TEXTMODULES

TEXTMODULES converts source code files from File Manager format to Common Lisp style plain text and back again. When exporting to plain text, a small number of File Manager coms types are supported. When importing from a plain text file, several convenience features are available including comment upgrade and conversion of specific named defmacros into defdefiners.

Note: The Text File Translator changes source code format only; this is **not** an Interlisp to Common Lisp translator.

All symbols described in this section are in the TEXTMODULES package, nicknamed TM.

This section describes the load and make processes, and the static format of text files and their File Manager counterparts. The File Manager counterparts are discussed in increasing detail until their programmability is covered.

Overview

The Text File Translator supports the development of portable Common Lisp source code in the Lisp Environment. It brings portable Common Lisp sources into the File Manager without losing any of their contents. It also makes new text files based on the File Manager's "filecoms."

The original file's function and ordering are retained, but exact formatting is not. The pretty printer causes all comments and expressions on the text file to be uniformly formatted.

Exporting a source file into text and back again will lose grouping of definitions under their coms.

Installation

Load TEXTMODULES.DFASL from the library.

Dependencies

Special support for editing and printing of comments is required. This are provided by the SEDIT-COMMONLISP file. Some caveats on the editing of presentations are mentioned below.

The support file is automatically loaded by the TEXTMODULES file. SEDIT-COMMONLISP cannot operate without TEXTMODULES and must be loaded by it or after it.

File Manager source files created with **load-textmodule** depend on having TEXTMODULES and SEDIT-COMMONLISP loaded.

Programmer's Interface

`load-textmodule` *pathname* &key *module install package upgrade-comment-length join-comments convert-loaded-files defdefiner-macros*

[Function]

Like **lisp:load**; the file indicated by *pathname* is loaded, but in addition filecoms ar created and other information is stored for the File Manager. Key arguments are described below.

(See below under **Text File Format** for a description of the format of text files which can be read by **load-textmodule**).

Local bindings of reader affecting variables are established and set to Common Lisp defaults, except for the readtable.

A special readtable is used which creates internal representations for objects normally lost during reading (see below under **Presentation types**).

If there are some simple forms to set up the read environment at the front of the file, they are recognized and moved into a newly created makefile environment (see below under **Makefile Environment** for a complete description of this).

Each form is read from the file (one at a time). If the form is recognized a description of it is given to the File Manager and its definition is installed. If the form is not recognized it is wrapped in a "top level form" filecom and then installed by stripping presentation objects and evaluating.

defun in a **let** at top-level is treated like any top-level form. Such forms should be edited directly in the filecoms. Not doing this can have curious consequences, since calling **ed** on the function name will not modify the definition in the **let** (which remains in the `FILECOMS` as a top level form).

No forms after the read environment forms should change the reader's environment.

When the file has been completely read its content description is given to the File Manager. Also added to the content description are properties declaring its `il:filetype` as `:compile-file` and makefile-environment as that of the text file (whether given by setting forms at the front of the file or by default).

Several key options are available:

module A string or symbol used to create the symbol used as the File Manager's name of this module. Strings have their case preserved. Symbols have their name strings taken. Defaults to the uppercased root name of the path.

install `T` or `NIL`. Indicates whether the definitions in the file should be installed in the running system. Any package setup makes it mandatory to install the definitions in a source file; e. g. since `:INSTALL NIL` means forms in the file are not evaluated, any `IN-PACKAGE` form would not be evaluated and the file would be read in the wrong package. This can sometimes be worked around using the `:PACKAGE` argument.

package A package name or package, defaults to `USER`. This is the package the file will be read into.

upgrade-comment-length A number or NIL. Defaults to the value of *upgrade-comment-length* (which defaults to 40). The length, in characters, at which single semicolon comments are upgraded to double semicolon comments.

join-comments T or NIL. Defaults to the value of *join-comments* (which defaults to T). Causes comments of the same level in the coms to be joined together. This makes for more efficient editor operation, but loses all formatting inside of comments; e.g. inter-comment line breaks are not preserved.

convert-loaded-files T, :QUERY or NIL. Defaults to the value of *convert-loaded-files* (which defaults to :QUERY). If a REQUIRE or LOAD statement is noticed at top level a recursive call to LOAD-TEXTMODULE will be made. With :QUERY turned on the user is first prompted. If the pathname specified in the LOAD or REQUIRE is computed based on variables in the file being loaded :INSTALL must be true. Complex systems that contain special loading functions will not be handled by this mechanism.

