17. FILE PACKAGE

*Warning: The subsystem within the Interlisp-D environment used for managing collections of definitions (of functions, variables, etc.) is known as the "File Package." This terminology is confusing, because the word "file" is also used in the more conventional sense as meaning a collection of data stored some physical media. Unfortunately, it is not possible to change this terminology at this time, because many functions and variables (***MAKEFILE***, ***FILEPKGTYPES***, etc.) incorporate the word "file" in their names. Eventually, the system and the documentation will be revamped to consistantly use the term "module" or "definition group" or "defgroup."*

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects---function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Interlisp can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Interlisp is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file package provides those capabilities. It removes from the user the burden of keeping track of where things are and what things have changed. The file package also keeps track of which files have been modified and need to be updated and recompiled.

The file package is integrated into many other system packages. For example, if only the compiled version of a file is loaded and the user attempts to edit a function, the file package will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some

program, the file package will automatically retrieve it from a file if it is not already in the user's working environment.

Some of the operations of the file package are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file package works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file package does, suppose that the symbolic file **FOO** contains the functions **FOO1** and **FOO2**, and that the file **BAR** contains the functions **BAR1** and **BAR2**. These two files could be loaded into the environment with the function **LOAD**:

```
←(LOAD 'FOO)
FILE CREATED 4-MAR-83 09:26:55
FOOCOMS
{DSK}FOO.;1
←(LOAD 'BAR)
FILE CREATED 4-MAR-83 09:27:24
BARCOMS
{DSK}BAR.;1
```

Now, suppose that we change the definition of **FOO2** with the editor, and we define two new functions, **NEW1** and **NEW2**. At that point, the file package knows that the in-memory definition of **FOO2** is no longer consistent with the definition in the file **FOO**, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function **FILES?** summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
←(FILES?)
FOO...to be dumped.
  plus the functions: NEW1,NEW2
want to say where the above go ? Yes
(functions)
NEW1  File name: BAR
NEW2  File name: ZAP
 new file ? Yes
NIL
```

The file package knows that the file **FOO** has been changed, and needs to be dumped back to permanent storage. This can be done with **MAKEFILE**.

