12.1 Greeting and Initialization Files

Many of the features of Interlisp are controlled by variables that the user can adjust to his or her own tastes. In addition, the user can modify the action of system functions in ways not specifically provided for by using **ADVISE** (page 15.11). In order to encourage customizing the Interlisp environment, Interlisp includes a facility for automatically loading initialization files (or "init files") when an Interlisp system is first started. Each user can have a separate "user init file" that customizes the Interlisp environment to his/her tastes. In addition, there can be a "site init file" that applies to all users at a given physical site, setting system variables that are the same for all users such as the name of the nearest printer, etc.

The process of loading init files, also known as "greeting", occurs when an Interlisp system created by **MAKESYS** (page 12.9) is started for the first time. The user can also explicitly invoke the greeting operation at any time via the function **GREET** (below). The process of greeting includes the following steps:

(1) Any previous greeting operation is undone. The side effects of the greeting operation are stored on a global variable as well as on the history list, thus enabling the previous greeting to be undone even if it has dropped off of the bottom of the history list.

(2) All of the items on the list **PREGREETFORMS** are evaluated.

(3) The site init file is loaded. **GREET** looks for a file by the name **{DSK}INIT.LISP**. If this is found, it is loaded. If it is not found, the system prints "**Please enter name of system init file (e.g. {server}<directory>INIT.extension):**" and waits for the user to type a file name, followed by a carriage return. If the user just types a carriage return without typing a file name, no site init file is loaded. Note: The site init file is loaded with *LDFLG* set to **SYSLOAD**, so that no file package information is saved, and nothing is printed out.

(4) The user init file is loaded. The user init file is found by using the variable **USERGREETFILES** (described below), which is normally set in the site init file. The user init file is loaded with normal file

package settings, but under errorset protection and with **PRETTYHEADER** set to **NIL** to suppress the "**FILE CREATED**" message.

(5) All of the items on the list **POSTGREETFORMS** are evaluated.

(6) A greeting is printed such as "**Hello,** *XXX*.", where *XXX* is the value of the variable **FIRSTNAME** (if non-**NIL**). The variable **GREETDATES** (below) can be set to modify this greeting for particular dates.

(GREET *NAME* **—)** [Function]

Performs the greeting for the user whose username is *NAME* (if *NAME* = **NIL**, uses the login name). When Interlisp first starts up, it performs **(GREET)**.

(GREETFILENAME *USER***)** [Function]

If *USER* is **T**, **GREETFILENAME** returns the file name of the site init file, asking the user if it doesn't exist. Otherwise, *USER* is interpreted to be a user's system name, and **GREETFILENAME** returns the file name for the user init file (if it exists).

USERGREETFILES [Variable]

USERGREETFILES specifies a series of file names to try as the user init file. The value of **USERGREETFILES** is a list, where each element is a list of litatoms. For each item in **USERGREETFILES**, the user name is substituted for the litatom **USER** and the value of **COMPILE.EXT** (page 18.13) is substituted for the litatom **COM**, and the litatoms are packed into a single file name. The first such file that is found is the user init file.

For example, suppose that the value of **USERGREETFILES** was

(({ERIS}< USER >LISP>INIT. COM)
({ERIS}< USER >LISP>INIT)
({ERIS}< USER >INIT. COM)
({ERIS}< USER >INIT))

If the user name was **JONES**, and the value of **COMPILE.EXT** was **DCOM**, then this would search for the files **{ERIS}<JONES>LISP>INIT.DCOM**, **{ERIS}<JONES>LISP>INIT**, **{ERIS}<JONES>INIT.DCOM**, and **{ERIS}<JONES>INIT**.

Note: The file name "specifications" in **USERGREETFILES** should be fully qualified, including all host and directory information. The directory search path (the value of **DIRECTORIES**, page 24.31) is *not* used to find the user greet files.

GREETDATES [Variable]

The value of **GREETDATES** can be used to specify special greeting messages for various dates. **GREETDATES** is a list of elements of

the form (*DATESTRING* . *STRING*), e.g. ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her **INIT.LISP** file by using a **ADDVARS** file package command like (**ADDVARS (GREETDATES ("8-FEB" . "Happy Birthday"))**). On the specified date, the **GREET** will use the indicated salutation.

Note: Users should try to make sure that their init file is "undoable". If they use the file package command "**P**" (page 17.40) to put expressions on the file to be evaluated, they should use the "undoable" version, e.g. /**SETSYNTAX** rather than **SETSYNTAX**, etc (see page 13.26). This is so another user can come up, do a **(GREET)** and have the first user's initialization undone.

It is impossible to give a complete list of all of the variables and functions that users may want to set in their init files. The default values for system variables are chosen in the hope that they will be correct for the majority of users, so many users get along with very small init files. The following describes some of the variables that users may want to reset in their init files:

Directories — The variables **DIRECTORIES** and **LISPUSERSDIRECTORIES** (page 24.31) contain lists of directories used when searching for files. **LOGINHOST/DIR** (page 24.11) determines the default directory used when calling **CONN** with no argument.

Fonts and Printing — The variables **DISPLAYFONTDIRECTORIES**, **DISPLAYFONTEXTENSIONS**, **INTERPRESSFONTDIRECTORIES**, and **PRESSFONTWIDTHSFILES** (page 27.31) must be set before fonts can be automatically loaded from files. **DEFAULTPRINTINGHOST** (page 29.4) should be set before attempting to generate hardcopy to a printer.

Network Systems — **CH.DEFAULT.ORGANIZATION** and **CH.DEFAULT.DOMAIN** (page 31.8) should be set to the default NS organization and domain, when using NS network communications. If **CH.NET.HINT** (page 31.9) is set, it can reduce the amount of time spent searching for a clearinghouse.

Interlisp-D Executive — The variable **PROMPT#FLG** (page 13.22) determines whether an "event number" is printed at the beginning of every input line. The function **CHANGESLICE** (page 13.21) can be used to change the number of events that are remembered on the history list.

Copyright Notices — **COPYRIGHTFLG**, **COPYRIGHTOWNERS**, and **DEFAULTCOPYRIGHTOWNER** (page 17.53) control the inclusion of copyright notices on source files.

Printing Functions — ****COMMENT**FLG** (page 26.43) determines how program comments are printed. **FIRSTCOL**, **PRETTYFLG**, and **CLISPIFYPRETTYFLG** (page 26.47) are among the many variables controlling how functions are pretty printed.

List Structure Editor

The variable **INITIALSLST** (page 16.76) is used when "time-stamps" are inserted in a function when it is edited. **EDITCHARACTERS** (page 16.76) is used to set the read macros used in the teletype editor.

12.2 Idle Mode

The Interlisp-D environment runs on small single-user computers, usually located in users' offices. Often, users leave their computers up and running for days, which can cause several problems. First, the phosphor in the video display screen can be perminantly marked if the same pattern is displayed for a long time (weeks). Second, if the user goes away, leaving an Interlisp-D system running, another person could possibly walk up and use the environment, taking advantage of any passwords that had been entered. To solve these problems, the Interlisp-D environment implements the concept of "idle mode."