defdefiner-macros A list of defmacro names. Defaults to the value of *defdefiner-macros* (which defaults to NIL). If a top-level defmacro is found whose name is on this list, the defmacro will be translated into an IL:FUNCTIONS defdefiner form. The defdefiner form then creates a macro that builds definers. Definers are the basic definition units maintained by the File Manager. DEFUN is itself a defdefiner macro. A particular DEFUN form is a definer for the named function (see the *Lisp Release Notes*, 4. Changes to Interlisp-D in Lyric/Medley, Section 17.8.2 Defining New File Manager Types, for more information on the defdefiner form).

Warning: Names on this list must be in the correct package, i.e. the one the file will be read in. A typical way to use this feature is:

- Examine the text source file for DEFMACRO forms that are used to create defdefiners.
- Make the package which the text file's IN-PACKAGE expression will later find.
- Do LOAD-TEXTMODULES, giving the :DEFDEFINER-MACROS key argument a list of fully package qualified symbols naming the defdefiners contained in the file.

make-textmodule *module* &key *type pathname filecoms width* [Function]

The File Manager's description of the file *module* is used to create a text file. *module* may be provided as either a string or a symbol. A string's case will be preserved. A symbol's name string is used. Keyword arguments are described below.

(See below under **File manager description of contents** for a description of filecoms that can be written out by **make-textmodule**.)

Local bindings of printer affecting variables are established and set to Common Lisp defaults, except for the readtable.

The file's environment is written out, based on its makefile-environment property (see below under **File manager source file format** for ways of expressing the environment).

The specially made description of the file's contents (from the File Manager) is iterated over to write out each form in the file.

Several key options are availible:

type A string, defaults to ".LISP". The file type extension to be used on the text file being written.

pathname A pathname, defaults to the module name merged with the extension and ***default-pathname-defaults***. The file which will contain the new text file.

filecoms A list of file commands may be supplied here. Defaults to the commands for the File Manager file named by module.

width A positive integer, defaults to 80. The width, in characters, of lines in the text file. Used by the prettyprinting routines for formatting.

Variables that Control Loading

join-comments [Variable]

`T` or `NIL`. Defaults to `T`. Causes comments of the same level in the file coms to be joined together. This makes for more efficient editor operation, but loses any formatting inside of comments, e.g. inter-comment line breaks are not preserved.

convert-loaded-files [Variable]

`NIL`, `:QUERY` or `T`. Defaults to `:QUERY`. Controls whether a `LOAD` or `REQUIRE` statement at top level in a loaded text file will cause the referred to file to be recursively load-textmodule'd. If the pathname specified in the `LOAD` or `REQUIRE` is computed based on variables in the file being loaded the `:INSTALL` argument to **load-textmodule** must be true.

upgrade-semicolon-comments [Variable]

`NIL` or a positive integer. Defaults to `40`. Controls whether and at what length (in characters) a single semicolon comment is upgraded to a double semicolon comment. `NIL` inhibits upgrading.

defdefiner-macros [Variable]

A list of defmacro names. Defaults to `NIL`. If a top-level defmacro is found by `LOAD-TEXTMODULES` whose name is on this list, the defmacro will be translated into an `IL:FUNCTIONS` defdefiner form. The defdefiner form creates a macro that builds definers. Definers are the basic definition units maintained by the File Manager. `DEFUN` is itself a defdefiner macro. A particular `DEFUN` form is a definer for the named defun (see the *Lisp Release Notes*, 4. Changes to Interlisp-D in Lyric/Medley, Section 17.8.2 Defining New File Manager Types, for more information on the defdefiner form).

Warning: Names on this list must be in the correct package, i.e. the one the file will be read in. A typical way to use this feature is:

- Examine the text source file for DEFMACRO forms that are used to create defdefiners.
- Make the package which the file's IN-PACKAGE expression will later find.
- Do LOAD-TEXTMODULES giving the :DEFDEFINER-MACROS keyword argument a list of fully package qualified symbols naming the defdefiners contained in the file.

Text File Format

TextModules creates and understands the format of portable pure Common Lisp text files with very simple and constrained package setup information. The overall form of these files is described here as a guide to what sort of files may be imported.