```
←(MAKEFILE 'FOO)
{DSK}FOO.;2
```

Since we added **NEW1** to the old file **BAR** and established a new file **ZAP** to contain **NEW2**, both **BAR** and **ZAP** now also need to be dumped. This is confirmed by a second call to **FILES?**:

← (FILES?)
BAR, ZAP...to be dumped.
FOO...to be listed.
FOO...to be compiled
NIL

We are also informed that the new version we made of **FOO** needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several **MAKEFILE**s to dump the files **BAR** and **ZAP**, we can simply call **CLEANUP**. Without any further user interaction, this will dump any files whose definitions have been modified. **CLEANUP** will also send any unlisted files to the printer and recompile any files which need to be recompiled. **CLEANUP** is a useful function to use at the end of a debugging session. It will call **FILES?** if any new objects have been defined, so the user does not lose the opportunity to say explicitly where those belong. In effect, the function **CLEANUP** executes all the operations necessary to make the user's permanent files consistent with the definitions in his current core-image.

← (CLEANUP)
FOO...compiling {DSK}FOO.;2
.
.
.
BAR...compiling {DSK}BAR.;2
.
.
.
ZAP...compiling {DSK}ZAP.;1
.
.
.

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note that the same name may have several definitions of different types. For example, a litatom may have both a function

definition and a variable definition. The file package also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file package, they are assigned simple *root names* and these are used by all file package operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function **ROOTFILENAME**; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file package also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is *X* is stored as the value of the litatom *X***COMS**. For example, the value of **FOOCOMS** is the filecoms for the file **FOO**. This variable can be directly manipulated, but the file package contains facilities such as **FILES?** which make constructing and updating filecoms easier, and in some cases automatic. See page 17.48.

The file package is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file package takes it into account when modifying filecoms, dumping files, etc. The file package also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (**LOAD**, **TCOMPL**, **PRETTYDEF**, etc.), as well as those functions that define or change data, (**EDITF**, **EDITV**, **EDITP**, DWIM corrections to user functions) interact with the file package. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using **EDITP**, only those properties whose values are actually changed (or added) are marked.

All file package operations can be disabled with **FILEPKGFLG**.

FILEPKGFLG [Variable]

The file package can be disabled by setting **FILEPKGFLG** to **NIL**. This will turn off noticing files and marking changes. **FILEPKGFLG** is initially **T**.

The rest of this chapter goes into further detail about the file package. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

17.1 Loading Files

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using **FILERDTBL** as the read table.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of **DEFINE**, **DEFINEQ**, **RPAQ**, **RPAQ?**, and **RPAQQ**. While a source file is being loaded, **DFNFLG** (page 10.10) is rebound to *LDFLG*. Thus, if *LDFLG* = **NIL**, and a function is redefined, a message is printed and the old definition saved. If *LDFLG* = **T**, the old definition is simply overwritten. If *LDFLG* = **PROP**, the functions are stored as "saved" definitions on the property lists under the property **EXPR** instead of being installed as the active definitions. If *LDFLG* = **ALLPROP**, not only function definitions but also variables set by **RPAQQ**, **RPAQ**, **RPAQ?** are stored on property lists (except when the variable has the value **NOBIND**, in which case they are set to the indicated value regardless of **DFNFLG**).

Another option is available for users who are loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG* = **SYSLOAD**, **LOAD** will: (1) Rebind **DFNFLG** to **T**, so old definitions are simply overwritten; (2) Rebind **LISPXHIST** to **NIL**, thereby making the **LOAD** not be undoable and eliminating the cost of saving undo information (See page 13.26); (3) Rebind **ADDSPELLFLG** to **NIL**, to suppress adding to spelling lists; (4) Rebind **FILEPKGFLG** to **NIL**, to prevent the file from being "noticed" by the file package; (5) Rebind **BUILDMAPFLG** to **NIL**, to prevent a file map from being constructed; (6) After the load has completed, set the filecoms variable and any filevars

variables to **NOBIND**; and (7) Add the file name to **SYSFILES** rather than **FILELST**.

Note: A filevars variable is any variable appearing in a file package command of the form (*FILECOM* * *VARIABLE*) (see page 17.44). Therefore, if the filecoms includes (**FNS** * **FOOFNS**), **FOOFNS** is set to **NOBIND**. If the user wants the value of such a variable to be retained, even when the file is loaded with *LDFLG* = **SYSLOAD**, then he should replace the variable with an equivalent, *non-atomic* expression, such as (**FNS** * (**PROGN FOOFNS**)).

All functions that have *LDFLG* as an argument perform spelling correction using **LOADOPTIONS** as a spelling list when *LDFLG* is not a member of **LOADOPTIONS**. **LOADOPTIONS** is initially (**NIL T PROP ALLPROP SYSLOAD**).

(LOAD *FILE LDFLG PRINTFLG***)** [Function]

Reads successive expressions from *FILE* (with **FILERDTBL** as read table) and evaluates each as it is read, until it reads either **NIL**, or the single atom **STOP**. Note that **LOAD** can be used to load both symbolic and compiled files. Returns *FILE* (full name).

If *PRINTFLG* = **T**, **LOAD** prints the value of each expression; otherwise it does not.

(LOAD? *FILE LDFLG PRINTFLG***)** [Function]

Similar to **LOAD** except that it does not load *FILE* if it has already been loaded, in which case it returns **NIL**.

Note: **LOAD?** loads *FILE* except when the *same* version of the file has been loaded (either from the same place, or from a copy of it from a different place). Specifically, **LOAD?** considers that *FILE* has already been loaded if the full name of *FILE* is on **LOADEDFILELST** (page 17.20) or the date stored on the **FILEDATES** property of the root file name of *FILE* is the same as the **FILECREATED** expression on *FILE*.

(LOADFNS *FNS FILE LDFLG VARS***)** [Function]

Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or **T**, meaning to load all of the functions on the file. *FILE* can be either a compiled or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for **LOAD**.

If *FILE* = **NIL**, **LOADFNS** will use **WHEREIS** (page 17.14) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed" (see page 17.19). If **WHEREIS** returns **NIL**, and the WHEREIS

library package has been loaded, **LOADFNS** will use the WHEREIS data base to find the file containing *FN*.

VARS specifies which non-**DEFINEQ** expressions are to be loaded (i.e., evaluated). It is interpreted as follows:

T Means to load all non-**DEFINEQ** expressions.

NIL Means to load none of the non-**DEFINEQ** expressions.

VARS Means to evaluate all variable assignment expressions (beginning with **RPAQ**, **RPAQQ**, or **RPAQ?**, see page 17.54).

Any other litatom Means the same as specifying a list containing that atom.

A list If *VARS* is a list that is not a valid function definition, each element in *VARS* is "matched" against each non-**DEFINEQ** expression, and if any elements in *VARS* "match" successfully, the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is **EQ** to either the **CAR** or the **CADR** of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (page 16.18), and matched with the entire expression (using **EDIT4E**, page 16.72). For example, if *VARS* was **(FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO)))**, this would cause **(RPAQQ FOOCOMS ...)**, all **DECLARE:**s, and all **DEFLISTs** which set up **MACROs** to be read and evaluated.

A function definition If *VARS* is a list and a valid function definition (**(FNTYP** *VARS*) is true), then **LOADFNS** will invoke that function on every non-**DEFINEQ** expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns **NIL**, the expression will be skipped; if it returns a non-**NIL** litatom (e.g. **T**), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. Note: The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a litatom, the file pointer is reset and the expression is **READ** or **SKREAD**. However, the file pointer is not reset when the function returns a list, so the function must leave it set immediately after the expression that it has presumably read.

LOADFNS returns a list of: (1) The names of the functions that were found; (2) A list of those functions not found (if any) headed by the litatom **NOT-FOUND:**; (3) All of the expressions that were evaluated; (4) A list of those members of *VARS* for which no corresponding expressions were found (if any), again headed by the litatom **NOT-FOUND:**. For example,

←(LOADFNS '(FOO FIE FUM) *FILE* **NIL '(BAZ (DEFLIST &)))**
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ...) (NOT-FOUND: (DEFLIST &)))

(**LOADVARS** *VARS FILE LDFLG*) [Function]

Same as (**LOADFNS NIL** *FILE LDFLG VARS*).

(**LOADFROM** *FILE FNS LDFLG*) [Function]

Same as (**LOADFNS** *FNS FILE LDFLG* T).

Once the file package has noticed a file, the user can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed (i.e., their contents become known to the file package; see page 17.19) when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with **LOADFROM**. Note that the user can also load some functions at the same time by giving **LOADFROM** a second argument, but it is normally used simply to inform the file package about the existence and contents of a particular file.

(**LOADBLOCK** *FN FILE LDFLG*) [Function]

Calls **LOADFNS** on those functions contained in the block declaration containing *FN* (See page 18.17). **LOADBLOCK** is designed primarily for use with symbolic files, to load the **EXPR**s for a given block. It will not load a function which already has an in-core **EXPR** definition, and it will not load the block name, unless it is also one of the block functions.

(**LOADCOMP** *FILE LDFLG*) [Function]

Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions under a **DECLARE: EVAL@COMPILE** (see page 17.40), and "notices" the function and variable names by adding them to the lists **NOFIXFNSLST** and **NOFIXVARSLST** (see page 21.21).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to **LOADCOMP** the others.

(**LOADCOMP?** *FILE LDFLG*) [Function]

Similar to **LOADCOMP**, except it does not load if file has already been loaded (with **LOADCOMP**), in which case its value is **NIL**.

Note: **LOADCOMP?** will load the file even if it has been loaded with **LOAD**, **LOADFNS**, etc. The only time it will not load the file is if the file has already been loaded with **LOADCOMP**.

FILESLOAD provides an easy way for the user to load a series of files, setting various options:

(FILESLOAD *FILE*$_1$ ... *FILE*$_N$**)** [NLambda NoSpread Function]

Loads the files *FILE*$_1$ ... *FILE*$_N$ (all arguments unevaluated). If any of these arguments are lists, they specify certain loading options for all following files (unless changed by another list). Within these lists, the following commands are recognized:

FROM *DIR* Search the specified directories for the file. *DIR* can either be a single directory, or a list of directories to search in order. For example, **(FILESLOAD (FROM {ERIS}<LISPCORE>SOURCES>) ...)** will search the directory **{ERIS}<LISPCORE>SOURCES>** for the files. If this is not specified, the default is to search the contents of **DIRECTORIES** (page 24.31).

If **FROM** is followed by the key word **VALUEOF**, the following word is evaluated, and the value is used as the list of directories to search. For example, **(FILESLOAD (FROM VALUEOF FOO) ...)** will search the directory list that is the value of the variable **FOO**.

As a special case, if *DIR* is a litatom, and the litatom *DIR***DIRECTORIES** is bound, the value of this variable is used as the directory search list. For example, since the variable **LISPUSERSDIRECTORIES** (page 24.32) is commonly used to contain a list of directories containing "library" packages, **(FILESLOAD (FROM LISPUSERS) ...)** can be used instead of **(FILESLOAD (FROM VALUEOF LISPUSERSDIRECTORIES) ...)**

Note: If a **FILESLOAD** is read and evaluated while loading a file, and it doesn't contain a **FROM** expression, the default is to search the directory containing the **FILESLOAD** expression before the value of **DIRECTORIES**. **FILESLOAD** expressions can be dumped on files using the **FILES** file package command (page 17.39).

SOURCE Load the source version of the file rather than the compiled version.

COMPILED Load the compiled version of the file.

Note: If **COMPILED** is specified, the compiled version will be loaded, if it is found. The source will not be loaded. If neither **SOURCE** or **COMPILED** is specified, the compiled version of the file will be loaded if it is found, otherwise the source will be loaded if it is found.

LOAD Load the file by calling **LOAD**, if it has not already been loaded. This is the default unless **LOADCOMP** or **LOADFROM** is specified.

Note: If **LOAD** is specified, **FILESLOAD** considers that the file has already been loaded if the root name of the file has a non-**NIL FILEDATES** property. This is a somewhat different algorithm than **LOAD?** uses. In particular, **FILESLOAD** will not load a newer version of a file that has already been loaded.

LOADCOMP Load the file with **LOADCOMP?** rather than **LOAD**. Automatically implies **SOURCE**.

LOADFROM Load the file with **LOADFROM** rather than **LOAD**.

NIL
T
PROP
ALLPROP
SYSLOAD The loading function is called with its *LDFLG* argument set to the specified token (see page 17.5). *LDFLG* affects the operation of the loading functions by resetting **DFNFLG** (page 10.10) to *LDFLG* during the loading. If none of these tokens are specified, the value of the variable **LDFLG** is used if it is bound, otherwise **NIL** is used.

NOERROR If **NOERROR** is specified, no error occurs when a file is not found.

Each list determines how all further files in the lists are loaded, unless changed by another list. The tokens above can be joined together in a single list. For example,

(FILESLOAD (LOADCOMP) NET (SYSLOAD FROM VALUEOF NEWDIRECTORIES) CJSYS)

will call **LOADCOMP?** to load the file **NET** searching the value of **DIRECTORIES**, and then call **LOADCOMP?** to load the file **CJSYS** with *LDFLG* set to **SYSLOAD**, searching the directory list that is the value of the variable **NEWDIRECTORIES**.

FILESLOAD expressions can be dumped on files using the **FILES** file package command (page 17.39).

17.2 Storing Files

(MAKEFILE *FILE OPTIONS REPRINTFNS SOURCEFILE***)** [Function]

Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms. Notices *FILE* if not previously noticed (see page 17.19). Then, it adds *FILE* to **NOTLISTEDFILES** and **NOTCOMPILEDFILES**.

OPTIONS is a litatom or list of litatoms which specify options. By specifying certain options, **MAKEFILE** can automatically compile or list *FILE*. Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies **C** or **RC**. The options are spelling corrected using the list **MAKEFILEOPTIONS**. If spelling correction fails, **MAKEFILE** generates an error. The options are interpreted as follows:

C
RC After making *FILE*, **MAKEFILE** will compile *FILE* by calling **TCOMPL** (if **C** is specified) or **RECOMPILE** (if **RC** is specified). If there are any block declarations specified in the filecoms for *FILE*, **BCOMPL** or **BRECOMPILE** will be called instead.

If **F**, **ST**, **STF**, or **S** is the *next* item on *OPTIONS* following **C** or **RC**, it is given to the compiler as the answer to the compiler's question **LISTING?** (see page 18.1). For example, **(MAKEFILE 'FOO '(C F LIST))** will dump **FOO**, then **TCOMPL** or **BCOMPL** it specifying that functions are not to be redefined, and finally list the file.

LIST After making *FILE*, **MAKEFILE** calls **LISTFILES** to print a hardcopy listing of *FILE*.

CLISPIFY **MAKEFILE** calls **PRETTYDEF** with **CLISPIFYPRETTYFLG = T** (see page 21.26). This causes **CLISPIFY** to be called on each function defined as an **EXPR** before it is prettyprinted.

Alternatively, if *FILE* has the property **FILETYPE** with value **CLISP** or a list containing **CLISP**, **PRETTYDEF** is called with **CLISPIFYPRETTYFLG** reset to **CHANGES**, which will cause **CLISPIFY** to be called on all functions marked as having been changed. If *FILE* has property **FILETYPE** with value **CLISP**, the compiler will **DWIMIFY** its functions before compiling them (see page 18.11).

FAST **MAKEFILE** calls **PRETTYDEF** with **PRETTYFLG = NIL** (see page 26.48). This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE **MAKEFILE** "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted. See page 17.15.

NEW **MAKEFILE** does *not* remake *FILE*. If **MAKEFILEREMAKEFLG = T** (the initial setting), the default for all calls to **MAKEFILE** is to remake. The **NEW** option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file, as described on page 17.15.

Note: *FILE* is not added to **NOTLISTEDFILES** if *FILE* has on its property list the property **FILETYPE** with value **DON'TLIST**, or a list containing **DON'TLIST**. *FILE* is not added to **NOTCOMPILEDFILES** if *FILE* has on its property list the property **FILETYPE** with value **DON'TCOMPILE**, or a list containing **DON'TCOMPILE**. Also, if *FILE* does not contain any function definitions, it is not added to **NOTCOMPILEDFILES**, and it is not compiled even when *OPTIONS* specifies **C** or **RC**.

If a remake is *not* being performed, **MAKEFILE** checks the state of *FILE* to make sure that the entire source file was actually **LOAD**ed. If *FILE* was loaded as a compiled file, **MAKEFILE** prints the message **CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED**. Similarly, if only some of the symbolic definitions were loaded via **LOADFNS** or **LOADFROM**, **MAKEFILE** prints **CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED**. In both cases, **MAKEFILE** will then ask the user if it should dump

anyway; if the user declines, **MAKEFILE** does not call **PRETTYDEF**, but simply returns (*FILE* **NOT DUMPED**) as its value.

The user can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property **FILEGROUP**. If *FILE* has a **FILEGROUP** property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding **PRETTYFLG**, **PRETTYTRANFLG**, and **CLISPIFYPRETTYFLG**, evaluating each expression on **MAKEFILEFORMS** (under errorset protection), and then calling **PRETTYDEF** (page 17.50).

Note: **PRETTYDEF** calls **PRETTYPRINT** with its second argument *PRETTYDEFLG* = **T**, so whenever **PRETTYPRINT** (and hence **MAKEFILE**) start printing a new function, the name of that function is printed if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

(MAKEFILES *OPTIONS FILES***)** [Function]

Performs **(MAKEFILE** *FILE OPTIONS***)** for each file on *FILES* that needs to be dumped. If *FILES* = **NIL**, **FILELST** is used. For example, **(MAKEFILES 'LIST)** will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on **FILELST**, **MAKEFILES** calls **ADDTOFILES?** to allow the user to specify where these go. **MAKEFILES** returns a list of all files that are made.

(CLEANUP $FILE_1$ $FILE_2$ ... $FILE_N$**)** [NLambda NoSpread Function]

Dumps, lists, and recompiles (with **RECOMPILE** or **BRECOMPILE**) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, **FILELST** is used. **CLEANUP** returns **NIL**.

CLEANUP uses the value of the variable **CLEANUPOPTIONS** as the *OPTIONS* argument to **MAKEFILE**. **CLEANUPOPTIONS** is initially **(RC)**, to indicate that the files should be recompiled. If **CLEANUPOPTIONS** is set to **(RC F)**, no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if $FILE_1$ is a list, it will be interpreted as the list of options regardless of the value of **CLEANUPOPTIONS**.

(FILES?) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are

any, **FILES?** then calls **ADDTOFILES?** to allow the user to specify where these go.

(ADDTOFILES? —) [Function]

Called from **MAKEFILES**, **CLEANUP**, and **FILES?** when there are typed definitions that have been marked as changed which do not belong to any file. **ADDTOFILES?** lists the names of the changed items, and asks the user if he wants to specify where these items should be put. If user answers **N**(o), **ADDTOFILES?** returns **NIL** without taking any action. If the user answers], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, **ADDTOFILES?** prints the name of each changed item, and accepts one of the following responses:

A file name
A filevar — If the user gives a file name or a variable whose value is a list (a filevar), the item is added to the corresponding file or list, using **ADDTOFILE**.

If the user response is not the name of a file on **FILELST** or a variable whose value is a list, the user will be asked whether it is a new file. If he says no, then **ADDTOFILES?** will check whether the item is the name of a list, i.e. whether its value is a list. If not, the user will be asked whether it is a new list.

line-feed — Same as the user's previous response.

space
carriage return — Take no action.

] — The item is marked as a dummy item by adding it to **NILCOMS**. This tells the file package simply to ignore this item.

[— The "definition" of the item in question is prettyprinted to the terminal, and then the user is asked again about its disposition.

(— **ADDTOFILES?** prompts with "**LISTNAME: (**", the user types in the name of a list, i.e. a variable whose value is a list, terminated by a **)**. The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and the user is asked again. For example, the user defines a new function **FOO3**, and when asked where it goes, types **(FOOFNS)**. If the command **(FNS * FOOFNS)** is found, **FOO3** will be added to the value of **FOOFNS**. If instead the user types **(FOOCOMS)**, and the command **(COMS * FOOCOMS)** is found, then **FOO3** will be added to a command for dumping functions that is contained in **FOOCOMS**.

Note: If the named list is not also the name of a file, the user can simply type it in without parenthesis as described above.

@ — **ADDTOFILES?** prompts with "**Near: (**", the user types in the name of an object, and the item is then inserted in a command for

dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(LISTFILES *FILE$_1$ FILE$_2$ ... FILE$_N$***)** [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, **NOTLISTEDFILES** is used. Each file listed is removed from **NOTLISTEDFILES** if the listing is completed. For each file not found, **LISTFILES** prints the message "*FILENAME* **NOT FOUND**" and proceeds to the next file.

LISTFILES calls the function **LISTFILES1** on each file to be listed. Normally, **LISTFILES1** is defined to simply call **SEND.FILE.TO.PRINTER** (page 29.1), but the user can advise or redefine **LISTFILES1** for more specialized applications.

Any lists inside the argument list to LISTFILES are interpreted as property lists that set the various printing options, such as the printer, number of copies, banner page name, etc (see page 29.1). Later properties override earlier ones. For example,

(LISTFILES FOO (HOST JEDI) FUM (#COPIES 3) FIE)

will cause one copy of **FOO** to be printed on the default printer, and 1 copy of **FUM** and 3 copies of **FIE** to be printed on the printer **JEDI**.

(COMPILEFILES *FILE$_1$ FILE$_2$ ... FILE$_N$***)** [NLambda NoSpread Function]

Executes the **RC** and **C** options of **MAKEFILE** for each of the specified files (unevaluated). If no files are given, **NOTCOMPILEDFILES** is used. Each file compiled is removed from **NOTCOMPILEDFILES**. If *FILE$_1$* is a list, it is interpreted as the *OPTIONS* argument to **MAKEFILES**. This feature can be used to supply an answer to the compiler's **LISTING?** question, e.g., **(COMPILEFILES (STF))** will compile each file on **NOTCOMPILEDFILES** so that the functions are redefined without the **EXPR**s definitions being saved.

(WHEREIS *NAME TYPE FILES FN***)** [Function]

TYPE is a file package type. **WHEREIS** sweeps through all the files on the list *FILES* and returns a list of all files containing *NAME* as a *TYPE*. **WHEREIS** knows about and expands all file package commands and file package macros. *TYPE* = **NIL** defaults to **FNS** (to retrieve function definitions). If *FILES* is not a list, the value of **FILELST** is used.

If *FN* is given, it should be a function (with arguments *NAME*, *FILE*, and *TYPE*) which is applied for every file in *FILES* that contains *NAME* as a *TYPE*. In this case, **WHEREIS** returns **NIL**.

If the WHEREIS library package has been loaded, **WHEREIS** is redefined so that *FILES* = **T** means to use the whereis package

