefined for the following functions - aref background-color caar cadr car cdar cddr cdr char cursor-pos double-click-time first fourth gc-limit get nth pixel profile-string rest second symbol-function symbol-value system-color text-color third viewport-extent viewport-origin window-data window-extent window-origin window-painter window-text . PLACE can also be the name of a variable, in which case the value of that variable is set to VALUE (as by setq).

defsetf can be used to define setf methods for any function or macro.

Setf works as a macro, creating a new expression to be evaluated whose head is the name or value of the setf method, and whose arguments are the arguments to PLACE followed by VALUE.

Setf returns VALUE.

Example:

```
(let ((x '(c b))) (setf (car x) 'c) x)

=> (c b)
```

defsetf **PLACE FUN**

Use defsetf to define a setf method for an arbitrary function or macro.

It defines the setf method for PLACE to be FUN, i.e. when the place for setf is an expression starting with PLACE, use FUN on the same arguments with the setf value added to the end to set a new value for PLACE. The return value for setf is the new value put into the given place, and it is the responsibility of FUN to return this value (the reason for this being that often a setf method ends up by itself calling setf, so requiring the setf method to return the value avoids setf itself doing the same work repeatedly).

Setf functions

defsetf
setf

make-array **SIZE &key INITIAL-ELEMENT**

Return an array containing SIZE elements which are initially all INITIAL-ELEMENT (default value nil).

Example:

```
(make-array 3 :initial-element 'trevor)
 => #(trevor trevor trevor)
```

aref **ARRAY INDEX**

Return member number INDEX of ARRAY. For an array of size n, the index goes from 0 to n-1.

Can be used with setf.

Example:

```
(aref #(a b c d e) 3)
 => d
```

vector **&rest ELEMENTS**

Returns a one-dimensional array whose members are ELEMENTS.

Example:

```
(vector 2 3 5 7)
 => #(2 3 5 7)
```

Array functions

aref
make-array
vector

char STRING POSITION

Retrieves character from POSITION in STRING. Positions in the string start from 0.

Can be used with setf to set the value of a character in a string.

Example:

```
(char "Hello" 1)
 => #\e
```

string-downcase STRING

Returns a string the same as STRING except that upper case characters are converted to lower case.

Example:

```
(string-downcase "AbCdEf")
 => "abcdef"
```

string-upcase **STRING**

Returns a string the same as STRING except that lower case characters are converted to upper case.

Example:

```
(string-upcase "AbCdEf")
 => "ABCDEF"
```

strcat **&rest STRING**

Returns a single string which is the concatenation of all the STRINGs.

Example:

```
(strcat "Goo" "d morn" "ing")
 => "Good morning"
```

make-string **LENGTH &key INITIAL-ELEMENT**

Make a string of LENGTH characters long with each character = INITIAL-ELEMENT (defaults to blank).

Examples:

```
(make-string 12 :initial-element #\Z)
 => "ZZZZZZZZZZZZ"
(make-string 12)
 => "            "
```

string OBJECT

Coerce OBJECT, which must be a character, symbol or string to an equivalent string - the name of a symbol, or a one letter string containing the character, or the same string respectively.

Examples:

```
(string "trevor")
 => "trevor"
(string 'trevor)
 => "trevor"
(string #\k)
 => "k"
```

string< **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is less than STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string< "abc" "abde")
 => t
(string< "abde" "abc")
 => ()
(string< "abc" "ABC")
 => ()
(string< "ABC" "abc")
 => t
(string< "abc" "abc")
 => ()
(string< "abc" "ABDE")
 => ()
(string< "ABDE" "abc")
 => t
```

string<= **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is less than or equal to STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string<= "abc" "abde")
 => t
(string<= "abde" "abc")
 => ()
(string<= "abc" "ABC")
 => ()
(string<= "ABC" "abc")
 => t
(string<= "abc" "abc")
 => t
(string<= "abc" "ABDE")
 => ()
(string<= "ABDE" "abc")
 => t
```

string= **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is equal to STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string= "abc" "abde")
 => ()
(string= "abde" "abc")
 => ()
(string= "abc" "ABC")
 => ()
(string= "ABC" "abc")
 => ()
(string= "abc" "abc")
 => t
(string= "abc" "ABDE")
 => ()
(string= "ABDE" "abc")
 => ()
```

string/= **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is not equal to STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string/= "abc" "abde")
 => t
(string/= "abde" "abc")
 => t
(string/= "abc" "ABC")
 => t
(string/= "ABC" "abc")
 => t
(string/= "abc" "abc")
 => ()
(string/= "abc" "ABDE")
 => t
(string/= "ABDE" "abc")
 => t
```

string>= **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is greater than or equal to STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string>= "abc" "abde")
 => ()
(string>= "abde" "abc")
 => t
(string>= "abc" "ABC")
 => t
(string>= "ABC" "abc")
 => ()
(string>= "abc" "abc")
 => t
(string>= "abc" "ABDE")
 => t
(string>= "ABDE" "abc")
 => ()
```

string> **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is greater than STRING2 otherwise nil (i.e. false).

Comparison is case-sensitive.

Examples:

```
(string> "abc" "abde")
 => ()
(string> "abde" "abc")
 => t
(string> "abc" "ABC")
 => t
(string> "ABC" "abc")
 => ()
(string> "abc" "abc")
 => ()
(string> "abc" "ABDE")
 => t
(string> "ABDE" "abc")
 => ()
```

string-lessp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is less than STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-lessp "abc" "abde")
 => t
(string-lessp "abde" "abc")
 => ()
(string-lessp "abc" "ABC")
 => ()
(string-lessp "ABC" "abc")
 => ()
(string-lessp "abc" "abc")
 => ()
(string-lessp "abc" "ABDE")
 => t
(string-lessp "ABDE" "abc")
 => ()
```

string-not-greaterp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is less than or equal to STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-not-greaterp "abc" "abde")
 => t
(string-not-greaterp "abde" "abc")
 => ()
(string-not-greaterp "abc" "ABC")
 => t
(string-not-greaterp "ABC" "abc")
 => t
(string-not-greaterp "abc" "abc")
 => t
(string-not-greaterp "abc" "ABDE")
 => t
(string-not-greaterp "ABDE" "abc")
 => ()
```

string-equalp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is equal to STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-equalp "abc" "abde")
 => ()
(string-equalp "abde" "abc")
 => ()
(string-equalp "abc" "ABC")
 => t
(string-equalp "ABC" "abc")
 => t
(string-equalp "abc" "abc")
 => t
(string-equalp "abc" "ABDE")
 => ()
(string-equalp "ABDE" "abc")
 => ()
```

string-not-equalp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is not equal to STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-not-equalp "abc" "abde")
 => t
(string-not-equalp "abde" "abc")
 => t
(string-not-equalp "abc" "ABC")
 => ()
(string-not-equalp "ABC" "abc")
 => ()
(string-not-equalp "abc" "abc")
 => ()
(string-not-equalp "abc" "ABDE")
 => t
(string-not-equalp "ABDE" "abc")
 => t
```

string-not-lessp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is greater than or equal to STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-not-lessp "abc" "abde")
 => ()
(string-not-lessp "abde" "abc")
 => t
(string-not-lessp "abc" "ABC")
 => t
(string-not-lessp "ABC" "abc")
 => t
(string-not-lessp "abc" "abc")
 => t
(string-not-lessp "abc" "ABDE")
 => ()
(string-not-lessp "ABDE" "abc")
 => t
```

string-greaterp **STRING1 STRING2**

Compare STRING1 and STRING2 using normal alphabetical order.Return t (i.e. true) if STRING1 is greater than STRING2 otherwise nil (i.e. false).

Comparison is case-insensitive.