If no keyboard or mouse action has occurred for a specified time, the Interlisp-D environment automatically enters idle mode. While idle mode is on, the display screen is blacked out, to protect the phosphor. Idle mode also runs a program to display some moving pattern on the black screen, so the screen doesn't appear broken. Usually, idle mode can be exited by pressing any key on the keyboard or mouse. However, the user can optionally specify that idle mode should erase the current password cache when it is entered, and require the next user to supply a password to exit idle mode.

Note: If either shift key is pressed while Interlisp-D is in idle mode, the current user name and the amount of time spent idling are displayed in the prompt window (which appears as long as the shift key is held down).

Idle mode can also be entered by calling the function **IDLE**, or by selecting the **Idle** menu command from the background menu (page 28.6). The **Idle** menu command has subitems that allow the user to interactively set the idle options (display program, erasing password, etc.) specified by the variable **IDLE.PROFILE**:

IDLE.PROFILE [Variable]

The value of this variable is a property list (page 3.15) which controls most aspects of idle mode. The following properties are recognized:

TIMEOUT Value is a number that determines how long (in minutes) Interlisp-D will wait before automatically entering idle mode. If **NIL**, idle mode will never be entered automatically. Default is 10 minutes.

FORGET If non-**NIL**, the user's password will be erased when idle mode is entered. Default is **NIL** (don't erase password).

Note: If the password is erased, any programs left running when idle mode is entered will fail if they try doing anything requiring passwords (such as accessing file servers).

ALLOWED.LOGINS Determines who can exit idle mode, as follows:

If the value is **NIL**, idle mode is exited without requesting login.

If the value is **LOGIN** (the default), login is required, but anyone is allowed to exit idle mode. This will overwrite the previous user's user name and password each time idle mode is exited.

If the value is one of **AUTHENTICATE**, **NS.AUTHENTICATE**, or **GV.AUTHENTICATE**, login is required and the password is checked with the net. Only allow users with accounts to exit idle mode. **NS.AUTHENTICATE** or **GV.AUTHENTICATE** specify that NS or grapevine authentication must be used, respectively. **AUTHENTICATE** indicates that either type of authentication can be tried.

If the value is a list, it should be a list of group and/or user names. The value **T** in the list means the user who was using the machine before idle mode was entered. If the value is a list, idle mode will only be exited if: (a) the new user's user name is in this list, (b) the new user is a member of a group whose name is on this list, or (c) if **T** is a member of the list, and the same user logs in with the same password.

DISPLAYFN The value of this property, which should be a function name or lambda expression, is called to display a moving pattern on the screen while in idle mode. This function is called with one argument, a window covering the whole screen. The default is **IDLE.BOUNCING.BOX** (below).

Note: Any function used as a **DISPLAYFN** should call **BLOCK** (page 23.5) frequently, so other programs can run during idle mode.

SAVEVM Value is a number that determines how long (in minutes) after idle mode is entered that **SAVEVM** (page 12.7) will be called to save the virtual memory. If **NIL**, **SAVEVM** is never called automatically from idle mode. Default is 10 minutes.

RESETVARS Value is a list of two-element lists: $((VAR_1\ EXP_1)\ (VAR_2\ EXP_2)\ ...)$. On entering idle mode, each variable VAR_N is bound to the value of the corresponding expression EXP_N. When idle mode is exited, each variable VAR_N is reset to its original value.

SUSPEND.PROCESS.NAMES Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.FUNCTIONS [Variable]

The value of this variable determines the menu raised by selecting the **Display** subitem of the **Idle** background menu command. It should be in the format used for the **ITEMS** field of a menu (page 28.39), with the selection of an item returning the appropriate display function.

(IDLE.BOUNCING.BOX *WINDOW BOX WAIT***)** [Function]

This is the default display function used for idle mode. *BOX* is bounced about *WINDOW*, with bounces taking place every *WAIT* milliseconds. *BOX* can be a string, a bitmap, a window (whose image will be bounced about), or a list containing any number of these (which will be cycled through). *BOX* defaults to the value of the variable **IDLE.BOUNCING.BOX**, which is is initially the string **"Interlisp-D"**. *WAIT* defaults to 1000 (one second).

12.3 Saving Virtual Memory State

Interlisp storage allocation occurs within a virtual memory space that is usually much larger than the physical memory on the computer. The virtual memory is stored as a large file on the computer's hard disk, called the virtual memory file. Interlisp controls the swapping of pages between this file and the real memory, swapping in virtual memory pages as they are accessed, and swapping out pages that have been modified. At any moment, the total state of the Interlisp virtual memory is stored partially in the virtual memory file, and partially in the real physical memory.

Interlisp provides facilities for saving the total state of the virtual memory, either on the virtual memory file, or in a file on an arbitrary file device. The function **LOGOUT** is used to write all altered (dirty) pages from the real memory to the virtual memory file and stop Interlisp, so that Interlisp can be restarted from the state of the **LOGOUT**. **SAVEVM** updates the virtual memory file without stopping Interlisp, which puts the virtual memory file into a consistant state (temporarily), so it could be restarted if the system crashes. **SYSOUT** and **MAKESYS** are used to save a copy of the total virtual memory state on a file, which can be loaded into another machine to restore the Interlisp state. **VMEM.PURE.STATE** can be used to "freeze" the current state of the virtual memory, so that Interlisp will come up in that state if it is restarted.

(LOGOUT *FAST***)** [Function]

Stops Interlisp, and returns control to the operating system. If Interlisp is restarted, it should come up in the same state as when the **LOGOUT** was called. **LOGOUT** will not affect the state of open files.

LOGOUT writes out all altered pages from real memory to the virtual memory file. If *FAST* is **T**, Interlisp is stopped without updating the virtual memory file. Note that after doing **LOGOUT T)** it will not be possible to restart Interlisp from the point of the **LOGOUT**, and it may not be possible to restart it at all. Typing **(LOGOUT T)** is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

If *FAST* is the litatom **?**, **LOGOUT** acts like *FLG* = **T** if the virtual memory file is consistant, otherwise it acts like *FLG* = **NIL**. This insures that the virtual memory image can be restarted as of ***some*** **previous state, not necessarily as of the LOGOUT.**

(SAVEVM —) [Function]

This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the **SAVEVM**, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the **SAVEVM)** should there be a system crash or other disaster.

If the system has been idle long enough (no keyboard or mouse activity), there are dirty pages to be written, and there are few enough dirty pages left to write that a **SAVEVM** would be quick, **SAVEVM** is automatically called. When **SAVEVM** is called automatically, the cursor is changed to a special cursor:  , stored in the variable **SAVINGCURSOR**. You can control how often **SAVEVM** is automatically called by setting the following two global variables:

SAVEVMWAIT [Variable]

SAVEVMMAX [Variable]

The system will call **SAVEVM** after being idle for **SAVEVMWAIT** seconds (initially 300) if there are fewer than **SAVEVMMAX** pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set **SAVEVMWAIT** = 0 and **SAVEVMMAX** = 10000, in which case **SAVEVM** will be called the first chance available after the first dirty page has been written.