data base, so **WHEREIS** will find *NAME* even if the file has not been loaded or noticed. *FILES* = **NIL** always means use **FILELST**.

17.3 Remaking a Symbolic File

Most of the time that a symbolic file is written using **MAKEFILE**, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the **REMAKE** option is specified. If the **NEW** option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of **MAKEFILEREMAKEFLG**: if **T** (its initial value), **MAKEFILE** will remake files unless the **NEW** option is given; if **NIL**, **MAKEFILE** will not remake unless the **REMAKE** option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then **MAKEFILE** will never attempt to remake the file, regardless of the setting of **MAKEFILEREMAKEFLG**, or whether the **REMAKE** option was specified.

When **MAKEFILE** is remaking a symbolic file, the user can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to **MAKEFILE**. Normally, both of these arguments are defaulted to **NIL**. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, **MAKEFILE** obtains the full name of the most recent version of the file (that it knows about) from the **FILEDATES** property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the **FILEDATES** property. If it does, **MAKEFILE** uses that file as *SOURCEFILE*. This procedure permits the user to **LOAD** or **LOADFROM** a file in a different directory, and still be able to remake the file with **MAKEFILE**. In the case where the most recent version of the file cannot be found, **MAKEFILE** will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the **FILECHANGES** property of the file. If both of these fail, **MAKEFILE** prints the message "**CAN'T FIND**

EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, **MAKEFILE** also checks to see how the file was originally loaded (see page 17.19). If the file was originally loaded as a compiled file, **MAKEFILE** will call **LOADVARS** to obtain those **DECLARE:** expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by **LOADFNS** (but not **LOADFROM**), then **LOADVARS** is called to obtain any non-**DEFINEQ** expressions. Before calling **LOADVARS** to re-load definitions, **MAKEFILE** asks the user, e.g. "**Only the compiled version of FOO was loaded, do you want to LOADVARS the (DECLARE: .. DONTCOPY ..) expressions from {DSK}<MYDIR>FOO.;3?**". The user can respond **Yes** to execute the **LOADVARS** and continue the **MAKEFILE**, **No** to proceed with the **MAKEFILE** without performing the **LOADVARS**, or **Abort** to abort the **MAKEFILE**. The use may wish to skip the **LOADVARS** if the user had circumvented the file package in some way, and loading the old definitions would overwrite new ones.

Note: Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (page 17.55), but it does not depend on this information.

17.4 Loading Files in a Distributed Environment

Each Interlisp source and compiled code file contains the full filename of the file, including the host and directory names, in a **FILECREATED** expression at the beginning of the file. The compiled code file also contains the full file name of the source file it was created from. In earlier versions of Interlisp, the file package used this information to locate the appropriate source file when "remaking" or recompiling a file.

This turned out to be a bad feature in distributed environments, where users frequently move files from one place to another, or where files are stored on removable media. For example, suppose you **MAKEFILE** to a floppy, and then copy the file to a file server. If you loaded and edited the file from a file server, and tried to do **MAKEFILE**, it would try to locate the source file on the floppy, which is probably no longer loaded.

Currently, the file package searches for sources file on the connected directory, and on the directory search path (on the variable **DIRECTORIES**). If it is not found, the host/directory information from the **FILECREATED** expression be used.

Warning: One situation where the new algorithm does the wrong thing is if you explicitly **LOADFROM** a file that is not on your directory search path. Future **MAKEFILE**s and **CLEANUP**s will search the connected directory and **DIRECTORIES** to find the source file, rather than using the file that the **LOADFROM** was done from. Even if the correct file is on the directory search path, you could still create a bad file if there is another version of the file in an earlier directory on the search path. In general, you should either explicitly specify the *SOURCEFILE* argument to **MAKEFILE** to tell it where to get the old source, or connect to the directory where the correct source file is.

17.5 Marking Changes

The file package needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file package operations (**LOAD**, **TCOMPL**, **PRETTYDEF**, etc.), as well as those functions that define or change data, (**EDITF**, **EDITV**, **EDITP**, DWIM corrections to user functions) interact with the file package by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file package. (Note that if a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if the user edits a property list using **EDITP**, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call **MARKASCHANGED** to mark the object as changed. For example, when a function is defined via **DEFINE** or **DEFINEQ**, or modified via **EDITF**, or a DWIM correction, the function is marked as being a changed object of type **FNS**. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type **RECORDS**, and so on for all of the other file package types.

The user can also call **MARKASCHANGED** directly to mark objects of a particular file package type as changed:

(MARKASCHANGED *NAME TYPE REASON***)** [Function]

Marks *NAME* of type *TYPE* as being changed. **MARKASCHANGED** returns *NAME*. **MARKASCHANGED** is undoable.

REASON is a litatom that indicated how *NAME* was changed. **MARKASCHANGED** recognizes the following values for *REASON*:

DEFINED Used to indicate the creation of *NAME*, e.g. from **DEFINEQ** (page 10.9).

CHANGED Used to indicate a change to *NAME*, e.g. from the editor.

DELETED Used to indicate the deletion of *NAME*, e.g. by **DELDEF** (page 17.27).

CLISP Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, **MARKASCHANGED** also accepts a *REASON* of **T** (= **DEFINED**) and **NIL** (= **CHANGED**). New programs should avoid using these values.

Note: The variable **MARKASCHANGEDFNS** is a list of functions that **MARKASCHANGED** calls (with arguments *NAME*, *TYPE*, and *REASON*). Functions can be added to this list to "advise" **MARKASCHANGED** to do additional work for all types of objects. The **WHENCHANGED** file package type property (page 17.31) can be used to specify additional actions when **MARKASCHANGED** gets called on specific types of objects.

(UNMARKASCHANGED *NAME TYPE***)** [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, **NIL** otherwise. **UNMARKASCHANGED** is undoable.

(FILEPKGCHANGES *TYPE LST***)** [NoSpread Function]

If *LST* is not specified (as opposed to being **NIL**), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (See page 17.21). If *LST* is specified, **FILEPKGCHANGES** sets the corresponding list. **(FILEPKGCHANGES)** returns a list of *all* objects marked as changed as a list of elements of the form **(***TYPENAME* . *CHANGEDOBJECTS***)**.

Some properties (e.g. **EXPR, ADVICE, MACRO, I.S.OPR**, etc..) are used to implement other file package types. For example, if the user changes the value of the property **I.S.OPR**, he is really changing an object of type **I.S.OPR**, and the effect is the same as though he had redefined the i.s.opr via a direct call to the function **I.S.OPR**. If a property whose value has been changed or added does not correspond to a specific file package type, then it is marked as a changed object of type **PROPS** whose *name* is **(***VARIABLENAME PROPNAME***)** (except if the property name has a property **PROPTYPE** with value **IGNORE**).

Similarly, if the user changes a variable which implements the file package type **ALISTS** (as indicated by the appearance of the property **VARTYPE** with value **ALIST** on the variable's property list), only those entries that are actually changed are marked as being changed objects of type **ALISTS**, and the "name" of the

object will be (*VARIABLENAME KEY*) where *KEY* is **CAR** of the entry on the alist that is being marked. If the variable corresponds to a specific file package type other than **ALISTS**, e.g. **USERMACROS**, **LISPXMACROS**, etc., then an object of that type is marked. In this case, the name of the changed object will be **CAR** of the corresponding entry on the alist. For example, if the user edits **LISPXMACROS** and changes a definition for **PL**, then the object **PL** of type **LISPXMACROS** is marked as being changed.

17.6 Noticing Files

Already existing files are "noticed" by **LOAD** or **LOADFROM** (or by **LOADFNS** or **LOADVARS** when the *VARS* argument is **T**. New files are noticed when they are constructed by **MAKEFILE**, or when definitions are first associated with them via **FILES?** or **ADDTOFILES?**. Noticing a file updates certain lists and properties so that the file package functions know to include the file in their operations. For example, **CLEANUP** will only dump files that have been noticed.

The user can explicitly tell the file package to notice a newly-created file by defining the filecoms for the file (see page 17.32), and calling **ADDFILE**:

(ADDFILE *FILE* — — —**)** [Function]

Tells the file package that *FILE* should be recognized as a file; it adds *FILE* to **FILELST**, and also sets up the **FILE** property of *FILE* to reflect the current set of changes which are "registered against" *FILE*.

The file package uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE [Property Name]

When a file is noticed, the property **FILE**, value **((***FILECOMS* . *LOADTYPE***))** is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file (see page 17.32). *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with **LOADFNS**, loaded as a compiled file, etc.

The property **FILE** is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. **CDR** of the **FILE** property records by type

those items that have been changed since the last **MAKEFILE**. Whenever a file is dumped, these items are moved to the property **FILECHANGES**, and **CDR** of the **FILE** property is reset to **NIL**.

FILECHANGES [Property Name]

The property **FILECHANGES** contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in **CDR** of the **FILE** property are added to the **FILECHANGES** property.

FILEDATES [Property Name]

The property **FILEDATES** contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with remaking a file (see page 17.15).

FILEMAP [Property Name]

The property **FILEMAP** is used to store the filemap for the file (see page 17.55). This is used to directly load individual functions from the middle of a file.

To compute the root name, **ROOTFILENAME** is applied to the name of the file as indicated in the **FILECREATED** expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file package detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one **FILECREATED** expressions. In this case, each of the files mentioned in the following **FILECREATED** expressions are noticed. For example, if the user performs **(BCOMPL '(FOO FIE))**, and subsequently loads **FOO.DCOM**, both **FOO** and **FIE** will be noticed.

When a file is noticed, its root name is added to the list **FILELST**:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by **LOAD**, **LOADFNS**, etc. For example, if the user performs **(LOAD '<NEWLISP>EDITA.COM;3)**, **EDITA** will be added to **FILELST**, but **<NEWLISP>EDITA.COM;3** is added to **LOADEDFILELST**. **LOADEDFILELST** is not used by the file package; it is maintained solely for the user's benefit.

17.7 Distributing Change Information

Periodically, the function **UPDATEFILES** is called to find which file(s) contain the elements that have been changed. **UPDATEFILES** is called by **FILES?**, **CLEANUP**, and **MAKEFILES**, i.e., any procedure that requires the **FILE** property to be up to date. This procedure is followed rather than updating the **FILE** property after each change because scanning **FILELST** and examining each file package command can be a time-consuming process; this is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

UPDATEFILES operates by scanning **FILELST** and interrogating the file package commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property **FILE** for the corresponding file. Thus, after **UPDATEFILES** has completed operating, the files that need to be dumped are simply those files on **FILELST** for which **CDR** of their **FILE** property is non-**NIL**. For example, if the user loads the file **FOO** containing definitions for **FOO1**, **FOO2**, and **FOO3**, edits **FOO2**, and then calls **UPDATEFILES**, **(GETPROP 'FOO 'FILE)** will be **((FOOCOMS . T) (FNS FOO2))**. If any objects marked as changed have not been transferred to the **FILE** property for some file, e.g., the user defines a new function but forgets (or declines) to add it to the file package commands for the corresponding file, then both **FILES?** and **CLEANUP** will print warning messages, and then call **ADDTOFILES?** to permit the user to specify on which files these items belong.

The user can also invoke **UPDATEFILES** directly:

(UPDATEFILES — —) [Function]

(UPDATEFILES) will update the **FILE** properties of the noticed files.

17.8 File Package Types

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file package uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a litatom), a definition of a given type (called the file package type). Note

that the same name may have several definitions of different types. For example, a litatom may have both a function definition and a variable definition. The file package also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file package type is an abstract notion of a class of objects which share the property that every object of the same file package type is stored, retrieved, edited, copied etc., by the file package in the same way. Each file package type is identified by a litatom, which can be given as an argument to the functions that manipulate typed definitions. The user may define new file package types, as described in page 17.32.

FILEPKGTYPES [Variable]

The value of **FILEPKGTYPES** is a list of all file package types, including any that may have been defined by the user.

The file package is initialized with the following built-in file package types:

ADVICE [File Package Type]

Used to access "advice" modifying a function (see page 15.9).

ALISTS [File Package Type]

Used to access objects stored on an association list that is the value of a litatom (see page 3.15).

A variable is declared to have an association list as its value by putting on its property list the property **VARTYPE** with value **ALIST**. In this case, each dotted pair on the list is an object of type **ALISTS**. When the value of such a variable is changed, only those entries in the association list that are actually changed or added are marked as changed objects of type **ALISTS** (with "name" (*LITATOM KEY*)). Objects of type **ALISTS** are dumped via the **ALISTS** or **ADDVARS** file package commands.

Note that some association lists are used to "implement" other file package types. For example, the value of the global variable **USERMACROS** implements the file package type **USERMACROS** and the values of **LISPXMACROS** and **LISPXHISTORYMACROS** implement the file package type **LISPXMACROS**. This is indicated by putting on the property list of the variable the property **VARTYPE** with value a list of the form (**ALIST** *FILEPKGTYPE*). For example, **(GETPROP 'LISPXHISTORYMACROS 'VARTYPE) => (ALIST LISPXMACROS)**.

COURIERPROGRAMS [File Package Type]

Used to access Courier programs (see page 31.15).

EXPRESSIONS [File Package Type]

Used to access lisp expressions that are put on a file by using the **REMEMBER** programmers assistant command (page 13.17), or by explicitly putting the **P** file package command (page 17.40) on the filecoms.

FIELDS [File Package Type]

Used to access fields of records. The "definition" of an object of type **FIELDS** is a list of all the record declarations which contain the name. See page 8.1.

FILEPKGCOMS [File Package Type]

Used to access file package commands and types. A single name can be defined both as a file package type and a file package command. The "definition" of an object of type **FILEPKGCOMS** is a list structure of the form **((COM .** *COMPROPS***) (TYPE .** *TYPEPROPS***))**, where *COMPROPS* is a property list specifying how the name is defined as a file package command by **FILEPKGCOM** (page 17.47), and *TYPEPROPS* is a property list specifying how the name is defined as a file package type by **FILEPKGTYPE** (page 17.32).

FILES [File Package Type]

Used to access files. This file package type is most useful for renaming files. The "definition" of a file is not a useful structure.

FILEVARS [File Package Type]

Used to access Filevars (see page 17.44).

FNS [File Package Type]

Used to access function definitions.

I.S.OPRS [File Package Type]

Used to access the definitions of iterative statement operators (see page 9.9).

LISPXMACROS [File Package Type]

Used to access programmer's assistant commands defined on the variables **LISPXMACROS** and **LISPXHISTORYMACROS** (see page 13.23).

MACROS [File Package Type]

Used to access macro definitions (see page 10.21).

PROPS [File Package Type]

Used to access objects stored on the property list of a litatom (see page 2.5). When a property is changed or added, an object of type **PROPS**, with "name" (*LITATOM PROPNAME*) is marked as being changed.

Note that some litatom properties are used to implement other file package types. For example, the property **MACRO** implements the file package type **MACROS**, the property **ADVICE** implements **ADVICE**, etc. This is indicated by putting the property **PROPTYPE**, with value of the file package type on the property list of the property name. For example, **(GETPROP 'MACRO 'PROPTYPE) => MACROS**. When such a property is changed or added, an object of the corresponding file package type is marked. If **(GETPROP** *PROPNAME* **'PROPTYPE) => IGNORE**, the change is ignored. The **FILE**, **FILEMAP**, **FILEDATES**, etc. properties are all handled this way. (Note that **IGNORE** cannot be the name of a file package type implemented as a property).

RECORDS [File Package Type]

Used to access record declarations (see page 8.1).

RESOURCES [File Package Type]

Used to access resources (see page 12.19).

TEMPLATES [File Package Type]

Used to access Masterscope templates (see page 19.18).

USERMACROS [File Package Type]

Used to access user edit macros (see page 16.62).

VARS [File Package Type]

Used to access top-level variable values.

17.8.1 Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, **(GETDEF 'FOO 'FNS)** will return the function definition of **FOO**, **(GETDEF 'FOO 'VARS)** will return the variable value of **FOO**, etc. All of the functions use the following conventions:

(1) All functions which make destructive changes are undoable.

(2) Any argument that expects a list of litatoms will also accept a single litatom, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

(3) *TYPE* is a file package type. *TYPE* = **NIL** is equivalent to *TYPE* = **FNS**. The singular form of a file package type is also recognized, e.g. *TYPE* = **VAR** is equivalent to *TYPE* = **VARS**.

(4) *FILES* = **NIL** is equivalent to *FILES* = **FILELST**.

(5) *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

CURRENT Get the definition currently in effect.

SAVED Get the "saved" definition, as stored by **SAVEDEF** (page 17.27).

FILE Get the definition contained on the (first) file determined by **WHEREIS** (page 17.14).

Note: **WHEREIS** is called with *FILES* = **T**, so that if the WHEREIS library package is loaded, the WHEREIS data base will be used to find the file containing the definition.

? Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by **WHEREIS**. Like specifying **CURRENT**, **SAVED**, and **FILE** in order, and taking the first definition that is found.

a file name
a list of file names — Get the definition from the first of the indicated files that contains one.

NIL In most cases, giving *SOURCE* = **NIL** (or not specifying it at all) is the same as giving **?**, to get either the current, saved, or filed definition. However, with **HASDEF**, *SOURCE* = **NIL** is interpreted as equal to *SOURCE* = **CURRENT**, which only tests if there is a current definition.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file package type using the function **FILEPKGTYPE**, described on page 17.32.

(GETDEF *NAME TYPE SOURCE OPTIONS***)** [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, **GETDEF** returns the expression which would be pretty printed when dumping *NAME* as *TYPE*. For example, for *TYPE* = **FNS**, an **EXPR** definition is returned, for *TYPE* = **VARS**, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR **GETDEF** causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains **NOERROR**. In this case,

GETDEF returns the value of the **NULLDEF** file package type property (page 17.30), usually **NIL**.

a string — If *OPTIONS* is or contains a string, that string will be returned if no definition is found (and **NOERROR** is not among the options). The caller can thus determine whether a definition was found, even for types for which **NIL** or **NOBIND** are acceptable definitions.

NOCOPY — **GETDEF** returns a copy of the definition unless *OPTIONS* is or contains **NOCOPY**.

EDIT — If *OPTIONS* is or contains **EDIT**, **GETDEF** returns a copy of the definition unless it is possible to edit the definition "in place." With some file package types, such as functions, it is meaningful (and efficient) to edit the definition by destructively modifying the list structure, without calling **PUTDEF**. However, some file package types (like records) need to be "installed" with **PUTDEF** after they are edited. The default **EDITDEF** (see page 17.31) calls **GETDEF** with *OPTIONS* of **(EDIT NOCOPY)**, so it doesn't use a copy unless it has to, and only calls **PUTDEF** if the result of editing is not **EQUAL** to the old definition.

NODWIM — A **FNS** definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains **NODWIM**.

(PUTDEF *NAME TYPE DEFINITION REASON***)** [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE* = **FNS**, does a **DEFINE**; for *TYPE* = **VARS**, does a **SAVESET**, etc.

For *TYPE* = **FILES**, **PUTDEF** establishes the command list, notices *NAME*, and then calls **MAKEFILE** to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

PUTDEF calls **MARKASCHANGED** (page 17.17) to mark *NAME* as changed, giving a reason of *REASON*. If *REASON* is **NIL**, the default is **DEFINED**.

Note: If *TYPE* = **FNS**, **PUTDEF** prints a warning if the user tries to redefine a function on the list **UNSAFE.TO.MODIFY.FNS** (page 10.10).

(HASDEF *NAME TYPE SOURCE SPELLFLG***)** [Function]

Returns **(OR** *NAME* **T)** if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG* = **T**, and returns the spelling-corrected *NAME*. Otherwise returns **NIL**.

(HASDEF NIL *TYPE***)** returns **T** if **NIL** has a valid definition.

Note: if *SOURCE* = **NIL**, *HASDEF* interprets this as equal to *SOURCE* = **CURRENT**, which only tests if there is a current definition.

(TYPESOF *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE*) [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which *NAME* has a definition. **FILEPKGTYPES** is used if *POSSIBLETYPES* is **NIL**.

(COPYDEF *OLD NEW TYPE SOURCE OPTIONS*) [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing **PUTDEF** on a copy of the definition retrieved by **(GETDEF** *OLD TYPE SOURCE OPTIONS*). *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, **(COPYDEF 'PDQ 'RST 'FILES)** sets up **RSTCOMS** to be a copy of **PDQCOMS**, changes things like **(VARS * PDQVARS)** to be **(VARS * RSTVARS)** in **RSTCOMS**, and performs a **MAKEFILE** on **RST** such that the appropriate definitions get copied from **PDQ**.

COPYDEF disables the **NOCOPY** option of **GETDEF**, so *NEW* will always have a *copy* of the definition of *OLD*.

Note: **COPYDEF** substitutes *NEW* for *OLD* throughout the definition of *OLD*. This is usually the right thing to do, but in some cases, e.g., where the old name appears within a quoted expression but was not used in the same context, the user must re-edit the definition.

(DELDEF *NAME TYPE*) [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(SHOWDEF *NAME TYPE FILE*) [Function]

Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows the user how *NAME* would be written to a file. Used by **ADDTOFILES?** (page 17.13).

(EDITDEF *NAME TYPE SOURCE EDITCOMS*) [Function]

Edits the definition of *NAME* as a *TYPE*. Essentially performs