Examples:

```
(string-greaterp "abc" "abde")
 => ()
(string-greaterp "abde" "abc")
 => t
(string-greaterp "abc" "ABC")
 => ()
(string-greaterp "ABC" "abc")
 => ()
(string-greaterp "abc" "abc")
 => ()
(string-greaterp "abc" "ABDE")
 => ()
(string-greaterp "ABDE" "abc")
 => t
```

String functions

char
make-string
strcat
string
string-downcase
string-equalp
string-greaterp
string-lessp
string-not-equalp
string-not-greaterp
string-not-lessp
string-upcase
string/=
string<
string<=
string=
string>
string>=

intern **STRING**

Create an interned symbol whose print name is STRING.

Examples:

```
(intern "jim")
 => jim
(eql (intern "jim") (intern "jim"))
 => t
```

make-symbol **STRING**

Create an un-interned symbol whose print name is STRING.

Examples:

```
(make-symbol "jim")
 => #:jim
(eql (make-symbol "jim") (make-symbol "jim"))
 => ()
```

symbol-name **SYMBOL**

Return print-name of SYMBOL.

Examples:

```
(symbol-name 'jim)
 => "jim"
(symbol-name '#:jim)
 => "jim"
(symbol-name :jim)
 => "jim"
```

gensym **OPTIONAL STRING**

Create an uninterned symbol with a unique name - its name is STRING (default value "g" with a counter appended to the end. The counter increments by 1 each time gensym is called.

Useful in macro definitions where you need to generate a guaranteed unique local variable name.

Examples:

```
(gensym)
 => #:g1314
(gensym)
 => #:g1315
(let ((sym (gensym))) (eql sym (intern (symbol-name sym))))
 => ()
```

defconstant **SYMBOL EXPR**

Define a constant called SYMBOL (unevaluated) and assign it the value returned from evaluating EXPR. A constant is a global variable whose value cannot be changed.

Example:

```
(defconstant e (exp 1))

=> e
```

set **SYMBOL EXPR**

A function that sets the global value of the variable SYMBOL to EXPR. Contrast with setq which does not evaluate the name of its variable and can refer to lexical variables.

Example:

```
(- (let ((a 300)) (set 'a 10) a) a)
 => 290
```

setq **SYMBOL EXPR**

Set the value of the variable SYMBOL (unevaluated) to EXPR (evaluated). Refers to the innermost defined lexical variable whose scope the setq lies in, otherwise to the global variable of that name. May take a series of arguments, treated as pairs of SYMBOL and EXPR, each pair being setq-ed in order.

Returns the last value set.

Examples:

```
(let ((a 20)) (setq a 30) a)

 => 30

(progn (setq a 20) (symbol-value 'a))

 => 20
```

psetq **SYMBOL EXPR ...**

Same as setq but if more than one SYMBOL-EXPR pair is given, all the EXPRs are evaluated first, and then assigned to the variables, so that any reference to a SYMBOL in an EXPR always refers to the value of the variable SYMBOL before starting the psetq.

Returns the last value set.

Example:

```
(let ((a 10) (b 20)) (psetq a b b a) (list a b))
 => (20 10)
```

symbol-value **SYMBOL**

Retrieve global value of the variable SYMBOL.

Can be used with setf.

Example:

```
(progn (setq a 25) (let ((a 3)) (symbol-value 'a)))
 => 25
```

symbol-function **SYMBOL**

Return the function value of SYMBOL.

Can be used with setf.

Example:

```
(symbol-function 'car)
 => <Builtin: car>
```

get **SYMBOL1 SYMBOL2**

Return the value of the SYMBOL2 property in SYMBOL1's property list, or nil if SYMBOL2 doesn't have a value in SYMBOL1's property list.

Can be used with setf.

Example:

```
(progn (setf (get 'jim 'age) 32) (get 'jim 'age))

=> 32
```

remprop **SYMBOL1 SYMBOL2**

Remove SYMBOL2 and its property from SYMBOL1's property list.

Examples:

```
(setf (get 'a 'b) 'c)
 => c
(get 'a 'b)
 => c
(remprop 'a 'b)
 => ()
(get 'a 'b)
 => ()
```

symbol-plist **SYMBOL**

Return value SYMBOLS's property list - consisting of a list of alternating property names and values.

Examples:

```
(setf (get 'jim 'age) 32)
 => 32
(symbol-plist 'jim)
 => (age 32)
```

boundp SYMBOL

Return t (i.e. true) if the global value of the variable SYMBOL has a value otherwise nil (i.e. false).

Examples:

```
(boundp (gensym))
 => ()
(setq a 45)
 => 45
(boundp 'a)
 => t
```

fboundp SYMBOL

Return t (i.e. true) if the function value of SYMBOL is defined otherwise nil (i.e. false).

Examples:

```
(fboundp (gensym))
 => ()
(fboundp 'car)
 => t
```

makunbound SYMBOL

Make the global value of the variable SYMBOL undefined.

Examples:

```
(setq a 32)
 => 32
(makunbound 'a)
 => a
(boundp 'a)
 => ()
```

fmakunbound SYMBOL

Make the function value of SYMBOL undefined.

Examples:

```
(defun a ())
 => a
(fboundp 'a)
 => t
(fmakunbound 'a)
 => a
(fboundp 'a)
 => ()
```

Symbol and variable functions

boundp
defconstant
fboundp
fmakunbound
gensym
get
intern
make-symbol
makunbound
psetq
remprop
set
setq
symbol-function
symbol-name
symbol-plist
symbol-value

cons **OBJECT1 OBJECT2**

Return a cons whose car is OBJECT1 and cdr is OBJECT2. If OBJECT2 is a list, this will be the list consisting of the elements of OBJECT2 with OBJECT1 added on to the front.

Examples:

```
(cons 2 3)
 => (2 . 3)
(cons 2 '(3 4))
 => (2 3 4)
(cons 2 ())
 => (2)
(cons 2 '(3 . 4))
 => (2 3 . 4)
```

car CONS

Return the car of the cons CONS. If CONS is a list then this is the first element of the list. Identical to first.

Can be used with setf.

Examples:

```
(car '(2 3))
 => 2
(car '(2 . 3))
 => 2
```

first LIST

Return the first element of LIST. Identical to car.

Can be used with setf.

Example:

```
(first '(2 3))
 => 2
```

cdr CONS

Return the cdr of the cons CONS. If CONS is a list then this is the sublist consisting of CONS minus its first element. Identical to first.

Can be used with setf.

Examples:

```
(cdr '(2))
 => ()
(cdr '(2 3 4))
 => (3 4)
(cdr '(2 . 3))
 => 3
```

rest LIST

Return the a list consisting of the elements of LIST minus the first one. Identical to cdr.

Can be used with setf.

Examples:

```
(rest '(2 3 4))
 => (3 4)
(rest '(2))
 => ()
```

rplaca CONS OBJECT

Replace the car of CONS with OBJECT, returning the changed CONS.

Example:

```
(rplaca '(c b) 'c)
 => (c b)
```

rplacd **CONS OBJECT**

Replace the cdr of CONS with OBJECT, returning the changed CONS.

Example:

```
(rplacd '(a . c) 'c)
 => (a . c)
```

list **&rest OBJECT**

Return a list whose elements are the OBJECTs in the order provided.

Examples:

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

append &rest LIST

Return a new list which is the concatenation of all the LISTs. Returned list shares structure with the last LIST, which doesn't have to be a true (non-dotted) list.

Examples:

```
(append '(a b) '(c) '(d e) '() '(f g h))
 => (a b c d e f g h)
(append)
 => ()
(append '(a b) '(c . d))
 => (a b c . d)
```

reverse LIST

Return a list whose elements are the elements of LIST in reverse order.

Example:

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

member **OBJECT LIST**

If OBJECT belongs to LIST, return the list consisting of OBJECT and all the elements that follow the first occurrence of OBJECT in LIST, otherwise return nil. Effectively a predicate, since if OBJECT is in LIST the result is non-nil. If a non-nil list is returned, this list shares structure with LIST.

Examples:

```
(member 'c '(a b c d e))

 => (c d e)

(member 'f '(a b c d e))

 => ()

(member "jim" '("tom" "jim"))

 => ()
```

remove **OBJECT LIST**

Return a new list consisting of those elements in LIST not eql to OBJECT.

Examples:

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

last LIST

Return the last element of LIST.