The function **SYSOUT** saves the current state of the Interlisp virtual memory on a file, known as a "sysout file", or simply a "sysout". The file package can be used to save particular function definitions and other arbitrary objects on files, but **SYSOUT** saves the *total* state of the system. This capability can be useful in many situations: for creating customized systems for other people to use, or to save a particular system state for debugging purposes. Note that a sysout file can be very large (thousands of pages), and can take a long time to create, so it is not to be done lightly. The file produced by **SYSOUT** can be loaded into the Interlisp virtual memory and restarted to restore the virtual memory to the exact state that it had when the sysout file was made. The exact method of loading a sysout depend on the implementation. For more information on loading sysout files, see the users guide for your computer.

(SYSOUT *FILE***)** [Function]

Saves the current state of the Interlisp virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so **(PROGN (SYSOUT 'FOO) (PRINT 'HELLO))** will cause **HELLO** to be printed after the sysout file is restarted.

SYSOUT can take a very long time (ten or fifteen minutes), particularly when storing a file on a remote file server. To display some indication that something is happening, the cursor is changed to:  . Also, as the sysout file is being written, the cursor is inverted line by line, to show that activity is taking place, and how much of the sysout has completed. For example, after the **SYSOUT** is about two-thirds done, the cursor would look like:  . The **SYSOUT** cursor is stored in the variable **SYSOUTCURSOR**.

If *FILE* is non-**NIL**, the variable **SYSOUTFILE** is set to the body of *FILE*. If *FILE* is **NIL**, then the value of **SYSOUTFILE** instead. Therefore, **(SYSOUT)** will save the current state on the next higher version of a file with the same name as the previous **SYSOUT**. Also, if the extension for *FILE* is not specified, the value of **SYSOUT.EXT** is used. **SYSOUT** sets **SYSOUTDATE** (page 12.13) to **(DATE)**, the time and date that the **SYSOUT** was performed.

If **SYSOUT** was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, **SYSOUT** returns **NIL**. Otherwise it returns the full file name of *FILE*.

Actually, **SYSOUT** "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, **SYSOUT** returns a list whose **CAR** is the full file name of *FILE*. For example, **(if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO))** will

cause **HELLO** to be printed when the sysout file is restarted, but not when **SYSOUT** is initially performed.

Note: **SYSOUT** does not save the state of any open files. **WHENCLOSE** (page 24.20) can be used to associate certain operations with open files so that when a **SYSOUT** is started up, these files will be reopened, and file pointers repositioned.

SYSOUT evaluates the expressions on **BEFORESYSOUTFORMS** before creating the sysout file. This variable initially includes expressions to: (1) Set the variables **SYSOUTDATE** and **SYSOUTFILE** as described above; (2) Default the sysout file name *FILE* according to the values of the variables **SYSOUTFILE** and **SYSOUT.EXT**, as described above; and (3) Perform any necessary operations on open files as specified by calls to **WHENCLOSE** (page 24.20).

After a sysout file is restarted (but *not* when it is initially created), **SYSOUT** evaluates the expressions on **AFTERSYSOUTFORMS**. This initially includes expressions to: (1) Perform any necessary operations on previously-opened files as specified by calls to **WHENCLOSE** (page 24.20); (2) Possibly print a message, as determined by the value of **SYSOUTGAG** (see below); and (3) Call **SETINITIALS** to reset the initials used for time-stamping (page 16.76).

SYSOUTGAG [Variable]

The value of **SYSOUTGAG** determines what is printed when a sysout file is restarted. If the value of **SYSOUTGAG** is a list, the list is evaluated, and no additional message is printed. This allows the user to print a message. If **SYSOUTGAG** is non-**NIL** and not a list, no message is printed. Finally, if **SYSOUTGAG** is **NIL** (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, the user is greeted by printing the value of **HERALDSTRING** (see below) followed by a greeting message. If the sysout file was made by a different user, a message is printed, warning that the currently-loaded user init file may be incorrect for the current user (see page 12.1);

(MAKESYS *FILE NAME***)** [Function]

Used to store a new Interlisp system on the "makesys file" *FILE*. Similar to **SYSOUT**, except that before the file is made, the system is "initialized" by undoing the greet history, and clearing the display.

When the system is first started up, a "herald" is printed identifying the system, typically "**Interlisp-***XX DATE* ...". If *NAME* is non-**NIL**, **MAKESYS** will use it instead of **Interlisp-***XX* in the herald. **MAKESYS** sets **HERALDSTRING** to the herald string printed out.

MAKESYS also sets the variable **MAKESYSDATE** (page 12.13) to **(DATE)**, i.e. the time and date the system was made.

Interlisp-D contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into the virtual memory file (making it non-continuable). The frequency with which this routine runs is determined by:

BACKGROUNDPAGEFREQ [Variable]

This variable determines how often the routine that writes out dirty pages is run. The *higher* **BACKGROUNDPAGEFREQ** is set, the *greater* the time between running the dirty page writing routine. Initially it is set to 4. The lower **BACKGROUNDPAGEFREQ** is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.

(VMEM.PURE.STATE *X***)** [NoSpread Function]

VMEM.PURE.STATE modifies the swapper's page replacement algorithm so that dirty pages are only written at the end of the virtual memory backing file. This "freezes" a given virtual memory state, so that Interlisp will come up in that state whenever it is restarted. This can be used to set up a "clean" environment on a pool machine, allowing each user to initialize the system simply by rebooting the computer.

The way to use **VMEM.PURE.STATE** is to set up the environment as you wish it to be "frozen," evaluate **(VMEM.PURE.STATE T)**, and then call any function that saves the virtual memory state **(LOGOUT, SAVEVM, SYSOUT,** or **MAKESYS)**. From that point on, whenever the system is restarted, it will return to the state as of the saving operation. Future **LOGOUT, SAVEVM,** etc. operations will not reset this state.

Note: When the system is running in "pure state" mode, it uses a significant amount of the virtual memory backing file to save the "frozen" memory image, so this will reduce the amount of virtual memory space available for use.

(VMEM.PURE.STATE) returns **T** if the system is running in "pure state" mode, **NIL** otherwise.

(REALMEMORYSIZE) [Function]

Returns the number of real memory pages in the computer.

(VMEMSIZE) [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk (see **SYSOUT**, page 12.8).

\LASTVMEMFILEPAGE [Variable]

Value is the total size of the virtual memory backing file. This variable is set when the system is started. It should not be set by the user.

Note: When the virtual memory expands to the point where the virtual memory backing file is almost full, a break will occur with the warning message "Your virtual memory backing file is almost full. Save your work & reload asap." When this happens, it is strongly suggested that you save any important work and reload the system. If you continue working past this point, the system will start slowing down considerably, and it will eventually stop working.

12.4 System Version Information

Interlisp-D runs on a number of different machines, with many possible hardware configurations. There have been a number of different releases of the Interlisp-D software. These facts make it difficult to answer the important question "what software/hardware environment are you running?" when reporting bugs. The following functions allow the novice to collect this information.