```
(PUTDEF NAME TYPE
  (EDITE (GETDEF NAME TYPE SOURCE)
    EDITCOMS))
```

(SAVEDEF *NAME TYPE DEFINITION*) [Function]

Sets the "saved" definition of *NAME* as a *TYPE* to *DEFINITION*. If *DEFINITION* = **NIL**, the current definition of *NAME* is saved.

If *TYPE* = **FNS** (or **NIL**), the function definition is saved on *NAME*'s property list under the property **EXPR**, or **CODE** (depending on the **FNTYP** of the function definition). If **(GETD** *NAME*) is non-**NIL**, but **(FNTYP** *FN*) = **NIL**, **SAVEDEF** saves the definition on the property name **LIST**. This can happen if a function was

somehow defined with an illegal expr definition, such as **(LAMMMMDA (X) ...)**.

If *TYPE* = **VARS**, the definition is stored as the value of the **VALUE** property of *NAME*. For other types, the definition is stored in an internal data structure, from where it can be retrieved by **GETDEF** or **UNSAVEDEF**.

(UNSAVEDEF *NAME TYPE* —**)** [Function]

Restores the "saved" definition of *NAME* as a *TYPE*, making it be the current definition. Returns *PROP*.

If *TYPE* = **FNS** (or **NIL**), **UNSAVEDEF** unsaves the function definition from the **EXPR** property if any, else **CODE**, and returns the property name used. **UNSAVEDEF** also recognizes *TYPE* = **EXPR**, **CODE**, or **LIST**, meaning to unsave the definition only from the corresponding property only.

If **DFNFLG** is not **T** (see page 10.10), the current definition of *NAME*, if any, is saved using **SAVEDEF**. Thus one can use **UNSAVEDEF** to switch back and forth between two definitions.

(LOADDEF *NAME TYPE SOURCE***)** [Function]

Equivalent to **(PUTDEF** *NAME TYPE* **(GETDEF** *NAME TYPE SOURCE***))**. **LOADDEF** is essentially a generalization of **LOADFNS**, e.g. it enables loading a single record declaration from a file. Note that **(LOADDEF** *FN***)** will give *FN* an **EXPR** definition, either obtained from its property list or a file, unless it already has one.

(CHANGECALLERS *OLD NEW TYPES FILES METHOD***)** [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, **(CHANGECALLERS 'NLSETQ 'ERSETQ)** will change all calls to **NLSETQ** to be calls to **ERSETQ**. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD* **->** *NEW* automatically, the user is allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, the user is asked whether he wants to make the replacement. If he responds with anything except **Yes** or **No**, the editor is invoked on the expression containing that occurrence.

There are two different methods for determining which functions are to be examined. If *METHOD* = **EDITCALLERS**, **EDITCALLERS** is used to search *FILES* (see page 16.74). If *METHOD* = **MASTERSCOPE**, then the Masterscope database is used instead. *METHOD* = **NIL** defaults to **MASTERSCOPE** if the

value of the variable **DEFAULTRENAMEMETHOD** is **MASTERSCOPE** and a Masterscope database exists, otherwise it defaults to **EDITCALLERS**.

(RENAME *OLD NEW TYPES FILES METHOD***)** [Function]

First performs **(COPYDEF** *OLD NEW TYPE***)** for all *TYPE* inside *TYPES*. It then calls **CHANGECALLERS** to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with **DELDEF**. For example, if the user has a function **FOO** which he now wishes to call **FIE**, he simply performs **(RENAME 'FOO 'FIE)**, and **FIE** will be given **FOO**'s definition, and all places that **FOO** are called will be changed to call **FIE** instead.

METHOD is interpreted the same as the *METHOD* argument to **CHANGECALLERS**, above.

(COMPARE *NAME1 NAME2 TYPE SOURCE1 SOURCE2***)** [Function]

Compares the definition of *NAME1* with that of *NAME2*, by calling **COMPARELISTS** (page 3.19) on **(GETDEF** *NAME1 TYPE SOURCE1***)** and **(GETDEF** *NAME2 TYPE SOURCE2***)**, which prints their differences on the terminal.

For example, if the current value of the variable **A** is **(A B C (D E F) G)**, and the value of the variable **B** on the file **<lisp>FOO** is **(A B C (D F E) G)**, then:

←(COMPARE 'A 'B 'VARS 'CURRENT '<lisp>FOO)
A from CURRENT and B from <lisp>TEST differ:
(E -> F) (F -> E)
T

(COMPAREDEFS *NAME TYPE SOURCES***)** [Function]

Calls **COMPARELISTS** (page 3.19) on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES* (interpreted as a list of source specifications).

17.8.2 Defining New File Package Types

All manipulation of typed definitions in the file package is done using the type-independent functions **GETDEF**, **PUTDEF**, etc. Therefore, to define a new file package type, it is only necessary to specify (via the function **FILEPKGTYPE**) what these functions should do when dealing with a typed definition of the new type. Each file package type has the following properties, whose values are functions or lists of functions:

Note: These functions are defined to take a *TYPE* argument so that the user may have the same function for more than one type.

GETDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by **GETDEF** (page 17.25), which passes its *OPTIONS* argument.

If there is no **GETDEF** property, a file package command for dumping *NAME* is created (by **MAKENEWCOM**). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file **FILEPKG.SCRATCH** (in Interlisp-D, this file is created on the **{CORE}** device). This expression is then read back in and returned as the current definition.

Note: In some situations, the function **HASDEF** (page 17.26) needs to call **GETDEF** to determine whether a definition exists. In this case, *OPTIONS* will include the litatom **HASDEF**, and it is permissable for a **GETDEF** function to return **T** or **NIL**, rather than creating a complex structure which will not be used.

NULLDEF [File Package Type Property]

The value of the **NULLDEF** property is returned by **GETDEF** (page 17.25) when there is no definition and the **NOERROR** option is supplied. For example, the **NULLDEF** of **VARS** is **NOBIND**.

FILEGETDEF [File Package Type Property]

This enables the user to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by **GETDEF** (page 17.25). Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by **GETDEF** when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

CANFILEDEF [File Package Type Property]

If the value of this property is non-**NIL**, this indicates that definitions of this file package type are not loaded when a file is loaded with **LOADFROM** (page 17.8). The default is **NIL**. Initially, only **FNS** has this property set to non-**NIL**.

PUTDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by **PUTDEF** (page 17.26).

HASDEF [File Package Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *SOURCE*, which should return **(OR** *NAME* **T)** if *NAME* is the name

of something of type *TYPE*. *SOURCE* is as interpreted by **HASDEF** (page 17.26), which uses this property.

EDITDEF [File Package Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *SOURCE*, and *EDITCOMS*, which should edit the definition of *NAME* as a type *TYPE* from the source *SOURCE*, interpreting the edit commands *EDITCOMS*. If sucessful, should return *NAME* (or a spelling-corrected *NAME*). If it returns **NIL**, the "default" editor is called. Used by **EDITDEF** (page 17.27).

DELDEF [File Package Type Property]

Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of *NAME* as a *TYPE* that is currently in effect. Used by **DELDEF** (page 17.27).

NEWCOM [File Package Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file package command to dump *NAME*, an object of type *TYPE*. The function should return the new file package command. Used by **ADDTOFILE** and **SHOWDEF**.

If *LISTNAME* is non-**NIL**, this means that the user specified *LISTNAME* as the filevar in his interaction with **ADDTOFILES?** (see page 17.44).

If no **NEWCOM** is specified, the default is to call **DEFAULTMAKENEWCOM**, which will construct and return a command of the form (*TYPE NAME*). **DEFAULTMAKENEWCOM** can be advised or redefined by the user.

WHENCHANGED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see **MARKASCHANGED**, page 17.17). Used for various applications, e.g. when an object of type **I.S.OPRS** changes, it is necessary to clear the corresponding translatons from **CLISPARRAY**.

The **WHENCHANGED** functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute **(RETFROM 'MARKASCHANGED)**.

Note: For backwards compatibility, the *REASON* argument passed to **WHENCHANGED** functions is either **T** (for **DEFINED**) and **NIL** (for **CHANGED**).

WHENFILED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED [File Package Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION [File Package Type Property]

Value is a string which describes instances of this type. For example, for type **RECORDS**, the value of **DESCRIPTION** is the string **"record declarations"**.

The function **FILEPKGTYPE** is used to define new file package types, or to change the properties of existing types. Note that it is possible to redefine the attributes of system file package types, such as **FNS** or **PROPS**.

(FILEPKGTYPE *TYPE* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N**)** [NoSpread Function]

Nospread function for defining new file package types, or changing properties of existing file package types. $PROP_i$ is one of the property names given above; VAL_i is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP***)** returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE***)** returns an alist of all of the defined properties of *TYPE*, using the property names as keys.

Note: Specifying *TYPE* as the litatom **TYPE** can be used to define one file package type as a synonym of another. For example, **(FILEPKGTYPE 'R 'TYPE 'RECORDS)** defines **R** as a synonym for the file package type **RECORDS**.

17.9 File Package Commands

The basic mechanism for creating symbolic files is the function **MAKEFILE** (page 17.10). For each file, the file package has a data structure known as the "filecoms", which specifies what typed descriptions are contained in the file. A filecoms is a list of file package commands, each of which specifies objects of a certain file package type which should be dumped. For example, the filecoms