Example:

```
(last '(u v w x y z))
 => z
```

nth **N LIST**

Return the (N+1)th element of LIST.

Can be used with setf.

Example:

```
(nth 4 '(a b c d e f))
 => e
```

nthcdr **N LIST**

Return the list consisting of the (N+1)th element of LIST and all those that follow. Returned list shares structure with LIST.

Example:

```
(nthcdr 4 '(a b c d e f))

 => (e f)
```

nconc **&rest LIST**

Returns the list which is the concatenation of all the LISTs, formed by destructively modifying the LISTs.

Examples:

```
(nconc '(a b c d e) '(d e))
 => (a b c d e)
(let ((a '(a b c d))) (nconc a '(c d)) a)
 => (a b c d)
```

delete **OBJECT LIST**

Return the list consisting of those elements of LIST not eql to OBJECT, formed by destructively modifying LIST.

Examples:

```
(delete 'a '(a b c d e))
 => (b c d e)
(let ((list '(a b c))) (delete 'a list) list)
 => (a b c)
```

second LIST

Return 2nd element of LIST.

Can be used with setf.

Example:

```
(second '(one two three four))
 => two
```

third LIST

Return 3rd element of LIST.

Can be used with setf.

Example:

```
(third '(one two three four))
 => three
```

fourth LIST

Return 4th element of LIST.

Can be used with setf.

Example:

```
(fourth '(one two three four))
 => four
```

caar CONS

(caarx) = (car (car x))

cadr CONS

(cadrx) = (car (cdr x))

cdar CONS

(cdarx) = (cdr (car x))

cddr CONS

(cddrx) = (cdr (cdr x))

caaar CONS

(caaarx) = (car (car (car x)))

caadr CONS

(caadrx) = (car (car (cdr x)))

cadar CONS

(cadarx) = (car (cdr (car x)))

caddr CONS

(caddrx) = (car (cdr (cdr x)))

cdaar CONS

(cdaarx) = (cdr (car (car x)))

cdadr CONS

(cdadrx) = (cdr (car (cdr x)))

cddar CONS

(cddarx) = (cdr (cdr (car x)))

cdddr CONS

(cdddrx) = (cdr (cdr (cdr x)))

caaaar CONS

(caaaarx) = (car (car (car (car x))))

caaadr CONS

(caaadrx) = (car (car (car (cdr x))))

caadar CONS

(caadarx) = (car (car (cdr (car x))))

caaddr CONS

(caaddrx) = (car (car (cdr (cdr x))))

cadaar CONS

(cadaarx) = (car (cdr (car (car x))))

cadadr CONS

(cadadrx) = (car (cdr (car (cdr x))))

caddar CONS

(caddarx) = (car (cdr (cdr (car x))))

cadddr CONS

(cadddrx) = (car (cdr (cdr (cdr x))))

cdaaar CONS

(cdaaarx) = (cdr (car (car (car x))))

cdaadr CONS

(cdaadrx) = (cdr (car (car (cdr x))))

cdadar CONS

(cdadarx) = (cdr (car (cdr (car x))))

cdaddr CONS

(cdaddrx) = (cdr (car (cdr (cdr x))))

cddaar CONS

(cddaarx) = (cdr (cdr (car (car x))))

cddadr CONS

(cddadrx) = (cdr (cdr (car (cdr x))))

cdddar CONS

(cdddarx) = (cdr (cdr (cdr (car x))))

cddddr CONS

(cddddrx) = (cdr (cdr (cdr (cdr x))))

assoc OBJECT ALIST

Return the first element of ALIST (if any) such that the car of the element is eql to OBJECT, otherwise return nil.

Examples:

```
(assoc '1 '((0 zero) (1 one) (2 two)))
 => (1 one)
(assoc '3 '((0 zero) (1 one) (2 two)))
 => ()
```

sublis ALIST OBJECT

Recursively substitute any occurrence of the car of an element in ALIST in OBJECT with the cdr of the same element. eql is used to judge equality of elements in ALIST and OBJECT. sublis looks deep into nested lists but not into arrays or structures.

Example:

```
(sublis '((cat . dog) (the . a)) '((the cat) sat (on (the mat))))

=> ((a dog) sat (on (a mat)))
```

subst **NEW-OBJECT OLD-OBJECT OBJECT**

Recursively substitute any occurrence of OLD-OBJECT in OBJECT with NEW-OBJECT. eql is used to judge equality of OLD-OBJECT and elements in OBJECT. subst looks deep into nested lists but not into arrays or structures.

Example:

```
(subst 'new 'old '(old (jim (old old))))
 => (new (jim (new new)))
```

mapcar **FUN LIST1 &rest LIST**

Apply FUN to each element of LIST1 and the corresponding elements of the LISTs , returning the results in a list.

It follows that the number of LISTs including LIST1 must be an acceptable number of arguments to FUN. An error occurs if any of the LISTs is shorter than LIST1.

Example:

```
(mapcar #'list '(a b c) '(1 2 3))

=> ((a 1) (b 2) (c 3))
```

mapc **FUN LIST1 &rest LIST**

Apply FUN to each element of LIST1 and the corresponding elements of the LISTs , returning LIST1.

It follows that the number of LISTs including LIST1 must be an acceptable number of arguments to FUN. An error occurs if any of the LISTs is shorter than LIST1.

maplist **FUN LIST1 &rest LIST**

Apply FUN to each sublist of LIST1 (starting with LIST1, then its cdr, and its cdr etc., up to but not including nil) and each corresponding sublist of the LISTs , returning the results in a list.

It follows that the number of LISTs including LIST1 must be an acceptable number of arguments to FUN. An error occurs if any of the LISTs is shorter than LIST1.

Example:

```
(maplist #'list '(a b c) '(1 2 3))

=> (((a b c) (1 2 3)) ((b c) (2 3)) ((c) (3)))
```

mapl **FUN LIST1 &rest LIST**

Apply FUN to each sublist of LIST1 (starting with LIST1, then its cdr, and its cdr etc., up to but not including nil) and each corresponding sublist of the LISTs , returning LIST1.

It follows that the number of LISTs including LIST1 must be an acceptable number of arguments to FUN. An error occurs if any of the LISTs is shorter than LIST1.

List and cons functions

append
assoc
caaaar
caaadr
caaar
caadar
caaddr
caadr
caar
cadaar
cadadr
cadar
caddar
cadddr
caddr
cadr
car
cdaaar
cdaadr
cdaar
cdadar
cdaddr
cdadr
cdar
cddaar
cddadr
cddar
cdddar
cddddr
cdddr
cddr
cdr
cons
delete
first
fourth
last
list
mapc
mapcar
mapl
maplist
member
nconc
nth
nthcdr
remove
rest
reverse
rplaca
rplacd
second

sublis
subst
third

eq **OBJECT1 OBJECT2**

Return t (i.e. true) if object1 is the same object as object2 otherwise nil (i.e. false).Note that the only type of object guaranteed to be eq if they have the same written representation is symbols.

Examples:

```
(eq 'a 'a)
 => t
(eq "a" "a")
 => ()
(eq 34 34)
 => ()
(eq 'a 'b)
 => ()
(let ((a "jim")) (eq a a))
 => t
```

eql **OBJECT1 OBJECT2**

Return t (i.e. true) if object1 is the same object as object2, or both objects are numbers with equal values otherwise nil (i.e. false).

Examples:

```
(eql 'a 'a)
 => t
(eql "a" "a")
 => ()
(eql 34 34)
 => t
(eql 'a 'b)
 => ()
(let ((a "jim")) (eql a a))
 => t
```

equal OBJECT1 OBJECT2

Return t (i.e. true) if object1 and object2 are the same object, or they are integers, floats or strings with the same values, or they are conses (or lists) whose members are equal. otherwise nil (i.e. false).Note that this implies that arrays and structs are not compared for equality of members.