An EMACS style mode line comment may optionally be present as the file's first item. It corresponds to the makefile-environment in the file manager.

```
; -*- Mode: LISP; Package: (FOO GLOBAL 1000); Base:10 -*-
```

mode	For some versions of the EMACs editor this will declare the major mode, which arranges key to command bindings for LISP instead of documents.
package	Name, used packages and initial space for symbols.
base	Numeric "ibase"

The mode line is generated by TextModules and is provided purely as a convenience in transporting code to EMACS based environments. It has no effect on the File Manager. The makefile-environment is actually instated using expressions directly following the mode line.

The Common Lisp community has agreed that portable text files will use only one reader environment and hence not switch packages or alter the readtable partway through. TextModules assumes that the reader environment is set up by the seven (plus two) standard environment modifying forms. These forms are recognized by the TextModules parser only if they appear at the front of the file and in order (comments being ignored):

Put	**provide**
In	**in-package**
Seven	**shadow**
Extremely	**export**
Random	**require** (or **il:filesload**)
User	**use-package**
Interface	**import**
...	**shadowing-import**
...	**setf *read-base***
Commands	Contents of module

Portable files may optionally add ***read-base*** setting and **shadowing-import** expressions. Also, **il:filesload** may be used in place of **require**, when a Lisp file (not containing a **provide** form) must be loaded.

Any one of these forms is optional, but they must appear first in the file (and in order) to be parsed into a makefile-environment when **load-textmodule** is called.

Warning: this software applies heuristics to the package forms to make them independent of the package environment they are read in. These may not work and it is recommended that the makefile-environment be checked for correctness after load-textmodule brings in the file. Complex package setups will almost certainly not be handled correctly and should be created in a separate file which the main text file can REQUIRE.

The contents of a text file are a sequence of forms. Certain forms are understood by the File Manager and hence specially recognized. These recognized forms include:

Comments	which are translated into Interlisp style comments
Definers	such as defun or defvar
eval-when	which is translated into a File Manager eval-when
Read time conditionals	which may become "unread objects"

All other forms are considered *top level forms* and simply saved as is.

Definers hold onto presentations, e.g., read time conditionals, as well as comments. Comments and presentations are always available to be edited.

To see if something is a definer form, examine the property list of its name (like **defun**). Use the Exec's **pl** (print property list) command to look for the property **:definer-for**.

Note that only the above kinds of forms will be recognized by the TextModules parser on a portable Common Lisp file.

There are a few somewhat common problems that can arise when importing a text file. Chief among these are "bootstrapping definitions" and circularity.

Known Problems

Occasionally, when starting up a complex software system, it is useful to install a temporary definition until the mechanism required by the actual definition is in place. This can cause a definition by the same name to appear in more than one place in a text file. The Textmodules system will simply use the latter definition in the file, causing the next loading of it to fail for lack of the lost bootstrap definition. It is recommended that bootstrap definitions be made into "top level forms", e.g. a `DEFUN` can become a `(SETF (SYMBOL-FUNCTION <name>) ...)` form, `DEFPARAMETER` can become `(SETF <name> ...)`, etc.

Also, some styles of programming may encourage creation of circular structure. Textmodules must map over top level forms to install presentations that contain comments, etc. Circular structures can cause these routines to fill memory with list structure.

File Manager Source Files

Unlike standard Common Lisp, the File Manager is designed to keep all of a file's contents resident in memory as structure, rather than text. This scheme allows very fast update and editing of definitions. To maintain its own source files the File Manager keeps descriptions of the format (makefile-environment) and contents (filecoms) of a file.

The makefile-environment of the file is used to note the readtable and package the rest of the file to be read and printed in, and any file dependencies.

The filecoms maintain a description of the top-level defining forms in a file, like function and variable definitions. They also store plain top-level forms. Within any form there can appear lisp data, like vectors or "number in a radix." How these are read and printed are controlled by presentation types (described separately; see below).

It is important to separate the environment of the file from its contents because the File Manager (not TextModules) first reads all the forms in the file, and then evaluates them. Text based source files sometimes change the package as needed. This cannot work for the File Manager since the file's forms are all read and then executed, i.e. the package changes would not occur until after the entire file had been read, and forms after any IN-PACKAGE form would have been read incorrectly.

The File Manager first reads the makefile-environment forms in a well known environment (INTERLISP package, INTERLISP readtable) evaluates them to find the environment of the rest of the source file, then reads the rest of the source file in that environment. This is why the package environment setup forms are so carefully parsed out of text files being imported into the environment.