(PRINT-LISP-INFORMATION *STREAM FILESTRING***)** [NoSpread Function]

Prints out a summary of the software and hardware environment that Interlisp-D is running in, and a list of all loaded patch files:

Interlisp-D version KOTO of 10-Sep-85 08:25:46
on 1108, microcode 5658, 8191 pages,
machine 222#0.125000.34652#0 on
Interlisp-D version 9-Sep-85 18:54:29
Patch files: GCPATCH dated 11-Sep-85 10:56:37

STREAM is the stream used to print the summary. If not given, it defaults to **T**.

FILESTRING is a string used to determine what loaded files should be listed as "patch files." All file names on **LOADEDFILELST** (page 17.20) that have *FILESTRING* as a substring as listed. If *FILESTRING* is not given, it defaults to the string "PATCH".

(LISP-IMPLEMENTATION-TYPE) [Function]

Returns a string identifying the type of Interlisp implementation that is running, e.g., "Interlisp-D".

(LISP-IMPLEMENTATION-VERSION) [Function]

Returns a string identifying the version of Interlisp that is running. Currently gives the system name and date, e.g., "KOTO of 10-Sep-85 08:25:46".

This uses the variables **MAKESYSNAME** and **MAKESYSDATE** (below), so it will change if the user uses **MAKESYS** (page 12.9) to create a custom sysout file, or explicitly changes these variables.

(SOFTWARE-TYPE) [Function]

Returns a string identifying the operating system that Interlisp is running under. Currently returns the string "Interlisp-D".

(SOFTWARE-VERSION) [Function]

Returns a string identifying the version of the operating system that Interlisp is running under. Currently, this returns the date that the Interlisp-D release was originally created, so it doesn't change over **MAKESYS** or **SYSOUT**.

(MACHINE-TYPE) [Function]

Returns a string identifying the type of computer hardware that Interlisp-D is running on, i.e., "1108", "1132", "1186", etc.

(MACHINE-VERSION) [Function]

Returns a string identifying the version of the computer hardware that Interlisp-D is running on. Currently returns the microcode version and real memory size.

(MACHINE-INSTANCE) [Function]

Returns a string identifying the particular machine that Interlisp-D is running on. Currently returns the machine's NS address.

(SHORT-SITE-NAME) [Function]

Returns a short string identifying the site where the machine is located. Currently returns **(ETHERHOSTNAME)** (if non-**NIL**) or the string "unknown".

(LONG-SITE-NAME) [Function]

Returns a long string identifying the site where the machine is located. Currently returns the same as **SHORT-SITE-NAME**.

SYSOUTDATE [Variable]

Value is set by **SYSOUT** (page 12.8) to the date before generating a virtual memory image file.

MAKESYSDATE [Variable]

Value is set by **MAKESYS** (page 12.9) to the date before generating a virtual memory image file.

MAKESYSNAME [Variable]

Value is a litatom identifying the release name of the current Interlisp-D system, e.g., **KOTO**.

(SYSTEMTYPE) [Function]

The **SYSTEMTYPE** function is intended to allow programmers to write system-dependent code. **SYSTEMTYPE** returns a litatom corresponding to the implementation of Interlisp: **D** (for Interlisp-D), **TOPS-20**, **TENEX**, **JERICO**, or **VAX**.

In Interlisp-D (and Interlisp-10), **(SELECTQ (SYSTEMTYPE) ...)** expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(MACHINETYPE) [Function]

Returns the type of machine that Interlisp-D is running on: either **DORADO** (for the Xerox 1132), **DOLPHIN** (for the Xerox 1100), or **DANDELION** (for the Xerox 1108).

DOVE

12.5 Date And Time Functions

(DATE *FORMAT***)** [Function]

Returns the current date and time as a string with format "*DD-MM-YY HH:MMM:SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., " **7-Jun-85 15:49:34**".

If *FORMAT* is a date format as returned by **DATEFORMAT** (below), it is used to modify the format of the date string returned by **DATE**.

(IDATE *STR***)** [Function]

STR is a date and time string. **IDATE** returns *STR* converted to a number such that if $DATE_1$ is before (earlier than) $DATE_2$, then **(IDATE** $DATE_1$**)** < **(IDATE** $DATE_2$**)**. If *STR* is **NIL**, the current date and time is used.

Note that different Interlisp implementations can have different internal date formats. However, **IDATE** always has the essential property that **(IDATE** *X***)** is less than **(IDATE** *Y***)** if *X* is before *Y*, and **(IDATE (GDATE** *N***))** equals *N*. Programs which do arithmetic other than numerical comparisons between **IDATE** numbers may not work when moved from one implementation to another.

Generally, it is possible to increment an **IDATE** number by an integral number of days by computing a "1 day" constant, the difference between two convenient **IDATE's**, e.g. **(IDIFFERENCE (IDATE " 2-JAN-80 12:00") (IDATE " 1-JAN-80 12:00"))**. This "1 day" constant can be evaluated at compile time.

IDATE is guaranteed to accept as input the dates that **DATE** will output. It will ignore the parenthesized day of the week (if any). **IDATE** also correctly handles time zone specifications for those time zones registered in the list **TIME.ZONES** (page 12.15).

(GDATE *DATE FORMAT* **—)** [Function]

Like **DATE**, except that *DATE* can be a number in internal date and time format as returned by **IDATE**. If *DATE* is **NIL**, the current time and date is used.

(DATEFORMAT KEY_1 **...** KEY_N**)** [NLambda NoSpread Function]

DATEFORMAT returns a date format suitable as a parameter to **DATE** and **GDATE**. KEY_1 ... KEY_N are a set of keywords (unevaluated). Each keyword affects the format of the date independently (except for **SLASHES** and **SPACES**). If the date returned by **(DATE)** with the default formatting was **" 7-Jun-85 15:49:34"**, the keywords would affect the formatting as follows:

NO.DATE Doesn't include the date information, e.g. **"15:49:34"**.

NUMBER.OF.MONTH Shows the month as a number instead of a name, e.g. **" 7-06-85 15:49:34"**.

YEAR.LONG Prints the year using four digits, e.g. **" 7-Jun-1985 15:49:34"**.

SLASHES Separates the day, month, and year fields with slashes, e.g. **" 7/Jun/85 15:49:34"**.

SPACES Separates the day, month, and year fields with spaces, e.g. **" 7 Jun 85 15:49:34"**.

NO.LEADING.SPACES By default, the day field will always be two characters long. If **NO.LEADING.SPACES** is specified, the day field will be one character for dates earlier than the 10th, e.g. **"7-Jun-85 15:49:34"** instead of **" 7-Jun-85 15:49:34"**.

NO.TIME Doesn't include the time information, e.g. **" 7-Jun-85"**.

TIME.ZONE Includes the time zone in the time specification, e.g. **" 7-Jun-85 15:49:34 PDT"**.

NO.SECONDS Doesn't include the seconds, e.g. **" 7-Jun-85 15:49"**.

DAY.OF.WEEK Includes the day of the week in the time specification, e.g. " **7-Jun-85 15:49:34 PDT (Friday)**".