Examples:

```
(equal 34 34)
 => t
(equal "jim" "fred")
 => ()
(equal '(1 2 3) '(1 2 3))
 => t
(equal '(1 2 #(3 4)) '(1 2 #(3 4)))
 => ()
(equal '(one two) '(one two))
 => t
```

Equality

eq
eql
equal

eval **EXPRESSION**

Evaluate EXPRESSION. Since eval is a function it effectively evaluates it's argument twice.

Examples:

```
(eval '(+ 2 3))
 => 5
(eval (cons '* '(3 4)))
 => 12
```

quote **EXPRESSION**

Returns ARG unevaluated. Particularly useful when you want to specify a constant symbol or list as an argument to a function. 'expr is an abbreviation for (quote expr), and the lisp system always uses this abbreviation when printing out quoted expressions

Examples:

```
'jim
 => jim
(list 'jim 'fred)
 => (jim fred)
(first ''tom)
 => quote
''jim
 => 'jim
```

backquote TEMPLATE

TEMPLATE is not evaluated, except for certain sub-expressions. The sub-expressions are evaluated "in-place". A two element list whose first element is the symbol comma returns the evaluation of the second element. A two element list whose first element is the symbol comma-at returns the evaluation of the second element and this result is "spliced" into the list that the comma-at list is a member of.

backquote is useful when you want to return an expression that is mainly constant except for a few sub-expressions, and commonly is used in macro definitions using defmacro.

backquote is evaluated recursively. This means that if a backquoted expression appears inside TEMPLATE, any comma or comma-at expressions immediately inside the backquoted expression are not evaluated, but any comma or comma-at expressions inside those expressions are evaluated. In general, evaluation is delayed until the number of comma or comma-at's is equal to the number of backquotes. This feature can be useful for writing macros that write macros.

The symbols backquote, comma and comma-at all have bracketless abbreviations as with quote. These are

`x = (backquote x)

,x = (comma x)

,@ = (comma-at x)

Examples:

```
`(1 2 ,(+ 1 2))
=> (1 2 3)
`(`(a ,(+ 1 ,(+ 1 2))))
=> (`(a ,(+ 1 3)))
`(a b ,@(list 'c 'd) e)
=> (a b c d e)
```

comma x

comma is not a function or form, but it has a special interpretation when used with backquote.

comma-at x

comma-at is not a function or form, but it has a special interpretation when used with backquote.

funcall **FUN &rest ARGS**

Call the function FUN with arguments ARGS.

Examples:

```
(funcall #'+ 2 3)
 => 5
(funcall #'(lambda (x y) (+ x y)) 3 4)
 => 7
```

apply **FUNCTION ARGS**

Call the function FUN with argument list ARGS.

Examples:

```
(apply #'+ '(2 3))
 => 5
(apply #'(lambda (x y) (+ x y)) (list 3 4))
 => 7
```

load **FILE :print PRINT :verbose VERBOSE**

Read each top-level lisp expression in FILE and evaluate each one as its read. If PRINT is t then print out the result from each expression. If VERBOSE is t then print a message at the beginning saying that the file is being loaded. Returns t if loading occurs successfully.

Evaluation functions

apply
backquote
comma
comma-at
eval
funcall
load
quote

> NUMBER1 NUMBER2

Return t (i.e. true) if NUMBER1 is greater than NUMBER2 otherwise nil (i.e. false).

Examples:

```
(> 2 3)
 => ()
(> 3 2)
 => t
(> 4.5 4.5)
 => ()
```

< NUMBER1 NUMBER2

Return t (i.e. true) if NUMBER1 is less than NUMBER2 otherwise nil (i.e. false).

Examples:

```
(< 2 3)
 => t
(< 3 2)
 => ()
(< 4.5 4.5)
 => ()
```

>= **NUMBER1 NUMBER2**

Return t (i.e. true) if NUMBER1 is greater than or equal to NUMBER2 otherwise nil (i.e. false).

Examples:

```
(>= 2 3)
 => ()
(>= 3 2)
 => t
(>= 4.5 4.5)
 => t
```

<= NUMBER1 NUMBER2

Return t (i.e. true) if NUMBER1 is less than or equal to NUMBER2 otherwise nil (i.e. false).

Examples:

```
(<= 2 3)
 => t
(<= 3 2)
 => ()
(<= 4.5 4.5)
 => t
```

= NUMBER1 NUMBER2

Return t (i.e. true) if NUMBER1 is equal to NUMBER2 otherwise nil (i.e. false).

Examples:

```
(= 2 3)
 => ()
(= 3 2)
 => ()
(= 4.5 4.5)
 => t
```

/= NUMBER1 NUMBER2

Return t (i.e. true) if NUMBER1 is not to equal to NUMBER2 otherwise nil (i.e. false).

Examples:

```
(/= 2 3)
 => t
(/= 3 2)
 => t
(/= 4.5 4.5)
 => ()
```

Comparing numbers

/=
≤
<=
=
≥
>=

logior **&rest INTEGERS**

Returns the bitwise or of the arguments.

Example:

```
(logior 1 3 7)
 => 7
```

logand &rest INTEGERS

Returns the bitwise and of the arguments.

Example:

```
(logand 1 3 7)
 => 1
```

logxor **&rest INTEGERS**

Returns the bitwise xor of the arguments.

Example:

```
(logxor 1 3 7)
 => 5
```

lognot **INTEGER**

Returns bitwise complement of INTEGER.

Examples:

```
(lognot 32)
 => -33
(lognot -1)
 => 0
```

Bitwise operations

logand
logior
lognot
logxor

not OBJECT

Return t (i.e. true) if OBJECT is not nil otherwise nil (i.e. false).Effectively returns logical negation of OBJECT, since nil is regarded as false and anything else as true.

Examples:

```
(not (< 3 2))
 => t
(not t)
 => ()
(not ())
 => t
```

and **&rest EXPRS**

EXPRS are not automatically evaluated. Rather, they are evaluated in order until one of them returns nil, or all the EXPRS have been evaluated. Returns result of last EXPR actually evaluated, or t if there are zero exprs. Effectively a logical OR of exprs with short-circuit evaluation, because nil is regarded as false and anything else as true.

Examples:

```
(and t t t)
 => t
(and t () t)
 => ()
(and (< 2 3) (< 3 4))
 => t
```

or **&rest EXPRS**

EXPRS are not automatically evaluated. Rather, they are evaluated in order until one of them returns non-nil, or all the EXPRS have been evaluated. Returns result of last EXPR actually evaluated, or nil if there are zero exprs. Effectively a logical OR of exprs with short-circuit evaluation, because nil is regarded as false and anything else as true.

Examples:

```
(or t t t)
 => t
(or t () t)
 => t
(or (> 2 3) (< 3 4))
 => t
(or (> 2 3) (> 3 4))
 => ()
```

Logical functions

and
not
or

consp OBJECT

Return t (i.e. true) if OBJECT is a cons otherwise nil (i.e. false).

Examples:

```
(consp 34)
 => ()
(consp ())
 => ()
(consp '(a b))
 => t
(consp '(a . b))
 => t
```

symbolp OBJECT

Return t (i.e. true) if OBJECT is a symbol otherwise nil (i.e. false).

Examples:

```
(symbolp 34)
 => ()
(symbolp 'jim)
 => t
(symbolp '(a b))
 => ()
```

keywordp OBJECT

Return t (i.e. true) if OBJECT is a keyword symbol, i.e. one that starts with a colon otherwise nil (i.e. false).

Examples:

```
(keywordp 34)
 => ()
(keywordp 'jim)
 => ()
(keywordp ':jim)
 => t
```

constantp **OBJECT**

Return t (i.e. true) if OBJECT evaluates to a constant value, i.e. it is either self-evaluating or it is a variable with a constant value. otherwise nil (i.e. false).

Examples:

```
(constantp 34)
 => t
(constantp '(car x))
 => ()
(constantp :jim)
 => t
(constantp 'ps_solid)
 => t
```

null OBJECT

Return t (i.e. true) if OBJECT is nil. otherwise nil (i.e. false).

Examples:

```
(null 3)
 => ()
(null ())
 => t
```

integerp **OBJECT**

Return t (i.e. true) if OBJECT is an integer otherwise nil (i.e. false).