File Manager source files created by **load-textmodule** depend on both the TEXTMODULES & SEDIT-COMMONLISP modules. These must be loaded before source files created with them can be reloaded.

File Manager Source File Format

The makefile-environment of a "managed" source file is used to control both how the exported text file and managed source files are printed. It is kept in a property named **il:makefile-environment** on the symbol with the root name of the file (in the INTERLISP package). This property is automatically generated when a portable text file is imported. The property is itself a plist containing :readtable, :package and :base values. The readtable used is called "LISP-FILE", a readtable defined by the TextModules program (hence File Manager source files created by **load-textmodules** depend on TextModules). The part of the makefile-environment of main interest is the :package. It sets the package in which the exported text file is printed.

Three forms of the makefile-environment's :package property are recognized.

- a string or symbol naming a package
- a **defpackage** statement
- a **let** statement

A string or symbol is simply taken to name a package.

A **defpackage** statement will have its portable components translated into a **let** statement as described below.

A **let** used for the makefile-environment should bind ***package*** and contain some form of the standard seven package and module setup forms (See above under "Text file format"). It should finally return the altered value of *package*. For example:

```
(let ((*package* *package*))
  ...environment setup forms.
  *package*
)
```

The forms in this expression must be written in a standard, pre-existing package, such as USER or XCL-USER. This is to break the circularity of writing a package defining form in the package it defines.

The package defining expressions in the let should follow all of the rules for portable text files (See above under "Text file format"), e.g., they should appear in order.

File Manager Description of Contents

When a portable text file is imported its contents are parsed to produce the File Manager's filecoms (File Commands). File commands are a high-level way of viewing and controlling the ordering of definitions in a file. The following describes the filecoms produced when a text file is imported.

Forms on the file are either recognized as top level defining forms or wrapped in a "top level form" filecom. Several forms are recognized by TextModules itself and others can be added (see below under "Making new specifiers"). The constructs that recognize filecoms, both to export and import plain text, are called *specifiers*.

The following are placed in the filecoms based on the parsed contents of the text file:

(il:* *type string***)** Contains a comment string. *type* is a symbol of one, two, three or four semicolons, or a vertical bar. This handles top level single, double or triple semicolon comments, as well as balanced comments. When viewed in SEdit these display in real comment format, instead of the internal list representation.

(eval-when *when* . *filecoms*) Wrapper with an evaluation time and containing more filecoms.

definers All definers are recognized, e.g. DEFUN, DEFMACRO, DEFVAR, DEFPARAMETER, DEFSTRUCT, etc. The definer specifier also converts DEFMACRO forms on the ***defdefiner-macros*** list to defdefiners during loading, and vice versa on printed to a text file.

(il:p (top-level-form *form*)) Top level form wrapper with a macro that calls the presentation translator. This filecom contains expressions which were not recognized and must be evaluated at load time. This kind of filecom also handles top level occurances of conditional read and read time evaluations (hash comma and hash dot). The top-level-form specifier also looks for LOADed or REQUIREd files and, depending on the variable ***convert-loaded-files***, attempts to convert the loaded files as well.

When the file is loaded and before evaluating these forms any presentation objects in them are stripped out (as for comments) or installed (as for read time evaluations). This is done by the TOP-LEVEL-FORM macro, which dispatches to the translation functions for the particular presentation objects. i.e., this allows comments to appear anywhere in the forms and not affect evaluation.

The above coms are created when a text file is imported. There are also a few specifiers provided to export filecoms, but not create them on import. These are convenient for exporting typical File Manager files. They are:

(il:coms . *filecoms*) Used to group together definitions in a File Manager source file. The *filecoms* are dumped onto the text file in order. No information is placed on the resulting text file to preserve the grouping of the filecoms; if the exported text file is later imported the coms grouping will not reappear.