DAY.SHORT If **DAY.OF.WEEK** is specified to include the day of the week, the week day is shortened to the first three letters, e.g. " **7-Jun-85 15:49:34 PDT (Fri)**". Note that **DAY.SHORT** has no effect unless **DAY.OF.WEEK** is also specified.

(CLOCK *N* **—)** [Function]

If *N* = 0, **CLOCK** returns the current value of the time of day clock i.e., the number of milliseconds since last system start up.

If *N* = 1, returns the value of the time of day clock when the user started up this Interlisp, i.e., difference between **(CLOCK 0)** and **(CLOCK 1)** is number of milliseconds (real time) since this Interlisp system was started.

If *N* = 2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

If *N* = 3, returns the number of milliseconds of compute time spent in garbage collections (all types).

(SETTIME *DT***)** [Function]

Sets the internal time-of-day clock. If *DT* = **NIL**, **SETTIME** attempts to get the time from the communications net; if it fails, the user is prompted for the time. If *DT* is a string in a form that **IDATE** recognizes, it is used to set the time.

The following variables are used to interpret times in different time zones. **\TimeZoneComp**, **\BeginDST**, and **\EndDST** are normally set automatically if your machine is connected to a network with a time server. For standalone machines, it may be necessary to set them by hand (or in your init file, see page 12.1) if you are not in the Pacific time zone.

TIME.ZONES [Variable]

Value is an association list that associates time zone specifications (PDT, EST, GMT, etc.) with the number of hours west of Greenwich (negative if east). If the time zone specification is a single letter, it is appended to "DT" or "ST" depending on whether daylight saving time is in effect. Initially set to:

((8 . P) (7 . M) (6 . C) (5 . E) (0 . GMT))

This list is used by **DATE** and **GDATE** when generating a date with the **TIME.ZONE** format is specified, and by **IDATE** when parsing dates.

\TimeZoneComp [Variable]

This variable should be initialized to the number of hours west of Greenwich (negative if east). For the U.S. west coast it is 8. For the east coast it is 5.

\BeginDST [Variable]

\EndDST [Variable]

\BeginDST is the day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day); **\EndDST** is the day on or before which Daylight Savings Time ends. Days are numbered with 1 being January 1, and counting the days as for a leap year. In the USA where Daylight Savings Time is observed, **\BeginDST** = 121 and **\EndDST** = 305. In a region where Daylight Savings Time is not observed at all, set **\BeginDST** to 367.

12.6 Timers and Duration Functions

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarm clock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Interlisp-D environment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

Three functions are provided: **SETUPTIMER**, **SETUPTIMER.DATE**, and **TIMEREXPIRED?**. Also several new i.s.oprs have been defined: **forDuration**, **during**, **untilDate**, **timerUnits**, **usingTimer**, and **resourceName** (reasonable variations on upper/lower case are permissible).

These functions use an object called a timer, which encodes a future clock time at which a signal is desired. A timer is constructed by the functions **SETUPTIMER** and **SETUPTIMER.DATE**, and is created with a basic clock "unit" selected from among **SECONDS**, **MILLISECONDS**, or **TICKS**. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is **MILLISECONDS**.

Currently, the **TICKS** unit is a function of the particular machine that Interlisp-D is running on. The Xerox 1132 has about 1680 ticks per millisecond; the Xerox 1108 has about 34.746 ticks per millisecond; the Xerox 1185 and 1186 have about 62.5 ticks per

millisecond. The advantage of using **TICKS** rather than one of the uniform interfaces is primarily speed; e.g., it may take over 400 microseconds to read the milliseconds clock (a software facility that uses the real clock), whereas reading the real clock itself may take less than ten microseconds. The disadvantage of the **TICKS** unit is its short roll-over interval (about 20 minutes) compared to the **MILLISECONDS** roll-over interval (about two weeks), and also the dependency on particular machine parameters.

(SETUPTIMER *INTERVAL OldTimer? timerUnits intervalUnits***)** [Function]

SETUPTIMER returns a timer that will "go off" (as tested by **TIMEREXPIRED?**) after a specified time-interval measured from the current clock time. **SETUPTIMER** has one required and three optional arguments:

INTERVAL must be a integer specifying how long an interval is desired. *timerUnits* specifies the units of measure for the interval (defaults to **MILLISECONDS**).

If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer. *intervalUnits* specifies the units in which the *OldTimer?* is expressed (defaults to the value of *timerUnits*.

(SETUPTIMER.DATE *DTS OldTimer?***)** [Function]

SETUPTIMER.DATE returns a timer (using the **SECONDS** time unit) that will "go off" at a specified date and time. *DTS* is a Date/Time string such as **IDATE** accepts (page 12.14). If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer.

SETUPTIMER.DATE operates by first subtracting **(IDATE)** from **(IDATE** *DTS***)**, so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(TIMEREXPIRED? *TIMER ClockValue.or.timerUnits***)** [Function]

If *TIMER* is a timer, and *ClockValue.or.timerUnits* is the time-unit of *TIMER*, **TIMEREXPIRED?** returns true if *TIMER* has "gone off".

ClockValue.or.timerUnits can also be a timer, in which case **TIMEREXPIRED?** compares the two timers (which must be in the same timer units). If **X** and **Y** are timers, then **(TIMEREXPIRED? X Y)** is true if **X** is set for an *earlier* time than **Y**.

There are a number of i.s.oprs that make it easier to use timers in iterative statements (page 9.9). These i.s.oprs are given below in the "canonical" form, with the second "word" capitalized, but the all-caps and all-lower-case versions are also acceptable.

forDuration *INTERVAL* [I.S. Operator]

during *INTERVAL* [I.S. Operator]

INTERVAL is an integer specifying an interval of time during which the iterative statement will loop.

timerUnits *UNITS* [I.S. Operator]

UNITS specifies the time units of the *INTERVAL* specified in **forDuration**.

untilDate *DTS* [I.S. Operator]

DTS is a Date/Time string (such as **IDATE** accepts) specifying when the iterative statement should stop looping.

usingTimer *TIMER* [I.S. Operator]

If **usingTimer** is given, *TIMER* is reused as the timer for **forDuration** or **untilDate**, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

resourceName *RESOURCE* [I.S. Operator]

RESOURCE specifies a resource name to be used as the timer storage (see page 17.24). If *RESOURCE* = **T**, it will be converted to an internal name.

Some examples:

```
(during 6MONTHS timerUnits 'SECONDS
 until (TENANT-VACATED? HouseHolder)
 do (DISMISS <for-about-a-day>)
   (HARRASS HouseHolder)
 finally (if (NOT (TENANT-VACATED? HouseHolder))
      then (EVICT-TENANT HouseHolder)))
```

This example shows that how is is possible to have two termination condition: (1) when the time interval of **6MONTHS** has elapsed, or (2) when the predicate **(TENANT-VACATED? HouseHolder)** becomes true. Note that the "finally" clause is executed regardless of which termination condition caused it.