Examples:

```
(integerp 3)
 => t
(integerp 4.2)
 => ()
(integerp 'jim)
 => ()
```

stringp OBJECT

Return t (i.e. true) if OBJECT is a string otherwise nil (i.e. false).

Examples:

```
(stringp 3)
 => ()
(stringp 45)
 => ()
(stringp "45")
 => t
```

characterp OBJECT

Return t (i.e. true) if OBJECT is a character otherwise nil (i.e. false).

Examples:

```
(characterp 45)
 => ()
(characterp 'jim)
 => ()
(characterp (char "jim" 2))
 => t
```

arrayp OBJECT

Return t (i.e. true) if OBJECT is an array otherwise nil (i.e. false).

Examples:

```
(arrayp 45)
 => ()
(arrayp '(a b))
 => ()
(arrayp #(a b))
 => t
```

structure-p **OBJECT**

Return t (i.e. true) if OBJECT is a structure otherwise nil (i.e. false).

Examples:

```
(structure-p 45)
 => ()
(structure-p 'jim)
 => ()
(structure-p (char "jim" 2))
 => ()
```

numberp OBJECT

Return t (i.e. true) if OBJECT is an integer or floating point number. otherwise nil (i.e. false).

Examples:

```
(numberp 34)
 => t
(numberp 34.6)
 => t
(numberp '(3 4))
 => ()
```

zerop **INTEGER**

Return t (i.e. true) if INTEGER equals zero otherwise nil (i.e. false).

Examples:

```
(zerop 0)
 => t
(zerop 1)
 => ()
(zerop -34)
 => ()
```

evenp INTEGER

Return t (i.e. true) if INTEGER is even otherwise nil (i.e. false).

Examples:

```
(evenp 0)
 => t
(evenp 1)
 => ()
(evenp 2)
 => t
```

oddp **INTEGER**

Return t (i.e. true) if INTEGER is odd otherwise nil (i.e. false).

Examples:

```
(oddp 0)
 => ()
(oddp 1)
 => t
(oddp 2)
 => ()
```

plusp NUMBER

Return t (i.e. true) if NUMBER is greater than zero otherwise nil (i.e. false).

Examples:

```
(plusp 0)
 => ()
(plusp 3)
 => t
(plusp -5.6)
 => ()
```

minusp **NUMBER**

Return t (i.e. true) if NUMBER is less than zero otherwise nil (i.e. false).

Examples:

```
(minusp 0)
 => ()
(minusp 3)
 => ()
(minusp -5.6)
 => t
```

atom OBJECT

Return t (i.e. true) if OBJECT is a number, symbol or character otherwise nil (i.e. false).

Examples:

```
(atom 3)
 => t
(atom 'jim)
 => t
(atom 4.5)
 => t
(atom '(a b))
 => ()
(atom #(a b c))
 => ()
```

listp OBJECT

Return t (i.e. true) if OBJECT is a cons or nil otherwise nil (i.e. false).Note - does not test whether a cons is a true list or a dotted list.

Examples:

```
(listp 3)
 => ()
(listp '(4))
 => t
(listp ())
 => t
(listp '(a b c))
 => t
```

true-listp **OBJECT**

Return t (i.e. true) if OBJECT is a true list i.e. repeatedly taking the cdr of OBJECT until a non-cons is found ends up with nil. otherwise nil (i.e. false).

Examples:

```
(true-listp 3)
 => ()
(true-listp ())
 => t
(true-listp '(3 4))
 => t
(true-listp '(3 4 . 5))
 => ()
```

Predicates

arrayp
atom
characterp
consp
constantp
evenp
integerp
keywordp
listp
minusp
null
numberp
oddp
plusp
stringp
structure-p
symbolp
true-listp
zerop

code-char **NUMBER**

Returns character with ASCII code NUMBER.

Examples:

```
(code-char 65)
 => #\A
(code-char 99)
 => #\c
```

char-code **CHARACTER**

Returns the ASCII code of CHARACTER.

Examples:

```
(char-code #\A)
 => 65
(char-code #\c)
 => 99
```

ASCII code

char-code
code-char

cascade-main-window

Cascade editor windows. Returns nil

move-main-window **RECTANGLE**

Reposition main editing window to position indicated by RECTANGLE.

show-main-window SHOW-WINDOW-CONST

Show main editor window according to SHOW-WINDOW-CONST which must be a show window style windows constant.. Returns nil.

open-file **FILENAME**

Open a new editor window for file FILENAME, unless FILENAME is already open, in which case bring its window to the front. Returns nil

set-cursor-color COLOR

Use to change color of editor cursor.

Editor commands

cascade-main-window
move-main-window
open-file
set-cursor-color
show-main-window

function EXPR

EXPR is unevaluated. EXPR can be symbol that is the name of the function or it can be a lambda expression. If EXPR is a symbol it returns the function value of that symbol. If it is a lambda expression it forms the closure of that expression. A "bracketless" abbreviation for function is #' i.e. #'name is equivalent to (function name), and the lisp system uses this abbreviation when writing out function expressions.

Examples:

```
#'car
 => <Builtin: car>
#'(lambda (x) (* x x))
 => <closure: (x) : #E((* x x))>
```

defun **NAME LAMBDA-LIST &rest STATEMENTS**

Creates a function closure with argument list LAMBDA-LIST and body STATEMENTS, and makes it the function value of the symbol NAME.

Effectively defines a new function called NAME which when called with arguments, binds those arguments to the arguments named in LAMBDA-LIST, executes STATEMENTS and then returns the value returned by the last of the STATEMENTS, or nil if there weren't any.

Examples:

```
(defun square (x) (* x x))
 => square
(square 12)
 => 144
```

macro-of-function **FUNCTION**

Make a macro out of FUNCTION.

Example:

```
(macro-of-function #'(lambda (x) `(cons ,x ,x)))
 => <Macro: <closure: (x) : #E(`(cons ,x ,x))>>
```

defmacro **NAME ARG-LIST &rest STATEMENTS**

Defines a new macro called NAME, that takes arguments given in ARG-LIST. When a list is evaluated where NAME is the first element, the rest of the elements in the list, without being evaluated first, are bound to the arguments specified in ARG-LIST, the STATEMENTS are executed, and then the result returned from the last of the STATEMENTS is evaluated in the lexical scope of the macro call.

Note the use of backquote in the example.

Examples:

```
(defmacro sum-pair (x) `(+ ,(first x) ,(second x)))
 => sum-pair
(sum-pair (2 3))
 => 5
```

Functions and Macros

defmacro
defun
function
macro-of-function

error **MESSAGE &optional DATA**

Signal an error, with MESSAGE and relevant DATA.

Error functions

error

gc

Performs garbage collection. Note that garbage collection is performed automatically and does not normally need to be explicitly requested.

gc-limit

Number of memory blocks (64k segments) used that triggers garbage collection. Can be used with setf. Minimum value 4, default = 16.

16 blocks = 1 megabyte of memory. This is the amount of memory used by Lisp objects, and does not include memory used by the editor or directly by the Lisp system.

Windows Enhanced mode allows the use of virtual memory which is disk storage pretending to be physical RAM. Long before Windows fails to allocate memory, physical RAM will be used up and Apteryx Lisp will be thrashing the hard disk. To avoid this, set a value for gc-limit that reflects available RAM, or the amount of RAM you wish to devote to Apteryx Lisp.

with-continuous-gc **&rest STATEMENTS**

Execute STATEMENTS in order, performing a garbage collection at every point where the need to do garbage collection is normally checked. This function is purely an aid to debugging transient crashes of Apteryx Lisp that might be caused by bugs in the garbage collector that arise in certain situations.