```
( (FNS FOO)
  (VARS FOO BAR BAZ)
```

(RECORDS XYZZY))

has a **FNS**, a **VARS**, and a **RECORDS** file package command. This filecoms specifies that the function definition for **FOO**, the variable values of **FOO**, **BAR**, and **BAZ**, and the record declaration for **XYZZY** should be dumped.

By convention, the filecoms of a file *X* is stored as the value of the litatom *X***COMS**. For example, **(MAKEFILE 'FOO.;27)** will use the value of **FOOCOMS** as the filecoms. This variable can be directly manipulated, but the file package contains facilities which make constructing and updating filecoms easier, and in some cases automatic (See page 17.48).

A file package command is an instruction to **MAKEFILE** to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file package types and file package commands; for each file package type, there is a file package command which is used for writing objects of that type to a file, and each file package command is used to write objects of a particular type. However, in some cases, the same file package type can be dumped by several different file package commands. For example, the file package commands **PROP**, **IFPROP**, and **PROPS** all dump out objects with the file package type **PROPS**. This means if the user changes an object of file package type **PROPS** via **EDITP**, a typed-in call to **PUTPROP**, or via an explicit call to **MARKASCHANGED**, this object can be written out with any of the above three commands. Thus, when the file package attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file package commands **PROP**, **IFPROP**, and **PROPS**, to see if the corresponding atom and property are specified. It is also permissible for a single file package command to dump several different file package types. For example, the user can define a file package command which dumps both a function definition and its macro. Conversely, some file package comands do not dump any file package types at all, such as the **E** command.

For each file package command, the file package must be able to determine what typed definitions the command will cause to be printed so that the file package can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file package type, the file package must be able to construct a command that will print out an object of that type. In other words, the file package must be able to map file package commands into file package types, and vice versa. Information can be provided to the file package about a particular file package command via the function **FILEPKGCOM** (page 17.47), and information about a particular file package type via the function **FILEPKGTYPE** (page 17.32). In the absence of other information, the default is simply that a file

package command of the form (*X NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X NAME*).

If a file package function is given a command or type that is not defined, it attempts spelling correction using **FILEPKGCOMSPLST** as a spelling list (unless **DWIMFLG** or **NOSPELLFLG** = **NIL**; see page 20.13). If successful, the corrected version of the list of file package commands is written (again) on the output file, since at this point, the uncorrected list of file package commands would already have been printed on the output file. When the file is loaded, this will result in *FILE***COMS** being reset, and may cause a message to be printed, e.g., **(FOOCOMS RESET)**. The value of **FOOCOMS** would then be the corrected version. If the spelling correction is unsuccessful, the file package functions generate an error, **BAD FILE PACKAGE COMMAND**.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file package command is documented in the following sections.

(USERMACROS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each litatom *LITATOM*$_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of *LITATOM*$_i$ to **USERMACROS**, and adds the names of the commands to the appropriate spelling lists.

If *LITATOM*$_i$ is not a user macro, a warning message **"no EDIT MACRO for** *LITATOM*$_i$**"** is printed.

17.9.1 Functions and Macros

(FNS *FN*$_1$ ... *FN*$_N$**)** [File Package Command]

Writes a **DEFINEQ** expression with the function definitions of *FN*$_1$ ... *FN*$_N$.

The user should never print a **DEFINEQ** expression directly onto a file himself (by using the ~ file package command, for example), because **MAKEFILE** generates the filemap of function definitions from the **FNS** file package commands (see page 17.55).

(ADVISE *FN*$_1$ ... *FN*$_N$**)** [File Package Command]

For each function *FN*$_i$, writes expressions to reinstate the function to its advised state when the file is loaded. See page 15.9.

Note: When advice is applied to a function programmatically or by hand, it is additive. That is, if a function already has some advice, further advice is added to the already-existing advice. However, when advice is applied to a function as a result of loading a file with an **ADVISE** file package command, the new advice replaces any earlier advice. **ADVISE** works this way to prevent problems with loading different versions of the same advice. If the user really wants to apply additive advice, a file package command such as **(P (ADVISE ...))** should be used (see page 17.40).

(ADVICE FN_1 ... FN_N**)** [File Package Command]

For each function FN_i, writes a **PUTPROPS** expression which will put the advice back on the property list of the function. The user can then use **READVISE** (page 15.12) to reactivate the advice.

(MACROS $LITATOM_1$ ... $LITATOM_N$**)** [File Package Command]

Each $LITATOM_i$ is a litatom with a **MACRO** definition (and/or a **DMACRO**, **10MACRO**, etc.). Writes out an expression to restore all of the macro properties for each $LITATOM_i$, embedded in a **DECLARE: EVAL@COMPILE** so the macros will be defined when the file is compiled. See page 10.21.

17.9.2 Variables

(VARS VAR_1 ... VAR_N**)** [File Package Command]

For each VAR_i, writes an expression to set its top level value when the file is loaded. If VAR_i is atomic, **VARS** writes out an expression to set VAR_i to the top-level value it had at the time the file was written. If VAR_i is non-atomic, it is interpreted as (*VAR FORM*), and **VARS** write out an expression to set *VAR* to the value of *FORM* (evaluated when the file is loaded).

VARS prints out expressions using **RPAQQ** and **RPAQ**, which are like **SETQQ** and **SETQ** except that they also perform some special operations with respect to the file package (see page 17.54).

Note: **VARS** cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a litatom which represents the array is dumped in the file instead of the array itself. The **HORRIBLEVARS** file package command (page 17.36) provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(INITVARS VAR_1 ... VAR_N**)** [File Package Command]

INITVARS is used for initializing variables, setting their values only when they are currently **NOBIND**. A variable value defined in an **INITVARS** command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

The format of an **INITVARS** command is just like **VARS**. The only difference is that if VAR_i is atomic, the current value is not dumped; instead **NIL** is defined as the initialization value. Therefore, **(INITVARS FOO (FUM 2))** is the same as **(VARS (FOO NIL)(FUM 2))**, if **FOO** and **FUM** are both **NOBIND**.

INITVARS writes out an **RPAQ?** expression on the file instead of **RPAQ** or **RPAQQ**.

(ADDVARS (VAR_1 . LST_1**)** ... **(**VAR_N . LST_N**))** [File Package Command]

For each **(**VAR_i . LST_i**)**, writes an **ADDTOVAR** (page 17.54) to add each element of LST_i to the list that is the value of VAR_i at the time the file is loaded. The new value of VAR_i will be the union of its old value and LST_i. If the value of VAR_i is **NOBIND**, it is first set to **NIL**.

For example, **(ADDVARS (DIRECTORIES LISP LISPUSERS))** will add **LISP** and **LISPUSERS** to the value of **DIRECTORIES**.

If LST_i is not specified, VAR_i is initialized to **NIL** if its current value is **NOBIND**. In other words, **(ADDVARS (***VAR***))** will initialize *VAR* to **NIL** if *VAR* has not previously been set.

(APPENDVARS (VAR_1 . LST_1**)** ... **(**VAR_N . LST_N**))** [File Package Command]

The same as **ADDVARS**, except that the values are added to the end of the lists (using **APPENDTOVAR**, page 17.55), rather than at the beginning.

(UGLYVARS VAR_1 ... VAR_N**)** [File Package Command]

Like **VARS**, except that the value of each VAR_i may contain structures for which **READ** is not an inverse of **PRINT**, e.g. arrays, readtables, user data types, etc. Uses **HPRINT** (page 25.17).

(HORRIBLEVARS VAR_1 ... VAR_N**)** [File Package Command]

Like **UGLYVARS**, except structures may also contain circular pointers. Uses **HPRINT** (page 25.17). The values of VAR_1 ... VAR_N are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over

HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, **UGLYVARS** should be used instead of **HORRIBLEVARS**.

(ALISTS (*VAR*$_1$ *KEY*$_1$ *KEY*$_2$ ...) ... (*VAR*$_N$ *KEY*$_3$ *KEY*$_4$ ...)) [File Package Command]

VAR$_i$ is a variable whose value is an association list, such as **EDITMACROS**, **BAKTRACELST**, etc. For each *VAR*$_i$, **ALISTS** writes out expressions which will restore the values associated with the specified keys. For example, **(ALISTS (BREAKMACROS BT BTV))** will dump the definition for the **BT** and **BTV** commands on **BREAKMACROS**.

Some association lists (**USERMACROS**, **LISPXMACROS**, etc.) are used to implement other file package types, and they have their own file package commands.

(SPECVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

(LOCALVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

(GLOBALVARS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

Outputs the corresponding compiler declaration embedded in a **DECLARE: DOEVAL@COMPILE DONTCOPY**. See page 18.5.

(CONSTANTS *VAR*$_1$ ... *VAR*$_N$) [File Package Command]

Like **VARS**, for each *VAR*$_i$ writes an expression to set its top level value when the file is loaded. Also writes a **CONSTANTS** expression to declare these variables as constants (see page 18.8). Both of these expressions are wrapped in a **(DECLARE: EVAL@COMPILE ...)** expression, so they can be used by the compiler.

Like **VARS**, *VAR*$_i$ can be non-atomic, in which case it is interpreted as **(*VAR FORM*)**, and passed to **CONSTANTS** (along with the variable being initialized to *FORM*).

17.9.3 Litatom Properties

(PROP *PROPNAME LITATOM*$_1$ ... *LITATOM*$_N$) [File Package Command]

Writes a **PUTPROPS** expression to restore the value of the *PROPNAME* property of each litatom *LITATOM*$_i$ when the file is loaded.

If *PROPNAME* is a list, expressions will be written for each property on that list. If *PROPNAME* is the litatom **ALL**, the values of all user properties (on the property list of each *LITATOM*$_i$) are

saved. **SYSPROPS** is a list of properties used by system functions. Only properties *not* on that list are dumped when the **ALL** option is used.

If $LITATOM_i$ does not have the property *PROPNAME* (as opposed to having the property with value **NIL**), a warning message **"NO** *PROPNAME* **PROPERTY FOR** $LITATOM_i$**"** is printed. The command **IFPROP** can be used if it is not known whether or not an atom will have the corresponding property.

(IFPROP *PROPNAME* $LITATOM_1$ ... $LITATOM_N$**)** [File Package Command]

Same as the **PROP** file package command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if **FOO1** has property **PROP1** and **PROP2**, **FOO2** has **PROP3**, and **FOO3** has property **PROP1** and **PROP3**, then **(IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3)** will save only those five property values.

(PROPS ($LITATOM_1$ $PROPNAME_1$**)** ... **(**$LITATOM_N$ $PROPNAME_N$**))** [File Package Command]

Similar to **PROP** command. Writes a **PUTPROPS** expression to restore the value of $PROPNAME_i$ for each $LITATOM_i$ when the file is loaded.

As with the **PROP** command, if $LITATOM_i$ does not have the property *PROPNAME* (as opposed to having the property with **NIL** value), a warning message **"NO** $PROPNAME_i$ **PROPERTY FOR** $LITATOM_i$**"** is printed.

17.9.4 Miscellaneous File Package Commands

(RECORDS REC_1 ... REC_N**)** [File Package Command]

Each REC_i is the name of a record (see page 8.1). Writes expressions which will redeclare the records when the file is loaded.

(INITRECORDS REC_1 ... REC_N**)** [File Package Command]

Similar to **RECORDS**, **INITRECORDS** writes expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records. However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(LISPXMACROS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each *LITATOM*$_i$ is defined on **LISPXMACROS** or **LISPXHISTORYMACROS** (see page 13.23). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to **LISPXCOMS**

(I.S.OPRS *OPR*$_1$ ... *OPR*$_N$**)** [File Package Command]

Each *OPR*$_i$ is the name of a user-defined i.s.opr (see page 9.20). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(RESOURCES *RESOURCE*$_1$ ... *RESOURCE*$_N$**)** [File Package Command]

Each *RESOURCES*$_i$ is the name of a resource (see page 12.19). Writes expressions which will redeclare the resource when the file is loaded.

(INITRESOURCES *RESOURCE*$_1$ ... *RESOURCE*$_N$**)** [File Package Command]

Parallel to **INITRECORDS** (page 17.38), **INITRESOURCES** writes expressions on a file to perform whatever initialization/allocation is necessary for the indicated resources, without writing the resource declaration itself.

(COURIERPROGRAMS *NAME*$_1$ ... *NAME*$_N$**)** [File Package Command]

Each *NAME*$_i$ is the name of a Courier program (see page 31.15). Writes expressions which will redeclare the Courier program when the file is loaded.

(TEMPLATES *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each *LITATOM*$_i$ is a litatom which has a Masterscope template (see page 19.21). Writes expressions which will restore the templates when the file is loaded.

(FILES *FILE*$_1$ ... *FILE*$_N$**)** [File Package Command]

Used to specify auxiliary files to be loaded in when the file is loaded. Dumps an expression calling **FILESLOAD** (page 17.9), with *FILE*$_1$ ... *FILE*$_N$ as the arguments. **FILESLOAD** interprets *FILE*$_1$ ... *FILE*$_N$ as files to load, possibly interspersed with lists used to specify certain loading options.

(FILEPKGCOMS *LITATOM*$_1$ ... *LITATOM*$_N$**)** [File Package Command]

Each litatom *LITATOM*$_i$ is either the name of a user-defined file package command or a user-defined file package type (or both). Writes expressions which will restore each command/type.

If $LITATOM_i$ is not a file package command or type, a warning message **"no FILE PACKAGE COMMAND for** $LITATOM_i$**"** is printed.

(* . *TEXT)* [File Package Command]

Used for inserting comments in a file. The file package command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another *, a form-feed is printed on the file before the comment.

(P EXP_1 ... EXP_N**)** [File Package Command]

Writes each of the expressions EXP_1 ... EXP_N on the output file, where they will be evaluated when the file is loaded.

(E $FORM_1$ ... $FORM_N$**)** [File Package Command]

Each of the forms $FORM_1$ ... $FORM_N$ is evaluated at *output* time, when **MAKEFILE** interpretes this file package command.

(COMS COM_1 ... COM_N**)** [File Package Command]

Each of the commands COM_1 ... COM_N is interpreted as a file package command.

(ORIGINAL COM_1 ... COM_N**)** [File Package Command]

Each of the commands COM_i will be interpreted as a file package command without regard to any file package macros (as defined by the **MACRO** property of the **FILEPKGCOM** function, page 17.47). Useful for redefining a built-in file package command in terms of itself.

Note that some of the "built-in" file package commands are defined by file package macros, so interpreting them (or new user-defined file package commands) with **ORIGINAL** will fail. **ORIGINAL** was never intended to be used outside of a file package command macro.

17.9.5 DECLARE:

(DECLARE: . *FILEPKGCOMS/FLAGS)* [File Package Command]

Normally expressions written onto a symbolic file are (1) evaluated when loaded; (2) copied to the compiled file when the symbolic file is compiled (see page 18.1); and (3) not evaluated at compile time. **DECLARE:** allows the user to override these defaults.

FILEPKGCOMS/FLAGS is a list of file package commands, possibly interspersed with "tags". The output of those file package commands within *FILEPKGCOMS/FLAGS* is embedded in a **DECLARE:** expression, along with any tags that are specified. For example, **(DECLARE: EVAL@COMPILE DONTCOPY (FNS ...) (PROP ...))** would produce **(DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ...) (PUTPROPS ...))**. **DECLARE:** is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the **DONTEVAL@LOAD** tag.

DECLARE: expressions are specially processed by the compiler. For the purposes of compilation, **DECLARE:** has two principal applications: (1) to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or (2) to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in **CDR** of a **DECLARE:** form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the **DECLARE:** by means of the tags **DONTCOPY**, **EVAL@COMPILE**, etc.

The tags are:

EVAL@LOAD
DOEVAL@LOAD Evaluate the following forms when the file is loaded (unless overridden by **DONTEVAL@LOAD**).

DONTEVAL@LOAD Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. **... EVAL@LOADWHEN T ...** is equivalent to **... EVAL@LOAD ...**

COPY
DOCOPY When compiling, copy the following forms into the compiled file.

DONTCOPY When compiling, do not copy the following forms into the compiled file.

Note: If the file package commands following **DONTCOPY** include record declarations for datatypes, or records with initialization forms, it is necessary to include a **INITRECORDS** file package command (page 17.38) outside of the **DONTCOPY** form so that the initialization information is copied. For example, if **FOO** was defined as a datatype,