Also note that since the millisecond clock will "roll over" about every two weeks, "**6MONTHS**" wouldn't be an appropriate interval if the timer units were the default case, namely **MILLISECONDS**.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
   do (CARRY.ON.AS.USUAL)
   finally (PROMPTPRINT "Have you had your 10-day
check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
(untilDate "31-DEC-83 23:59:59" usingTimer \RandomTimer
 when (WINNING?) do (RETURN)
 finally (ERROR "You've been losing this whole year!"))
```

Here we see a usage of an explicit date for the time interval; also, the user has squirreled away some storage (as the value of **\RandomTimer**) for use by the call to **SETUPTIMER** in this loop.

```
(forDuration SOMEINTERVAL
 resourceName \INNERLOOPBOX
 timerunits 'TICKS
 do (CRITICAL.INNER.LOOP))
```

For this loop, the user doesn't want any **CONS**ing to take place, so **\INNERLOOPBOX** will be defined as a resource which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a **WITH-RESOURCES** call. Furthermore, he has specified a time unit of **TICKS**, for lower overhead in this critical inner loop. In fact specifying a **resourceName** of **T** would have been the same as specifying it to be **\ForDurationOfBox**; this is just a simpler way to specify that a resource is wanted, without having to think up a name.

12.7 Resources

Interlisp is based on the use of a storage-management system which allocates memory space for new data objects, and automatically reclaims the space when no longer in use. More generally, Interlisp manages shared "resources", such as files, semaphors for processes, etc. The protocols for allocating and freeing such resources resemble those of ordinary storage management.

Sometimes users need to explicitly manage the allocation of resources. They may desire the efficiency of explicit reclamation of certain temporary data; or it may be expensive to initialize a complex data object; or there may be an application that must not allocate new cells during some critical section of code.

The file package type **RESOURCES** is available to help with the definition and usage of such classes of data; the definition of a **RESOURCE** specifies prototype code to do the basic management operations. The filepkg command **RESOURCES** (page 17.39) is

used to save such definitions on files, and **INITRESOURCES** (page 17.39) causes the initialization code to be output.

The basic needs of resource management are (1) obtaining a data item from the Lisp memory management system and configuring it to be a totally new instance of the resource in question, (2) freeing up an instance which is no longer needed, (3) getting an instance of the resource for temporary usage [whether "fresh" or a formerly freed-up instance], and (4) setting up any prerequisite global data structures and variables. A resources definition consists of four "methods": **INIT**, **NEW**, **GET**, and **FREE**; each "method" is a form that will specialize the definition for four corresponding user-level macros **INITRESOURCE**, **NEWRESOURCE**, **GETRESOURCE**, and **FREERESOURCE**. **PUTDEF** is used to make a resources definition, and the four components are specified in a proplist:

(PUTDEF
'*RESOURCENAME*
'RESOURCES
'(NEW *NEW-INSTANCE-GENERATION-CODE*
FREE *FREEING-UP-CODE*
GET *GET-INSTANCE-CODE*
INIT *INITIALIZATION-CODE*))

Each of the *xxx-CODE* forms is a form that will appear as if it were the body of a substitution macro definition for the corresponding macro [see the discussion on the macros below].

12.7.1 A Simple Example

Suppose one has several pieces of code which use a 256-character string as a scratch string. One could simply generate a new string each time, but that would be inefficient if done repeatedly. If the user can guarantee that there are no re-entrant uses of the scratch string, then it could simply be stored in a global variable. However, if the code might be re-entrant on occasion, the program has to take precautions that two programs do not use the same scratch string at the same time. [Note: this consideration becomes very important in a multi-process environment. It is hard to guarantee that two processes won't be running the same code at the same time, without using elaborate locks.] A typical tactic would be to store the scratch string in a global variable, and set the variable to **NIL** whenever the string is in use (so that re-entrant usages would know to get a "new" instance). For example, assuming the global variable **TEMPSTRINGBUFFER** is initialized to **NIL**:

```
[DEFINEQ (WITHSTRING NIL
  (PROG ((BUF (OR (PROG1 TEMPSTRINGBUFFER
                         (SETQ TEMPSTRINGBUFFER NIL))
```

```
        (ALLOCSTRING 256))))
  ... use the scratch string in the variable BUF ...
  (SETQ TEMPSTRINGBUFFER BUF)
  (RETURN]
```

Here, the basic elements of a "resource" usage may be seen: (1) a call **(ALLOCSTRING 256)** allocates fresh instances of "buffer", (2) after usage is completed the instance is returned to the "free" state, by putting it back in the global variable **TEMPSTRINGBUFFER** where a subsequent call will find it, (3) the prog-binding of **BUF** will get an existing instance of a string buffer if there is one -- otherwise it will get a new instance which will later be available for reuse, and (4) some initialization is performed before usage of the resource (in this case, it is the setting of the global variable **TEMPSTRINGBUFFER**).

Given the following resources definition:

```
(PUTDEF
  'STRINGBUFFER
  'RESOURCES
  '(NEW  (ALLOCSTRING 256)
    FREE (SETQ TEMPSTRINGBUFFER (PROG1 . ARGS))
    GET  (OR (PROG1 TEMPSTRINGBUFFER
                (SETQ TEMPSTRINGBUFFER NIL))
             (NEWRESOURCE TEMPSTRINGBUFFER)))
    INIT (SETQ TEMPSTRINGBUFFER NIL)))
```

we could then redo the example above as

```
(DEFINEQ (WITHSTRING NIL
  (PROG ((BUF (GETRESOURCE STRINGBUFFER)))
    ... use the string in the variable BUF ...
    (FREERESOURCE STRINGBUFFER BUF)
    (RETURN]
```

The advantage of doing the coding this way is that the resource management part of **WITHSTRING** is fully contained in the expansions of **GETRESOURCE** and **FREERESOURCE**, and thus there is no confusion between what is **WITHSTRING** code and what is resource management code. This particuar advantage will be multiplied if there are other functions which need a "temporary" string buffer; and of course, the resultant modularity makes it much easier to contemplate minor variations on, as well as multiple clients of, the **STRINGBUFFER** resource.

In fact, the scenario just shown above in the **WITHSTRING** example is so commonly useful that an abbreviation has been added; if a resources definition is made with *only* a **NEW** method, then appropriate **FREE**, **GET**, and **INIT** methods will be inferred, along with a coordinated globalvar, to be parallel to the above definition. So the above definition could be more simply written

```
(PUTDEF 'STRINGBUFFER
    'RESOURCES
    '(NEW (ALLOCSTRING 256)))
```

and every thing would work the same.

The macro **WITH-RESOURCES** simplifies the common scoping case, where at the beginning of some piece of code, there are one or more **GETRESOURCE** calls the results of which are each bound to a lambda variable; and at the ending of that code a **FREERESOURCE** call is done on each instance. Since the resources are locally bound to variables with the same name as the resource itself, the definition for **WITHSTRING** then simplifies to