(il:vars . *descriptors*)	As described in the *IRM*. The *descriptors* are exported as **defparameter** forms. If the exported text file is later imported these **vars** coms will reappear under **variables** coms.
(il:initvars . *descriptors*)	As described in the *IRM*. The *descriptors* are exported as **defvar** forms. If the exported text file is later imported these **initvars** coms will reappear under **variables** coms.
(il:constants . *descriptors*)	As described in the *IRM*. The *descriptors* are exported as **defconstant** forms. If the exported text file is later imported these **constants** coms will reappear under **variables** coms.
(il:props . *descriptors*)	As described in the *IRM*. The *descriptors* are exported as **(setf (getf ...) ...)** forms. If the exported text file is later imported these **props** coms will reappear under **p** coms (top-level forms).
(il:prop *props* . *symbols*)	As described in the *IRM*. The *props* and *symbols* descriptors are used to generate forms for export, e.g. **(setf (getf 'foo 'bar) 21)**. If the exported text file is later imported these **prop** coms will reappear under **p** coms (top-level forms).
(il:files . *items*)	As described in the *IRM*. The *items* are used to generate forms for export, e.g. **(load** "**Foo.lisp**"**)**. All options except **noerror** are ignored, the latter will cause the **:if-does-not-exist nil** key argument to be included in the load expression. If the exported text file is later imported these **files** coms will reappear under **p** coms (top-level forms).

Making New Specifiers

Specifiers are the glue that relate forms on a plain text file to filecoms in a File Manager source file. They can be considered addenda to the filepkgtype mechanism of the File Manager.

specifiers [Variable]

A list of specifiers (its default contents are described below). New specifiers should be added to this list. This list is searched linearly; its order is significant mostly in that the default top-level form recognizer must always be last.

make-specifier &key *name filecom-p form-p add-form install-form print-filecom* [Function]

This function creates new specifiers for inclusion on the ***specifiers*** list. A specifier maps between the forms in a text file's contents and filecoms. It is the basis for importing and exporting top-level forms. Specifiers can be nested, as for EVAL-WHEN.

To do all of this a specifer contains functions that recognize forms of its kind on the text file and coms in filecoms, as well as functions that add the definition to the filecoms and install the definition as the one to be used at runtime. Finally, there is a function which prints a form onto a text file based on a com on the fileoms.

name A string naming the specifier.

filecom-p Predicate on FILECOM which returns true if it is the one used to represent this specifier in the filecoms of a managed file.

form-p Predicate on FORM, a form read from a text file being imported, which returns true if this is the specifier for the definition in FORM.

add-form Function of FORM and FILECOMS. FORM is a form read from a text file being imported, and which has already been confirmed with the *form-p* method. Should make the definition in FORM available (editable) in the programming environment (File Manager). It should make the definition editable and add a filecom for FORM to the FILECOMS description. It should return the new FILECOMS description. To *add-form* runs of subforms use **add-form** and **form-specifier** (see below).

Care should be used when making a definition editable. The simplest instance of this occurs when the FORM's definition is a definer. In this case its evaluation may be wrapped in a binding of **il:dfnflg** to **il:prop** to ensure that the definition form goes into the table of current definitions *without being evaluated*.

Adding a filecom to the FILECOMS should be done in a way that preserves ordering. The simplest way to do this is to append the new filecom to the end of the current FILECOMS.

install-form Function of a FORM which makes the definition in FORM the current one to be used in execution. If the defining mode flag indicates that the file is being loaded for editing only this function *will not be called* during loading of the form (il:dfnflg is set by the :install option to load-textmodule). To *install* runs of subforms use **install-form** and **form-specifier** (see below).

Care should be used when making a definition executable. The simplest instance of this occurs when the FORM's definition is a definer. In this case its evaluation may be wrapped in a binding of `il:dfnflg` to `t` to ensure that the definition form *is actually evaluated*.

print-filecom Function of `FILECOM` and `STREAM` which should generate a new line and pretty print a form, representing the `FILECOM`, onto the stream. To print runs of subforms use `print-filecom` and `filecom-specifier` (see below).

The semantics of the *add-form* and *install-form* methods remove some confusion between loading a definition into memory for editing (loading PROP) and installing that definition as the currently executable one (loading T). The *add-form* method makes the definition editable and the *install-form* makes it executable.

The following functions are used to handle subforms EG. In the eval-when specifier there are subforms that need to be parsed.

`form-specifier` *form* [Function]

Searches the current list of specifiers in an attempt to recognize *form* (as read from a text file being imported). Returns a specifier for *form*. If none is found a warning is signalled and a "do nothing" specifier is returned.

`filecom-specifier` *filecom* [Function]

Searches the current list of specifiers in an attempt to recognize *filecom* (as found in the filecoms of the managed file). Returns a specifier for *filecom*. If none is found a warning is signalled and a "do nothing" specifier is returned.