```
(DECLARE: DONTCOPY (RECORDS FOO))
(INITRECORDS FOO)
```

would copy the data type declaration for **FOO**, but would not copy the whole record declaration.

COPYWHEN When compiling, if the next form evaluates to non-**NIL**, copy the following forms into the compiled file.

EVAL@COMPILE
DOEVAL@COMPILE When compiling, evaluate the following forms.

DONTEVAL@COMPILE When compiling, do not evaluate the following forms.

EVAL@COMPILEWHEN When compiling, if the next form evaluates to non-**NIL**, evaluate the following forms.

FIRST For expressions that are to be copied to the compiled file, the tag **FIRST** can be used to specify that the following expressions in the **DECLARE:** are to appear at the front of the compiled file, before anything else except the **FILECREATED** expressions (see page 17.51). For example, **(DECLARE: COPY FIRST (P (PRINT** *MESS1* **T)) NOTFIRST (P (PRINT** *MESS2* **T)))** will cause **(PRINT** *MESS1* **T)** to appear first in the compiled file, followed by any functions, then **(PRINT** *MESS2* **T)**.

NOTFIRST Reverses the effect of **FIRST**.

The value of **DECLARETAGSLST** is a list of all the tags used in **DECLARE:** expressions. If a tag not on this list appears in a **DECLARE:** file package command, spelling correction is performed using **DECLARETAGSLST** as a spelling list.

Note that the function **LOADCOMP** (page 17.8) provides a convenient way of obtaining information from the **DECLARE:** expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(BLOCKS $BLOCK_1$ ... $BLOCK_N$**)** [File Package Command]

For each $BLOCK_i$, writes a **DECLARE:** expression which the block compile functions interpret as a block declaration. See page 18.17.

17.9.6 Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the **DECLARE:** file package command with tag **DONTCOPY** (page 17.40), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into

Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The **EXPORT** mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file package commands in the **EXPORT** file package command. Thereafter, **GATHEREXPORTS** can be used to make a single file containing all the exports.

(EXPORT COM_1 ... COM_N**)** [File Package Command]

This command is used for "exporting" definitions. Like **COM**, each of the commands COM_1 ... COM_N is interpreted as a file package command. The commands are also flagged in the file as being "exported" commands, for use with **GATHEREXPORTS** (see page 17.43).

(GATHEREXPORTS *FROMFILES TOFILE FLG***)** [Function]

FROMFILES is a list of files containing **EXPORT** commands. **GATHEREXPORTS** extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = **EVAL**, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(IMPORTFILE *FILE RETURNFLG***)** [Function]

If *RETURNFLG* is **NIL**, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is **T**, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(CHECKIMPORTS *FILES NOASKFLG***)** [Function]

Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were

taken (**GATHEREXPORTS** and **IMPORTFILE** note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, the user is asked for permission to **IMPORTFILE** each such file. If *NOASKFLG* is non-**NIL**, **IMPORTFILE** is performed without asking.

For example, suppose file **FOO** contains records **R1**, **R2**, and **R3**, macros **BAR** and **BAZ**, and constants **CON1** and **CON2**. If the definitions of **R1**, **R2**, **BAR**, and **BAZ** are needed by files other than **FOO**, then the file commands for **FOO** might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
  (EXPORT (RECORDS R1 R2)
      (MACROS BAR BAZ))
  (RECORDS R3)
  (CONSTANTS BAZ))
```