```
(DEFINEQ (WITHSTRING NIL
  (WITH-RESOURCES (STRINGBUFFER)
  ... use the string in the variable STRINGBUFFER ...]
```

12.7.2 Trade-offs in More Complicated Cases

This simple example presumes that the various functions which use the resource are generally not re-entrant. While an occasional re-entrant use will be handled correctly (another example of the resource will simply be created), if this were to happen too often, then many of the resource requests will create and throw away new objects, which defeats one of the major purposes of using resources. A slightly more complex **GET** and **FREE** method can be of much benefit in maintaining a pool of available resources; if the resource were defined to maintain a list of "free" instances, then the **GET** method could simply take one off the list and the **FREE** method could just push it back onto the list. In this simple example, the **SETQ** in the **FREE** method defined above would just become a "**push**", and the first clause of the **GET** method would just be **(pop TEMPSTRINGBUFFER)**

A word of caution: if the datatype of the resource is something very small that Interlisp system is "good" at allocating and reclaiming, then explicit user storage management will probably not do much better than the combination of cons/createcell and the garbage collector. This would especially be so if more complicated **GET** and **FREE** methods were to be used, since their overhead would be closer to that of the built-in system facilities. Finally, it must be considered whether retaining multiple instances of the resource is a net gain; if the re-entrant case is truly rare, it may be more worthwhile to retain at most one instance, and simply let the instances created by the rarely-used case be reclaimed in the normal course of garbage collection.

12.7.3 Macros for Accessing Resources

Four user-level macros are defined for accessing resources:

(NEWRESOURCE *RESOURCENAME . ARGS***)** [Macro]

(FREERESOURCE *RESOURCENAME . ARGS***)** [Macro]

(GETRESOURCE *RESOURCENAME . ARGS***)** [Macro]

(INITRESOURCE *RESOURCENAME . ARGS***)** [Macro]

Each of these macros behave as if they were defined as a substitution macro of the form

(*(RESOURCENAME . ARGS) MACROBODY*)

where the expression *MACROBODY* is selected by using the "code" supplied by the corresponding method from the *RESOURCENAME* definition.

Note that it is possible to pass "arguments" to the user's resource allocation macros. For example, if the **GET** method for the resource **FOO** is **(GETFOO . ARGS)**, then **(GETRESOURCE FOO X Y)** is transformed into **(GETFOO X Y)**. This form was used in the **FREE** method of the **STRINGBUFFER** resource described above, to pass the old **STRINGBUFFER** object to be freed.

(WITH-RESOURCES ($RESOURCE_1$ $RESOURCE_2$ ...**)** $FORM_1$ $FORM_2$ ...**)** [Macro]

The **WITH-RESOURCES** macro binds lambda variables of the same name as the resources (for each of the resources $RESOURCE_1$, $RESOURCE_2$, etc.) to the result of the **GETRESOURCE** macro; then executes the forms $FORM_1$, $FORM_2$, etc., does a **FREERESOURCE** on each instance, and returns the value of the last form (evaluated and saved before the **FREERESOURCE**s).

Note: **(WITH-RESOURCES** *RESOURCE* ...**)** is interpreted the same as **(WITH-RESOURCES (***RESOURCE***)** ...**)**. Also, the singular name **WITH-RESOURCE** is accepted as a synonym for **WITH-RESOURCES**.

12.7.4 Saving Resources in a File

Resources definitions may be saved on files using the **RESOURCES** file package command (page 17.39). Typically, one only needs the full definition available when compiling or interpreting the code, so it is appropriate to put the file package command in a **(DECLARE: EVAL@COMPILE DONTCOPY** ...**)** declaration, just as one might do for a **RECORDS** declaration. But

just as certain record declarations need *some* initialization in the run-time environment, so do most resources. This initialization is specified by the resource's **INIT** method, which is executed automatically when the resource is defined by the **PUTDEF** output by the **RESOURCES** command. However, if the **RESOURCES** command is in a **DONTCOPY** expression and thus is not included in the compiled file, then it is necessary to include a separate **INITRESOURCES** command (page 17.39) in the filecoms to insure that the resource is properly initialized.

12.8 Pattern Matching

Interlisp provides a fairly general pattern match facility that allows the user to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match. Essentially, the user writes "Does the (expression) X look like (the pattern) P?" For example, **(match X with (& 'A -- 'B))** asks whether the second element of **X** is an **A**, and the last element a **B**. The implementation of the matching is performed by computing (once) the equivalent Interlisp expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP or PLANNER. For example, the translation of **(match X with (& 'A -- 'B))** is:

```
(AND (EQ (CADR X) 'A)
     (EQ (CAR (LAST (CDDR X))) 'B))
```

Thus the pattern match facility is really a pattern match compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. As a result, the pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in Interlisp expressions comparable to those he would generate by hand. Wherever possible, already existing Interlisp functions are used in the translation, e.g., the translation of **($ 'A $)** uses **MEMB**, **($ ('A $) $)** uses **ASSOC**, etc.

The syntax for pattern match expressions is **(match** *FORM* **with** *PATTERN***)**, where *PATTERN* is a list as described below. If *FORM* appears more than once in the translation, and it is not either a

variable, or an expression that is easy to (re)compute, such as **(CAR Y)**, **(CDDR Z)**, etc., a dummy variable will be generated and bound to the value of *FORM* so that *FORM* is not evaluated a multiple number of times. For example, the translation of **(match (FOO X) with ($ 'A $))** is simply **(MEMB 'A (FOO X))**, while the translation of **(match (FOO X) with ('A 'B --))** is:

```
[PROG ($$2)
  (RETURN
    (AND (EQ (CAR (SETQ $$2 (FOO X)))
             'A)
         (EQ (CADR $$2) 'B]
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in **NIL**, i.e., there are no **LISTP** checks inserted in the translation to check tails. For example, the translation of **(match X with ('A & --))** is **(AND (EQ (CAR X) (QUOTE A)) (CDR X))**, which will match with **(A B)** as well as **(A . B)**. Similarly, the pattern match compiler does not insert **LISTP** checks on elements, e.g., **(match X with (('A --) --))** translates simply as **(EQ (CAAR X) 'A)**, and **(match X with (($1 $1 --) --))** as **(CDAR X)**. Note that the user can explicitly insert **LISTP** checks himself by using **@**, as described below, e.g., **(match X with (($1 $1 --)@LISTP --))** translates as **(CDR (LISTP (CAR X)))**.

Note: The insertion of **LISTP** checks for *elements* is controlled by the variable **PATLISTPCHECK**. When **PATLISTPCHECK** is **T**, **LISTP** checks are inserted, e.g., **(match X with (('A --) --))** translates as: **(EQ (CAR (LISTP (CAR (LISTP X)))) 'A)**. **PATLISTPCHECK** is initially **NIL**. Its value can be changed within a particular function by using a local CLISP declaration (see page 21.13).

Note: Pattern match expressions are translated using the DWIM and CLISP facilities, using all CLISP declarations in effect (standard/fast/undoable) (see page 21.12).

12.8.1 Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. For example, in the editor's pattern matcher, "**--**" (page 16.19) matches any arbitrary segment of a list, while **&** or a subpattern match only one element of a list. Those patterns which may match a segment of a list are called *segment* patterns; those that match a single element are called *element* patterns.

12.8.2 Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or **&** — Matches an arbitrary element of a list.

'EXPRESSION — Matches only an element which is equal to the given expression e.g., **'A**, **'(A B)**.