Garbage Collection

gc
gc-limit
with-continuous-gc

open **NAME &key :direction DIRECTION**

Opens file called NAME for input or output depending on direction which has to take one of the values :input or :output, with the default being :input. Returns the opened file handle.

close **FILE**

Closes file FILE. Returns nil.

eofp **&optional INPUT-FILE**

Return t (i.e. true) if INPUT-FILE has reached end of file otherwise nil (i.e. false). Default value of INPUT-FILE is *standard-input*

read **&optional INPUT-FILE**

Reads an lisp object in from INPUT-FILE and returns it. Default value of INPUT-FILE is *standard-input*

read-line **&optional INPUT-FILE**

Reads a line in from INPUT-FILE and returns it as a string (without the end-of-line character). Default value of INPUT-FILE is *standard-input*

terpri **&optional OUTPUT-FILE**

Write an end-of-line character out to OUTPUT-FILE. Returns nil. Default value of OUTPUT-FILE is *standard-output*.

write-byte **NUMBER &optional OUTPUT-FILE**

Writes byte NUMBER out to OUTPUT-FILE. Using princ and code-char will not have the desired effect because writing out the end-of-line character (= ASCII 10) results in writing out characters 13 and 10, because that is how MS-DOS represents an end of line. Returns nil. Default value of OUTPUT-FILE is *standard-output*.

print **OBJECT &optional OUTPUT-FILE**

Prints standard representation of OBJECT followed by an end-of-line to OUTPUT-FILE. Returns OBJECT. Default value of OUTPUT-FILE is *standard-output*.

prin1 **OBJECT &optional OUTPUT-FILE**

Prints standard representation of OBJECT to OUTPUT-FILE. Returns OBJECT. Default value of OUTPUT-FILE is *standard-output*.

princ **OBJECT &optional OUTPUT-FILE**

Prints standard representation of OBJECT to OUTPUT-FILE, unless OBJECT is a character or string, in which case just the character or characters of the string are printed out. Returns OBJECT. Default value of OUTPUT-FILE is *standard-output*.

format **OUTPUT-FILE FORMAT-STRING &rest OBJECTS**

Prints OBJECTS formatted as directed by FORMAT-STRING to OUTPUT-FILE. FORMAT-STRING is a string. Character sequences in FORMAT-STRING starting with ~ are interpreted as special commands, other characters are printed out as is. The ~ command sequences are -

~% : print end-of-line

~~ : print character ~

~A : print next object using princ

~S : print next object using prin1

If t is specified for OUTPUT-FILE, then *standard-output* is used.

format returns nil.

File input and output

close
eofp
format
open
prin1
princ
print
read
read-line
terpri
write-byte

struct-fields **(STRUCT-NAME)**

Return the list of field names for STRUCTURE-NAME.

Examples:

```
(defstruct address number street suburb city country)
 => address
(struct-fields 'address)
 => (number street suburb city country)
```

struct-ref **STRUCT INDEX**

Return field number INDEX of STRUCT. For a structure of size n, the index goes from 0 to n-1.

Can be used with setf.

Examples:

```
(defstruct word letters part-of-speech)
 => word
(struct-ref (make-word :letters "box" :part-of-speech 'noun) 1)
 => noun
```

defstruct **STRUCT-NAME &rest FIELD-NAMES**

defstruct is a special form - its arguments are not evaluated. All the arguments must be symbols.

It defines a new struct type of type STRUCT-NAME and returns STRUCT-NAME.

A struct is a data object consisting of a fixed number of components, that are referred to by the FIELD-NAMES. When the a struct type is defined using defstruct, a number of functions associated with the struct type are defined. These are a constructor function, accessor functions for the fields and a predicate function used to check if an object is of that struct type.

The names of these functions are derived from STRUCT-NAME and the FIELD-NAME. For example, defining a struct using

Example:

(defstruct person age sex)

=> person

defines a struct type called person with fields age and sex.

The constructor function would be called make-person. It would take keyworded arguments where the keywords had the same print-names as the field names.

Examples:

(progn (setq me (make-person :age 30 :sex 'male)))

=> #S(person age 30 sex male)

me

=> #S(person age 30 sex male)

The above example might represent a male person aged 30.

The accessor functions for the struct type person will be person-age and person-sex, and these can be used with setf. Fields of a struct can also be referred to with the help of with-struct.

Examples:

(person-age me)

=> 30

(setf (person-sex me) 'female)

=> female

(person-sex me)

=> female

The predicate will be called person-p. It returns t if its argument is a struct of type person, otherwise nil.

Examples:

(person-p me)

=> t

(person-p 'you)

=> ()

with-struct **(STRUCT-TYPE EXPR) &rest STATEMENTS**

Special form used to access the fields of a struct. EXPR must evaluate to a struct of type STRUCT-TYPE. All the STATEMENTS in the scope of the with-struct can refer to the fields of the struct resulting from EXPR using lexical variables whose names are the names of the fields of STRUCT-TYPE.

Example:

```
(with-struct (person me) (setf age 29) (list age sex))

=> (29 female)
```

This example follows on from the examples for defstruct.

Structure functions

defstruct
struct-fields
struct-ref
with-struct

macroexpand-1 EXPRESSION

If EXPRESSION is a macro call, return the result of expanding the macro, otherwise return EXPRESSION unchanged.

Examples:

```
(defmacro call-self (arg &rest args) `(,arg ',arg ,@args))
 => call-self
(macroexpand-1 '(call-self jim 3))
 => (jim 'jim 3)
```

macroexpand **EXPRESSION**

Repeatedly expand EXPRESSION as a macro, until a result is returned that is not a macro.

Examples:

```
(defmacro tom (arg) `(jim ,arg))
 => tom
(defmacro jim (arg) `(fred ,arg))
 => jim
(macroexpand '(tom 34))
 => (fred 34)
```

full-macroexpand **EXPRESSION**

Repeatedly expand EXPRESSION as a macro call and all macro calls within it until it is completely expanded.It is a consequence of the rules of evaluation of lisp, that if all macro calls in an expression are for macros already defined, and eval does not appear in the expression, then all macro calls in the expression can be pre-expanded, and the expanded form of the expression will return the same result when evaluated as the original expression.

If the function eval is called in the expression, this will not invalidate the macro expansion, but the argument to eval will not have its macro calls expanded because its value cannot be known until run-time.

Examples:

```
(defmacro jimmy (arg) `(tomas ,arg))
 => jimmy
(defmacro tomas (arg) `(fred ,arg))
 => tomas
(full-macroexpand '(jim (tomas 34)))
 => (fred (fred 34))
(full-macroexpand '(dotimes (i (jimmy 4)) (+ 1 (tomas 2))))
 => (dotimes (i (fred 4)) (+ 1 (fred 2)))
```

Macro expansion

full-macroexpand
macroexpand
macroexpand-1

array-to-code ARRAY

Converts ARRAY into equivalent compiled code object. Array elements come in pairs - the first of a pair is the symbol naming the opcode, and the second is the argument to the opcode.

the-compiler

If the variable *auto-compile* is set to non-nil, then the value of (the-compiler) is used to compile all expressions before evaluation. The initial value is nil, and if equal to nil expressions are compiled using the full-macroexpand function.

Loading the file "compile.lsc" defines a suitable value for (the-compiler) which compiles expressions into compiled code objects.

Can be used with setf.

compile-and-load **SOURCE-FILE COMPILED-FILE**

Loads SOURCE-FILE by compiling and then evaluating expressions, while writing compiled expressions out to COMPILED-FILE. Loading COMPILED-FILE using load should then have equivalent effect to loading SOURCE-FILE. Typically SOURCE-FILE has a .lsp extension, and COMPILED-FILE is a file of the same name but with a .lsc extension.

This function is only defined if the "compile.lsc" file has been loaded.

Compilation functions

point X Y

Construct a point with coordinates X and Y. Each coordinate must be a 16 bit signed integer i.e. in the range -32768 to 32767.

Example:

```
(point -32 45)
 => #<Point: -32 45>
```

point-x **POINT**

Return the x coordinate of the point POINT.

Example:

```
(point-x (point 32 43))
 => 32
```

point-y **POINT**

Return the y coordinate of the point POINT.

Example:

```
(point-y (point 32 43))
 => 43
```

Point functions

point
point-x
point-y

extent X Y

Return an extent with width and height X and Y. X and Y must be 16-bit signed integers, i.e. in the range -32768 to 32767.