None of the commands inside this **DECLARE:** would appear on **FOO**'s compiled file, but **(GATHEREXPORTS '(FOO) 'MYEXPORTS)** would copy the record definitions for **R1** and **R2** and the macro definitions for **BAR** and **BAZ** to the file **MYEXPORTS**.

17.9.7 FileVars

In each of the file package commands described above, if the litatom * follows the command type, the form following the *, i.e., **CADDR** of the command, is evaluated and its value used in executing the command, e.g., **(FNS * (APPEND FNS1 FNS2))**. When this form is a litatom, e.g. **(FNS * FOOFNS)**, we say that the variable is a "filevar". Note that **(COMS *** *FORM***)** provides a way of *computing* what should be done by **MAKEFILE**.

Example:

```
←(SETQ FOOFNS '(FOO1 FOO2 FOO3))
(FOO1 FOO2 FOO3)
←(SETQ FOOCOMS
  '( (FNS * FOOFNS)
     (VARS FIE)
     (PROP MACRO FOO1 FOO2)
     (P (MOVD 'FOO1 'FIE1))]
←(MAKEFILE 'FOO)
```

would create a file **FOO** containing:

```
(FILECREATED "time and date the file was made" . "other
information")
(PRETTYCOMPRINT FOOCOMS)
(RPAQQ FOOCOMS ((FNS * FOOFNS) ...)
(RPAQQ FOOFNS (FOO1 FOO3 FOO3))
```

(DEFINEQ "*definitions of* FOO1, FOO2, *and* FOO3")
(RPAQQ FIE "*value of* FIE")
(PUTPROPS FOO1 MACRO *PROPVALUE*)
(PUTPROPS FOO2 MACRO *PROPVALUE*)
(MOVD (QUOTE FOO1) (QUOTE FIE1))
STOP

Note: For the **PROP** and **IFPROP** commands (page 17.37), the * follows the property name instead of the command, e.g., **(PROP MACRO * FOOMACROS)**. Also, in the form **(* * comment ...)**, the word **comment** is not treated as a filevar.

17.9.8 Defining New File Package Commands

A file package command is defined by specifying the values of certain properties. The user can specify the various attributes of a file package command for a new command, or respecify them for an existing command. The following properties are used:

MACRO [File Package Command Property]

Defines how to dump the file package command. Used by **MAKEFILE**. Value is a pair (*ARGS* . *COMS*). The "arguments" to the file package command are substituted for *ARGS* throughout *COMS*, and the result treated as a list of file package commands. For example, following **(FILEPKGCOM 'FOO 'MACRO '((X Y) . *COMS*))**, the file package command **(FOO A B)** will cause **A** to be substituted for **X** and **B** for **Y** throughout *COMS*, and then *COMS* treated as a list of commands.

The substitution is carried out by **SUBPAIR** (page 3.14), so that the "argument list" for the macro can also be atomic. For example, if **(X . *COMS*)** was used instead of **((X Y) . *COMS*)**, then the command **(FOO A B)** would cause **(A B)** to be substituted for **X** throughout *COMS*.

Note: Filevars are evaluated *before* substitution. For example, if the litatom * follows *NAME* in the command, **CADDR** of the command is evaluated substituting in *COMS*.

ADD [File Package Command Property]

Specifies how (if possible) to add an instance of an object of a particular type to a given file package command. Used by **ADDTOFILE**. Value is *FN*, a function of three arguments, *COM*, a file package command **CAR** of which is **EQ** to *COMMANDNAME*, *NAME*, a typed object, and *TYPE*, its type. *FN* should return **T** if it (undoably) adds *NAME* to *COM*, **NIL** if not. If no **ADD** property is specified, then the default is (1) if **(CAR *COM*) = *TYPE*** and **(CADR *COM*) = ***, and **(CADDR *COM*)** is a filevar (i.e. a literal atom), add

NAME to the value of the filevar, or (2) if (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) is not *, add *NAME* to (**CDR** *COM*).

Actually, the function is given a fourth argument, *NEAR*, which if non-**NIL**, means the function should try to add the item after *NEAR*. See discussion of **ADDTOFILES?**, page 17.13.

DELETE [File Package Command Property]

Specifies how (if possible) to delete an instance of an object of a particular type from a given file package command. Used by **DELFROMFILES**. Value is *FN*, a function of three arguments, *COM*, *NAME*, and *TYPE*, same as for **ADD**. *FN* should return **T** if it (undoably) deletes *NAME* from *COM*, **NIL** if not. If no **DELETE** property is specified, then the default is (1) (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) = *, and (**CADDR** *COM*) is a filevar (i.e. a literal atom), and *NAME* is contained in the value of the filevar, then remove *NAME* from the filevar, or (2) if (**CAR** *COM*) = *TYPE* and (**CADR** *COM*) is not *, and *NAME* is contained in (**CDR** *COM*), then remove *NAME* from (**CDR** *COM*).

If *FN* returns the value of **ALL**, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS [File Package Command Property]

CONTAIN [File Package Command Property]

Determines whether an instance of an object of a given type is contained in a given file package command. Used by **WHEREIS** and **INFILECOMS?**. Value is *FN*, a function of three arguments, *COM*, a file package command **CAR** of which is **EQ** to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is **NIL**, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is **T**, *FN* should return **T** if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than **T** or **NIL**, return **T** if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

Note that it is sufficient for the **CONTENTS** function to simply return the list of items of type *TYPE* in command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a **CONTENTS** property is specified and the corresponding function application returns **NIL** and (**CAR** *COM*) = *TYPE*, then the operation indicated by *NAME* is performed (1) on the value

of (**CADDR** *COM*), if (**CADR** *COM*) = *, otherwise (2) on (**CDR** *COM*). In other words, by specifying a **CONTENTS** property that returns **NIL**, e.g. the function **NILL**, the user specifies that a file package command of name **FOO** produces objects of file package type **FOO** and only objects of type **FOO**.

If the **CONTENTS** property is not provided, the command is simply expanded according to its **MACRO** definition, and each command on the resulting command list is then interrogated.

Note that if *COMMANDNAME* is a file package command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of **CONS**es and garbage collections. By informing the file package as to what this command actually does and does not produce via the **CONTENTS** property, this expansion is avoided. For example, suppose the user has a file package command called **GRAMMARS** which dumps various property lists but no functions. The file package could ignore this command when seeking information about **FNS**.

The function **FILEPKGCOM** is used to define new file package commands, or to change the properties of existing commands. Note that it is possible to redefine the attributes of system file package commands, such as **FNS** or **PROPS**, and to cause unpredictable results.

(FILEPKGCOM *COMMANDNAME* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N**)** [NoSpread Function]

Nospread function for defining new file package commands, or changing properties of existing file package commands. $PROP_i$ is one of of the property names described above; VAL_i is the value to be given that property of the file package command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME PROP***)** returns the value of the property *PROP*, without changing it.

(FILEPKGCOM *COMMANDNAME***)** returns an alist of all of the defined properties of *COMMANDNAME*, using the property names as keys.

Note: Specifying *TYPE* as the litatom **COM** can be used to define one file package command as a synonym of another. For example, **(FILEPKGCOM 'INITVARIABLES 'COM 'INITVARS)** defines **INITVARIABLES** as a synonym for the file package command **INITVARS**.

17.10 Functions for Manipulating File Command Lists

The following functions may be used to manipulate filecoms. The argument *COMS* does *not* have to correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user defined file package command.

Note: The following functions will accept a file package command as a valid value for their *TYPE* argument, even if it does not have a corresponding file package type. User-defined file package commands are expanded as necessary.

(INFILECOMS? *NAME TYPE COMS* —**)** [Function]

COMS is a list of file package commands, or a variable whose value is a list of file package commands. *TYPE* is a file package type. **INFILECOMS?** returns **T** if *NAME* of type *TYPE* is "contained" in *COMS*.

If *NAME* = **NIL, INFILECOMS?** returns a list of all elements of type *TYPE*.

If *NAME* = **T, INFILECOMS?** returns **T** if there are *any* elements of type *TYPE* in *COMS*.

(ADDTOFILE *NAME TYPE FILE NEAR LISTNAME***)** [Function]

Adds *NAME* of type *TYPE* to the file package commands for *FILE*. If *NEAR* is given and it is the name of an item of type *TYPE* already on *FILE*, then *NAME* is added to the command that dumps *NEAR*. If *LISTNAME* is given and is the name of a list of items of *TYPE* items on *FILE*, then *NAME* is added to that list. Uses **ADDTOCOMS** and **MAKENEWCOM**. Returns *FILE*. **ADDTOFILE** is undoable.

(DELFROMFILES *NAME TYPE FILES***)** [Function]

Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-**NIL** litatom, **(LIST** *FILES***)** is used. *FILES* = **NIL** defaults to **FILELST**. Returns a list of files from which *NAME* was actually removed. Uses **DELFROMCOMS**. **DELFROMFILES** is undoable.

Note: Deleting a function will also remove the function from any **BLOCKS** declarations in the filecoms.

(ADDTOCOMS *COMS NAME TYPE NEAR LISTNAME***)** [Function]

Adds *NAME* as a *TYPE* to *COMS*, a list of file package commands or a variable whose value is a list of file package commands. Returns **NIL** if **ADDTOCOMS** was unable to find a command appropriate for adding *NAME* to *COMS*. *NEAR* and *LISTNAME*

are described in the discussion of **ADDTOFILE**. **ADDTOCOMS** is undoable.

Note that the exact algorithm for adding commands depends the particular command itself. See discussion of the **ADD** property, in the description of **FILEPKGCOM**, page 17.47.

Note: **ADDTOCOMS** will not attempt to add an item to any command which is inside of a **DECLARE:** unless the user specified a specific name via the **LISTNAME** or **NEAR** option of **ADDTOFILES?**.

(DELFROMCOMS *COMS NAME TYPE***)** [Function]

Deletes *NAME* as a *TYPE* from *COMS*. Returns **NIL** if **DELFROMCOMS** was unable to modify *COMS* to delete *NAME*. **DELFROMCOMS** is undoable.