`add-form` *form filecoms &optional specifier* [Function]

Adds the *form* to the *filecoms* description based on the add method in the *specifier*. If *specifier* is not provided `form-specifier` will be used to get it from *form*. Returns the new filecoms. `nil` is used as an empty contents description.

`install-form` *form &optional specifier* [Function]

If the current definition mode `(il:dfnflg)` allows it, installs the *form* as current and executable based on the *specifier*. If *specifier* is not provided `form-specifier` will be used to get it from *form*.

`print-filecom` *filecom stream* &optional *specifier* [Function]

Prints a new line on *stream* and then pretty prints a form representing the filecom onto the *stream*. If *specifier* is not provided `filecom-specifier` will be used to get it from *filecom*.

Presentation Objects

Presentation objects represent things that normally disappear during reading, like comments or numbers written in a particular base. Each presentation object must be capable of being read from a text file, edited with SEdit, installed as it would be when normally read, and printed to a text file in its original form.

Presentations Supported by Lisp

Many presentations are already supported by SEdit :

#\character	Character object
#:symbol	Uninterned symbol
#'function	Hash quote function abbreviation
; comment	
;; comment	
;;; comment	
;;;; comment	Semicolon comments. Internal formatting is preserved when these are imported, including CRs and tabs, etc. Adjacent comments are *not* smashed together so that line breaks are preserved. A single leading space in a comment is ignored, since comments are always printed with a single leading space.

These comment types are represented internally in the same way as in Lisp, i.e., a list beginning with the symbol

IL:*, following by a symbol (interned in the INTERLISP package) containing one through four semicolons.

#|*comment***|#** A balanced comment. Imported and exported by TextModules. Directly supported by SEdit as though it were a "level 5" semicolon comment.

This comment type is represented internally in a manner similar to semicolon comments, but where the symbol containing semicolons is replaced by a symbol whose name is the vertical bar character.

Presentations Supported by SEDIT-COMMONLISP

Several presentations are specially supported by SEDIT-COMMONLISP. Any of these can be created using the following commands in SEdit:

Read time conditionals	Control-N	Hash minus
	Control-P	Hash plus
Read time evaluation	Control-Q	Hash dot
Load time evaluation	Control-F	Hash comma
Octal notation	Control-I	Hash "O"
Hexidecimal notation	Control-J	Hash "X"
Binary notation	Control-K	Hash "B"

#+*feature form*
#-*feature form* Read time conditionals.

A conditional expression can be either *unread* or *read* depending on whether the truth of its features expression and sign parse true (see *Common Lisp, the Language*). *Read* conditional expressions are stored as structure. *Unread* conditional expressions are stored as strings due to the potential inclusion of, e.g., numbers of higher precision, symbols in unknown packages, or the inclusion of unknown reader macros.

Unread read time conditionals are read by remembering file position, doing a read suppress read, backing up to the original position and saving all the characters between in a string. This means that streams from which conditional read presentations are read must be capable of random access (the TTY is not).

These are represented by hash-plus and hash-minus structures.

Editing of read time conditional presentations is not quite WYSIWYG. Feature symbols should always be given as keywords (this is done implicitly by the lisp reader, but not in SEdit) and unread forms appear in strings and must be edited as such.

#.*form* Read time evaluation. Hash dot is represented by the hash-dot presentation.

#.*form* Load time evaluation. Hash comma is represented by the hash-comma structure.

#O*rational*

#X*rational*	
#B*rational*	Rational representations (**O**ctal, he**X**idecimal, and **B**inary). These are represented by the hash-o, hash-x and hash-b structures.

Presentations Not Directly Supported

It is not possible to directly edit the following presentations in SEdit:

#(*contents*)	Vectors
#*rank*A(*contents*)	Arrays
#S(*name field1 value1 . . .*)	Structures
#*1010101	Bit vectors

Any of these may be edited by opening an inspector from SEdit: selecting the object and using the Meta-E command on it. Any of these may be created in SEdit by inserting the appropriate **make-** expression, selecting it, and using the Meta-Z command with **cl:eval** as the mutating function.

Presentations Not Supported

The following standard Common Lisp presentations are not supported by either SEdit or TextModules:

#*n=object* and #*n*#	Object tag and reference notation
#*base*R*number*	Radix notation

[This page intentionally left blank]