Example:

```
(extent 34 56)
 => #<Extent: 34 56>
```

extent-x EXTENT

Return the x component (i.e. the width) of the extent EXTENT.

Example:

```
(extent-x (extent 34 56))
 => 34
```

extent-y **EXTENT**

Return the y component (i.e. the width) of the extent EXTENT.

Example:

```
(extent-y (extent 34 56))
 => 56
```

text-extent STRING

Return the extent of STRING, if it were to be output in the current device context.

Extent functions

extent
extent-x
extent-y
text-extent

rect **POINT1 POINT2**

Return the rectangle with opposite corners POINT1 and POINT2.

Example:

```
(rect (point 0 0) (point 32 45))
 => #<Rect: top-left:(0, 0), bottom-right:(32, 45)>
```

rect-top **RECTANGLE**

Return the Y coordinate of the top side of RECTANGLE.

Example:

```
(rect-top (rect (point 0 15) (point 100 200)))
 => 15
```

rect-left **RECTANGLE**

Return the X coordinate of the left-hand side of RECTANGLE.

Example:

```
(rect-left (rect (point 0 15) (point 100 200)))
 => 0
```

rect-bottom **RECTANGLE**

Return the Y coordinate of the bottom side of RECTANGLE.

Example:

```
(rect-bottom (rect (point 0 15) (point 100 200)))
 => 200
```

rect-right **RECTANGLE**

Return the X coordinate of the right-hand side of RECTANGLE.

Example:

```
(rect-right (rect (point 0 15) (point 100 200)))
 => 100
```

rect-width **RECTANGLE**

Return the width of RECTANGLE.

Example:

```
(rect-width (rect (point 0 15) (point 100 200)))
 => 100
```

rect-height **RECTANGLE**

Return the height of RECTANGLE.

Example:

```
(rect-height (rect (point 0 15) (point 100 200)))
 => 185
```

copy-rect **RECTANGLE**

Return a copy of RECTANGLE, i.e. equal to but not eq to RECTANGLE.

Example:

```
(copy-rect (rect (point 0 0) (point 10 10)))
 => #<Rect: top-left:(0, 0), bottom-right:(10, 10)>
```

intersect-rect **RECTANGLE1 RECTANGLE2**

Return the rectangle which is the intersection of RECTANGLE1 and RECTANGLE2.

Example:

```
(intersect-rect (rect (point 0 0) (point 20 30)) (rect (point 10 10) (point 40 50)))
 => #<Rect: top-left:(10, 10), bottom-right:(20, 30)>
```

is-rect-empty **RECTANGLE**

Return t (i.e. true) if RECTANGLE is empty. otherwise nil (i.e. false).

Examples:

```
(is-rect-empty (rect (point 0 0) (point 10 10)))
 => ()
(is-rect-empty (rect (point 0 0) (point 0 10)))
 => t
```

point-in-rect **POINT RECTANGLE**

Return t (i.e. true) if POINT is in RECTANGLE otherwise nil (i.e. false).

Examples:

```
(point-in-rect (point 10 10) (rect (point 0 0) (point 20 20)))
 => t
(point-in-rect (point 30 30) (rect (point 0 0) (point 20 20)))
 => ()
```

Rectangle functions

copy-rect
intersect-rect
is-rect-empty
point-in-rect
rect
rect-bottom
rect-height
rect-left
rect-right
rect-top
rect-width

rgb **RED-INTENSITY GREEN-INTENSITY BLUE-INTENSITY**

Return a color with the specified intensities. Each intensity must be a number from 0 to 255 (i.e. an unsigned byte).

Example:

```
(rgb 0 128 255)
 => #<Color: 0 128 255>
```

red-component COLOR

Return the red component of COLOR as a number from 0 to 255.

Examples:

```
(red-component white)
 => 255
(red-component yellow)
 => 255
```

green-component COLOR

Return the green component of COLOR as a number from 0 to 255.

Examples:

```
(green-component white)
 => 255
(green-component yellow)
 => 255
```

blue-component COLOR

Return the blue component of COLOR as a number from 0 to 255.

Examples:

```
(blue-component white)
 => 255
(blue-component yellow)
 => 0
```

background-color

Return the current background-color of the current device context.

Can be used with setf.

nearest-color COLOR

Return the nearest color to COLOR that is available in the current palette of the current device context.

text-color

Return the current color for text output using textout for the current device context.

Can be used with setf.

system-color **COLOR-TYPE**

Return specified system color. COLOR-TYPE must be a system colour windows constant..

Can be used with setf.

Color functions

background-color
blue-component
green-component
nearest-color
red-component
rgb
system-color
text-color

client-to-screen WINDOW POINT

Convert POINT with x and y coordinates given relative to the origin (top left hand corner) of the WINDOW's client area to a point with x and y coordinates given relative the the top left hand corner of the screen.

screen-to-client **WINDOW POINT**

Convert POINT with x and y coordinates given relative to the origin (top left hand corner) of the screen to a point with x and y coordinates given relative the the top left hand corner of the WINDOW's client area.

Conversion functions

client-to-screen
screen-to-client

exit-windows

Exit Microsoft Windows (TM) (without asking running applications if its OK to exit, so this function is a bit dangerous).

Exiting windows

exit-windows

create-solid-brush COLOR

Return a solid brush of the specified COLOR.

Example:

```
(create-solid-brush green)
 => #<Brush: solid, #<Color: 0 255 0>>
```

create-hatch-brush HATCH-STYLE COLOR

Return a hatched brush of the specified HATCH-STYLE, which must be a brush hatch style windows constant., and of the specified COLOR.

Example:

```
(create-hatch-brush hs_vertical red)
 => #<Brush: hatched, #<Color: 255 0 0>>
```

Brush functions

create-hatch-brush
create-solid-brush

create-pen **STYLE THICKNESS COLOR**

Return a pen of the specified STYLE, which must be a pen style windows constant., and of the specified THICKNESS and COLOR.

Example:

```
(create-pen ps_solid 5 red)
 => #<Pen: solid, thickness: (5, 5), #<Color: 255 0 0>>
```

Pen functions

create-pen

create-font **NAME HEIGHT**

Create a font of the given name and height

Example:

```
(create-font "Times" 12)
 => #<Font: "Times", 12>
```

Font functions

create-font

with-dc **WINDOW &rest EXPRESSIONS**

A control structure. Evaluate WINDOW, and set the current device context to be a device context for that window. Evaluate each EXPRESSION in turn and return the value of the last one. Finally, reset the current device context to be the device context that it was previously.

with-select **(&rest DRAWING-OBJECTS) &rest EXPRESSIONS**

A control structure. Evaluate each DRAWING-OBJECT, and select it into the current device context. Then evaluate each EXPRESSION, and finally un-select the selected drawing objects, replacing them with the originally selected objects. A drawing object can be a pen, a brush, or a font. Returns the result of the last EXPRESSION.

with-selected-objects **DRAWING-OBJECTS &rest EXPRESSIONS**

A control structure. Evaluate DRAWING-OBJECTs to get a list of drawing objects, and select it into the current device context. Then evaluate each EXPRESSION, and finally un-select the selected drawing objects, replacing them with the originally selected objects. A drawing object can be a pen, a brush, or a font. Returns the result of the last EXPRESSION.

print-window WINDOW

Print out contents of window on currently selected printer

draw-rect **RECTANGLE**

Draw RECTANGLE in the current device context with the currently selected pen and brush.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

draw-arc **RECTANGLE START-POINT END-POINT**

Draw an arc of the ellipse bounded by RECTANGLE in the current device context starting at START-POINT and ending at END-POINT, with the currently selected pen. The start and end points do not have to be on the actual ellipse - the actual points will be on the lines joining the specified points and the centre of the ellipse.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

draw-focus-rect RECTANGLE

Draw a focus-style RECTANGLE in the current device context using XOR (which is self-reversing). Returns t.

draw-ellipse **RECTANGLE**

Draw an ellipse whose sides are bounded by RECTANGLE in the current device context with the currently selected pen and brush.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

clip-box

Return the smallest rectangle that bounds the clipping boundary of the current device context.

current-position

Return the point that is the current position in the current device context(as used bymove-to and line-to.

dc-origin

Return the current origin of the current device context.

pixel **POINT**

Return the color of POINT in the current device context.