(MAKENEWCOM *NAME TYPE* — —**)** [Function]

Returns a file package command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of **NEWCOM**, page 17.32.

(MOVETOFILE *TOFILE NAME TYPE FROMFILE***)** [Function]

Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with **DELFROMFILES** and **ADDTOFILE**).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in the user's environment.

(FILECOMSLST *FILE TYPE* —**)** [Function]

Returns a list of all objects of type *TYPE* in *FILE*.

(FILEFNSLST *FILE***)** [Function]

Same as **(FILECOMSLST** *FILE* **'FNS)**.

(FILECOMS *FILE TYPE***)** [Function]

Returns **(PACK*** *FILE* **(OR** *TYPE* **'COMS))**. Note that **(FILECOMS 'FOO)** returns the litatom **FOOCOMS**, not the value of **FOOCOMS**.

(SMASHFILECOMS *FILE***)** [Function]

Maps down **(FILECOMSLST** *FILE* **'FILEVARS)** and sets to **NOBIND** all filevars (see page 17.44), i.e. any variable used in a command of the form **(***COMMAND * VARIABLE***)**. Also sets **(FILECOMS** *FILE***)** to **NOBIND**. Returns *FILE*.

17.11 Symbolic File Format

The file package manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file package functions. In general, there is no reason for the user to manually change the contents of a symbolic file. However, in order to allow users to extend the file package, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(PRETTYDEF *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*)
[Function]

Writes a symbolic file in **PRETTYPRINT** format for loading, using **FILERDTBL** as its read table. **PRETTYDEF** returns the name of the symbolic file that was created.

PRETTYDEF operates under a **RESETLST** (see page 14.24), so if an error occurs, or a control-D is typed, all files that **PRETTYDEF** has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone. The **RESETLST** also means that any **RESETSAVE**s executed in the file package commands will also be protected.

PRTTYFNS is an optional list of function names. It is equivalent to including **(FNS** * *PRTTYFNS*) in the file package commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when **PRETTYDEF** did not use a list of file package commands, and should be specified as **NIL**.

PRTTYFILE is the name of the file on which the output is to be written. If *PRTTYFILE* = **NIL**, the primary output file is used. If *PRTTYFILE* is atomic the file is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of **PRETTYDEF**, and the primary output file is restored. Finally, if *PRTTYFILE* is a list, **CAR** of *PRTTYFILE* is assumed to be the file name, and is opened if not already open. In this case, the file is left open at end of **PRETTYDEF**.

PRTTYCOMS is a list of file package commands interpreted as described on page 17.32. If *PRTTYCOMS* is atomic, its top level value is used and an **RPAQQ** is written which will set that atom to the list of commands when the file is subsequently loaded. A **PRETTYCOMPRINT** expression (see below) will also be written which informs the user of the named atom or list of commands when the file is subsequently loaded. In addition, if any of the functions in the file are nlambda functions, **PRETTYDEF** will automatically print a **DECLARE:** expression suitable for informing the compiler about these functions, in case the user recompiles the file without having first loaded the nlambda functions (see page 18.8).

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see page 17.15). *REPRINTFNS* can be a list of functions to be prettyprinted, or **EXPRS**, meaning prettyprint all functions with **EXPR** definitions, or **ALL** meaning prettyprint all functions either defined as **EXPR**s, or with **EXPR** properties. Note that doing a remake with *REPRINTFNS* = **NIL** makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE* = **T** means to use most recent version (i.e., highest number) of *PRTTYFILE*, the second argument to **PRETTYDEF**. If *SOURCEFILE* cannot be found, **PRETTYDEF** prints the message "*FILE* **NOT FOUND, SO IT WILL BE WRITTEN ANEW**", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both **NIL**.

PRETTYDEF calls **PRETTYPRINT** with its second argument *PRETTYDEFLG* = **T**, so whenever **PRETTYPRINT** starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if **PRETTYPRINT** is given a litatom which is not defined as a function but is known to be on one of the files noticed by the file package, **PRETTYPRINT** will load in the definition (using **LOADFNS**) and print it. This is not done when **PRETTYPRINT** is called from **PRETTYDEF**.

(PRINTFNS *X* **—)** [Function]

X is a list of functions. **PRINTFNS** prettyprints a **DEFINEQ** epression that defines the functions to the primary output stream using the primary read table. Used by **PRETTYDEF** to implement the **FNS** file package command.

(PRINTDATE *FILE CHANGES***)** [Function]

Prints the **FILECREATED** expression at beginning of **PRETTYDEF** files. *CHANGES* used by the file package.

(FILECREATED *X***)** [NLambda NoSpread Function]

Prints a message (using **LISPXPRINT**) followed by the time and date the file was made, which is **(CAR** *X***)**. The message is the value of **PRETTYHEADER**, initially **"FILE CREATED"**. If **PRETTYHEADER = NIL**, nothing is printed. **(CDR** *X***)** contains information about the file, e.g., full name, address of file map, list of changed items, etc. **FILECREATED** also stores the time and date the file was made on the property list of the file under the property **FILEDATES** and performs other initialization for the file package.

(PRETTYCOMPRINT *X*) [NLambda Function]

Prints *X* (unevaluated) using **LISPXPRINT**, unless **PRETTYHEADER = NIL**.

PRETTYHEADER [Variable]

Value is the message printed by **FILECREATED**. **PRETTYHEADER** is initially **"FILE CREATED"**. If **PRETTYHEADER = NIL**, neither **FILECREATED** nor **PRETTYCOMPRINT** will print anything. Thus, setting **PRETTYHEADER** to **NIL** will result in "silent loads". **PRETTYHEADER** is reset to **NIL** during greeting (page 12.1).

(FILECHANGES *FILE TYPE*) [Function]

Returns a list of the changed objects of file package type *TYPE* from the **FILECREATED** expression of *FILE*. If *TYPE* = **NIL**, returns an alist of all of the changes, with the file package types as the **CAR**s of the elements..

(FILEDATE *FILE* —) [Function]

Returns the file date contained in the **FILECREATED** expression of *FILE*.

(LISPSOURCEFILEP *FILE*) [Function]

Returns a non-**NIL** value if *FILE* is an Interlisp source file, **NIL** otherwise.

17.11.1 Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the **FILECREATED** expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

(* Copyright (c) 1981 by Foo Bars Corporation)

Once a file has a copyright notice then every version will have a new copyright notice inserted into the file without user intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)

When a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner (if **COPYRIGHTFLG = T**). The user may specify one of the names

from **COPYRIGHTOWNERS**, or give one of the following responses:

(1) Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string

(2) Type a right-square-bracket, which specifies that the user really does not want a copyright notice.

(3) Type "**NONE**" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

```
((BBN "Bolt Beranek and Newman Inc.")
 (XEROX "Xerox Corporation"))
```

then for a new file **FOO** the following interaction will take place:

```
Do you want to Copyright FOO? Yes
Copyright owner: (user typed ?)
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[ - new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN
```

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG [Variable]

The value of **COPYRIGHTFLG** determines whether copyright information is maintained in files. Its value is interpreted as follows:

NIL The system will preserve old copyright information, but will not ask the user about copyrighting new files. This is the default value of **COPYRIGHTFLG**.

T When a file is made, if it has no copyright information, the system will ask the user to specify the copyright owner.

NEVER The system will neither prompt for new copyright information nor preserve old copyright information.

DEFAULT The value of **DEFAULTCOPYRIGHTOWNER** (below) is used for putting copyright information in files that don't have any other copyright. The prompt "**Copyright owner for file xx:**" will still be printed, but the default will be filled in immediately.

COPYRIGHTOWNERS [Variable]

COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to **ASKUSER** and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER [Variable]

If the user does not respond in **DWIMWAIT** seconds to the copyright query, the value of **DEFAULTCOPYRIGHTOWNER** is used.

17.11.2 Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file package variables.

(RPAQ *VAR VALUE*) [NLambda Function]

An nlambda function like **SETQ** that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(RPAQQ *VAR VALUE*) [NLambda Function]

An nlambda function like **SETQQ** that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(RPAQ? *VAR VALUE*) [NLambda Function]

Similar to **RPAQ**, except that it does nothing if *VAR* already has a top level value other than **NOBIND**. Returns *VALUE* if *VAR* is reset, otherwise **NIL**.

RPAQ, **RPAQQ**, and **RPAQ?** generate errors if *X* is not a litatom. All are affected by the value of **DFNFLG** (page 10.10). If **DFNFLG = ALLPROP** (and the value of *VAR* is other than **NOBIND**), instead of setting *X*, the corresponding value is stored on the property list of *VAR* under the property **VALUE**. All are undoable.

(ADDTOVAR *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Each X_i that is not a member of the value of *VAR* is added to it, i.e. after **ADDTOVAR** completes, the value of *VAR* will be **(UNION (LIST** X_1 X_2 ... X_N) *VAR*). **ADDTOVAR** is used by **PRETTYDEF** for implementing the **ADDVARS** command. It performs some file package related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(**APPENDTOVAR** *VAR* X_1 X_2 ... X_N) [NLambda NoSpread Function]

Similar to **ADDTOVAR**, except that the values are added to the end tof the list, rather than at the beginning.

(**PUTPROPS** *ATM* $PROP_1$ VAL_1 ... $PROP_N$ VAL_N) [NLambda NoSpread Function]

Nlambda nospread version of **PUTPROP** (none of the arguments are evaluated). For $i = 1...N$, puts property $PROP_i$, value VAL_i, on the property list of *ATM*. Performs some file package related operations, i.e., "notices" that the corresponding properties have been changed.

(**SAVEPUT** *ATM PROP VAL*) [Function]

Same as **PUTPROP**, but marks the corresponding property value as having been changed (used by the file package).

17.11.3 File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see **GETFILEPTR**, page 25.19) for each **DEFINEQ** expression in the file, the begin and end address for each function definition within the **DEFINEQ**, and the begin and end address for each compiled function.

MAKEFILE, **PRETTYDEF**, **LOADFNS**, **RECOMPILE**, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables **LOADFNS** to load selected function definitions simply by setting the file pointer to the corresponding address using **SETFILEPTR**, and then performing a single **READ**. Similarly, the file map is heavily used by the "remake" option of **MAKEFILE** (page 17.15): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by **MAKEFILE**, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that **PRETTYPRINT** *know* that it is printing a **DEFINEQ** expression. For this reason, the user should never print a **DEFINEQ** expression onto a file himself, but should instead always use the **FNS** file package command (page 17.34).

The file map is stored on the property list of the root name of the file, under the property **FILEMAP**. In addition, **MAKEFILE** writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the **FILECREATED**

expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file. In most cases, **LOAD** and **LOADFNS** do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with **BUILDMAPFLG = NIL**, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, **LOAD** and **LOADFNS** will build maps for a compiled file when it is loaded, and store it on the property **FILEMAP**. Similary, **LOADFNS** will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG [Variable]

Whenever a file is read by **LOAD** or **LOADFNS**, or written by **MAKEFILE**, a file map is automatically built unless **BUILDMAPFLG = NIL**. (**BUILDMAPFLG** is initially **T**.)

While building the map will not help the first reference to a file, it will help in future references. For example, if the user performs **(LOADFROM 'FOO)** where **FOO** does not contain a file map, the **LOADFROM** will be (slightly) slower than if **FOO** did contain a file map, but subsequent calls to **LOADFNS** for this version of **FOO** will be able to use the map that was built as the result of the **LOADFROM**, since it will be stored on **FOO**'s **FILEMAP** property.

USEMAPFLG [Variable]

If **USEMAPFLG = T** (the initial setting), the functions that use file maps will first check the **FILEMAP** property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a **FILECREATED** expression that also contains the address of a file map. If the file map is not on the **FILEMAP** property or in the file, a file map will be built (unless **BUILDMAPFLG = NIL**).

If **USEMAPFLG = NIL**, the **FILEMAP** property and the file will not be checked for the file map. This allows the user to recover in those cases where the file and its map for some reason do not agree. For example, if the user uses a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error **FILEMAP DOES NOT AGREE WITH CONTENTS OF** *FILE*. In such cases, the user can set **USEMAPFLG** to **NIL**, causing the map contained in the file to be ignored, and then reexecute the operation.