Can be used with setf.

viewport-origin

Return the origin of the viewport of the current device context.

Can be used with setf.

viewport-extent

Return the extent of the viewport of the current device context.

Can be used with setf.

window-origin

Return origin of window in viewport of the current device context.

Can be used with setf.

window-extent

Return extent of window in viewport of the current device context.

Can be used with setf.

line-to **POINT**

Draw a line from current position in the current device context to POINT.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

move-to **POINT**

Move current position in the current device context to POINT (without drawing anything.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

draw-pie **RECTANGLE START-POINT END-POINT**

Draw a pie section of the ellipse bounded by RECTANGLE in the current device context starting and ending at the lines from the origin of the ellipse to START-POINT and END-POINT, with the currently selected pen.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

textout **STRING POINT**

Output STRING to the current device context starting at POINT, using the currently selected font and current text color.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

flood-fill **POINT COLOR**

"Color in" the area in the current device context bounded by COLOR, that POINT is in, using the currently selected brush.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

draw-frame-rect **RECTANGLE BRUSH**

Draw a line around RECTANGLE using BRUSH in the current device context.Return t (i.e. true) if it was drawn successfully otherwise nil (i.e. false).

set-raster-op **BINARY-RASTER-OP**

Set the binary raster operation windows constant. that defines how new colors are combined with existing colors in drawing operations on the current device context.

repaint WINDOW

Mark WINDOW to be repainted (using its repaint function). Returns nil

invert-rect **RECTANGLE**

Invert the colors of RECTANGLE in the current device context. Returns nil

Graphics functions

clip-box
current-position
dc-origin
draw-arc
draw-ellipse
draw-focus-rect
draw-frame-rect
draw-pie
draw-rect
flood-fill
invert-rect
line-to
move-to
pixel
print-window
repaint
set-raster-op
textout
viewport-extent
viewport-origin
window-extent
window-origin
with-dc
with-select
with-selected-objects

make-window **CAPTION-STRING &key DATA RECT PAINTER**

Make a window. CAPTION-STRING is the caption that goes on the title bar. DATA is the window data, (accessed using window-data), RECT is the rectangle that defines the window's initial size and PAINTER is the function that repaints the window (accessible using window-painter and taking as arguments the window and the rectangle that needs repainting). Returns handle to the window.

close-window **WINDOW**

Close WINDOW. Returns handle to closed window.

window-data WINDOW

Data associated with window. Typically this would be data that the window displayed.

Can be used with setf.

window-painter WINDOW

The function that is called when WINDOW needs repainting, for example when part of the window that was hidden is revealed. The function must take two arguments - WINDOW and the rectangle being repainted. Typically the function would repaint the area specified by the rectangle, although over-painting is OK, e.g. the painter function can ignore the rectangle all together and just repaint the whole window. The function does not need to call with-dc as the repainting automatically sets the current device context to be a device context for WINDOW for the duration of the execution of the repaint function.

Can be used with setf.

client-rect WINDOW

Return the client rectangle of WINDOW, i.e. that part of the window excluding the title bar, menu bar and scroll bars that may be drawn on.

window-text WINDOW

The caption on the window.

Can be used with setf.

window-rect WINDOW

The rectangle that specifies the position of WINDOW on the screen.

is-iconic **WINDOW**

Return t (i.e. true) if WINDOW is iconic otherwise nil (i.e. false).

is-window-enabled **WINDOW**

Return t (i.e. true) if WINDOW is enabled otherwise nil (i.e. false).

is-window-visible **WINDOW**

Return t (i.e. true) if WINDOW is visible otherwise nil (i.e. false).

is-zoomed **WINDOW**

Return t (i.e. true) if WINDOW is zoomed (i.e. maximized) otherwise nil (i.e. false).

move-window **WINDOW RECTANGLE REPAINT**

Move WINDOW to position on screen specified by RECTANGLE and repaint if REPAINT is non-nil. Returns nil.

bring-window-to-top **WINDOW**

Bring WINDOW to top of other windows. Returns nil.

open-icon **WINDOW**

Open up iconized WINDOW.Return t (i.e. true) if WINDOW was successfully restored.
otherwise nil (i.e. false).

point-visible **POINT**

Return t (i.e. true) if POINT is visible in the current device context otherwise nil (i.e. false).

rect-visible **RECTANGLE**

Return t (i.e. true) if any part of RECTANGLE is visible in the current device context otherwise nil (i.e. false).

Window functions

bring-window-to-top
client-rect
close-window
is-iconic
is-window-enabled
is-window-visible
is-zoomed
make-window
move-window
open-icon
point-visible
rect-visible
window-data
window-painter
window-rect
window-text

current-time

Returns the elapsed time since the system was rebooted in milliseconds.

double-click-time

The double-click threshold in milliseconds, i.e. the maximum time between two mouse clicks that will be considered a double-click (applies to whole of Microsoft Windows (TM)).

Can be used with setf.

Time functions

current-time
double-click-time

cursor-pos

The position of the cursor in screen coordinates.

Can be used with setf.

Cursor functions

cursor-pos

run-program **COMMAND :show STYLE**

Execute COMMAND (given as a string). Command is the name of either a Windows or Dos executable or batch file, possibly followed by some parameters. If the directory of the executable or batch file is not given it will be looked for in the directories specified in the PATH environment variable.

STYLE is a show window style windows constant., and determines how the window for the command is initially shown.

The command is executed asynchronously, i.e. control is returned to the caller as soon as the command has been started. It is not possible to wait for execution of the command to finish.

The return value is an integer giving information about success or otherwise of the attempt to execute the command.

set-current-directory **DIRECTORY**

Sets current DOS directory to DIRECTORY. Don't forget that to write a backslash in a string you have to write it twice.

current-directory **&optional DRIVE**

Returns current DOS directory. DRIVE is the drive number, 0=current,1=A,2=B etc, and defaults to the current drive.

Example:

```
(current-directory 3)
 => "C:\\LISP"
```

mem

Prints to the output window a report on memory usage, in particular the number of blocks (64k segments) used by Lisp objects. This does not count fixed overhead or editor buffers.

sp

Returns value of stack pointer.

dos-environment

Returns the DOS environment string.

free-heap-space

Returns the amount of free memory in bytes.

keyboard-code-page

Returns the number of the current keyboard code page.

profile-int **APPLICATION-NAME KEY-STRING DEFAULT**

Retrieves an integer value from the WIN.INI file by APPLICATION-NAME, KEY-STRING and DEFAULT if not found.

profile-string **APPLICATION-NAME KEY-STRING DEFAULT**

Retrieves a string value from the WIN.INI file by APPLICATION-NAME, KEY-STRING and DEFAULT if not found.

Can be used with setf.

When used with setf, the DEFAULT argument must be omitted.

windows-directory

Returns Windows directory pathname.

system-directory

Returns Windows system subdirectory pathname.

tick-count

Returns time since system started in milliseconds.

lo-word **INTEGER**

Returns low 16 bits of 32 bit INTEGER.

Example:

```
(lo-word 65539)
 => 3
```

hi-word INTEGER

Returns high 16 bits of 32 bit INTEGER.

Example:

```
(hi-word 65539)
 => 1
```

make-long **LOW HIGH**

Returns 32 bit integer from two 16 bit integers LOW and HIGH.

Example:

```
(make-long 3 1)
 => 65539
```

System functions

current-directory
dos-environment
free-heap-space
hi-word
keyboard-code-page
lo-word
make-long
mem
profile-int
profile-string
run-program
set-current-directory
sp
system-directory
tick-count
windows-directory

swap-mouse-buttons

Swap left and right mouse buttons.

restore-mouse-buttons

Restore left and right mouse buttons to original setting.

Mouse functions

restore-mouse-buttons
swap-mouse-buttons

beep

Beep.

Sound functions

beep

Builtin functions and macros

Index