EQ, **MEMB**, and **ASSOC** are automatically used in the translation when the quoted expression is atomic, otherwise **EQUAL**, **MEMBER**, and **SASSOC**.

= *FORM* — Matches only an element which is **EQUAL** to the value of *FORM*, e.g., **= X**, **= (REVERSE Y)**.

= = *FORM* — Same as **=**, but uses an **EQ** check instead of **EQUAL**.

ATOM — The treatment depends on setting of **PATVARDEFAULT**. If **PATVARDEFAULT** is **'** or **QUOTE**, same as *'ATOM*. If **PATVARDEFAULT** is **=** or **EQUAL**, same as **=** *ATOM*. If **PATVARDEFAULT** is **= =** or **EQ**, same as **= =** *ATOM*. If **PATVARDEFAULT** is **←** or **SETQ**, same as *ATOM*←**&**. **PATVARDEFAULT** is initially **'**.

PATVARDEFAULT can be changed within a particular function by using a local CLISP declaration (see page 21.13).

Note: numbers and strings are always interpreted as though **PATVARDEFAULT** were **=**, regardless of its setting. **EQ**, **MEMB**, and **ASSOC** are used for comparisons involving small integers.

($PATTERN_1$ ... $PATTERN_N$**)** — Matches a list which matches the given patterns, e.g., **(& &)**, **(-- 'A)**.

ELEMENT-PATTERN@FN — Matches an element if *ELEMENT-PATTERN* matches it, and *FN* (name of a function or a **LAMBDA** expression) applied to that element returns non-**NIL**. For example, **&@NUMBERP** matches a number and **('A --)@FOO** matches a list whose first element is A, and for which **FOO** applied to that list is non-**NIL**.

For "simple" tests, the function-object is applied before a match is attempted with the pattern, e.g., **((-- 'A --)@LISTP --)** translates as **(AND (LISTP (CAR X)) (MEMB 'A (CAR X)))**, not the other way around. *FN* may also be a *FORM* in terms of the variable **@**, e.g., **&@(EQ @ 3)** is equivalent to **= 3**.

***** — Matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-**NIL** if the match succeeds and **NIL** if it fails. However, if a * appears in the pattern, the expression generated could also return **NIL** if the match succeeds and * was matched to **NIL**. For example, **(match X with ('A * --))** translates as **(AND (EQ (CAR X) 'A) (CADR X))**, so if **X** is equal to **(A NIL B)** then **(match X with ('A * --))** returns **NIL** even though the match succeeded.

~ELEMENT-PATTERN Matches an element if the element is *not* matched by *ELEMENT-PATTERN*, e.g., **~'A**, **~ = X**, **~(-- 'A --)**.

(*ANY* *ELEMENT-PATTERN ELEMENT-PATTERN* ...) Matches if any of the contained patterns match.

12.8.3 Segment Patterns

$ or -- Matches any segment of a list (including one of zero length).

The difference between **$** and **--** is in the type of search they generate. For example, **(match X with ($ 'A 'B $))** translates as **(EQ (CADR (MEMB 'A X)) 'B)**, whereas **(match X with (-- 'A 'B $))** translates as:

```
[SOME X (FUNCTION (LAMBDA ($$2 $$1)
 (AND (EQ $$2 'A)
   (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of **($ 'A 'B $)** would be "Is the element following the *first* **A** a **B**?", whereas a paraphrase of **(-- 'A 'B $)** would be "Is there *any* **A** immediately followed by a **B**?" Note that the pattern employing **$** will result in a more efficient search than that employing **--**. However, **($ 'A 'B $)** will not match with **(X Y Z A M O A B C)**, but **(-- 'A 'B $)** will.

Essentially, once a pattern following a **$** matches, the **$** never resumes searching, whereas **--** produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both **--** and **$** will be the same. For example, both **(match X with ($ 'A $3 $))** and **(match X with (-- 'A $3 --))** translate as **(CDDDR (MEMB (QUOTE A) X))**, because if there are not three elements following the first **A**, there certainly will not be three elements following subsequent **A**'s, so there is no reason to continue searching, even for **--**. Similarly, **($ 'A $ 'B $)** and **(-- 'A -- 'B --)** are equivalent.

$2, $3, etc. Matches a segment of the given length. Note that **$1** is not a segment pattern.

!*ELEMENT-PATTERN* Matches any segment which *ELEMENT-PATTERN* would match as a list. For example, if the value of **FOO** is **(A B C)**, **! = FOO** will match the segment **... A B C ...** etc.

Note: Since **!** appearing in front of the last pattern specifies a match with some *tail* of the given expression, it also makes sense in this case for a **!** to appear in front of a pattern that can only match with an atom, e.g., **($2 !'A)** means match if **CDDR** of the expression is the atom **A**. Similarly, **(match X with ($! 'A))** translates to **(EQ (CDR (LAST X)) 'A)**.

!*ATOM* treatment depends on setting of **PATVARDEFAULT**. If **PATVARDEFAULT** is **'** or **QUOTE**, same as **!'***ATOM* (see above

discussion). If **PATVARDEFAULT** is **=** or **EQUAL**, same as **! =** *ATOM*. If **PATVARDEFAULT** is **= =** or **EQ**, same as **! = =** *ATOM*. If **PATVARDEFAULT** is **←** or **SETQ**, same as *ATOM*←**$**.

The atom "." is treated *exactly* like "!". In addition, if a pattern ends in an atom, the "." is first changed to "!", e.g., **($1 . A)** and **($1 ! A)** are equivalent, even though the atom "." does not explicitly appear in the pattern.

One exception where "." is not treated like "!": "." preceding an assignment does not have the special interpretation that "!" has preceding an assignment (see below). For example, **(match X with ('A . FOO←'B))** translates as:

```
(AND (EQ (CAR X) 'A)
     (EQ (CDR X) 'B)
     (SETQ FOO (CDR X)))
```

but **(match X with ('A ! FOO←'B))** translates as:

```
(AND (EQ (CAR X) 'A)
     (NULL (CDDR X))
     (EQ (CADR X) 'B)
     (SETQ FOO (CDR X)))
```

SEGMENT-PATTERN@FUNCTION-OBJECT Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-**NIL**. For example, **($@CDDR 'D $)** matches **(A B C D E)** but not **(A B D E)**, since **CDDR** of **(A B)** is **NIL**.

Note: an **@** pattern applied to a segment will require *computing* the corresponding structure (with **LDIFF**) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

12.8.4 Assignments

Any pattern element may be preceded by "*VARIABLE*←", meaning that if the match succeeds (i.e., everything matches), *VARIABLE* is to be set to the thing that matches that pattern element. For example, if **X** is **(A B C D E)**, **(match X with ($2 Y←$3))** will set **Y** to **(C D E)**. Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match, e.g., **(match X with (Y←$1 = Y --))** will *not* match with **(A A B C ...)**, unless, of course, the value of **Y** was **A** before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a **!**, the assignment is to the *tail* of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern. For example, if **X** is **(A B C D E)**, **(match X with ($!Y←'C 'D $))** sets **Y** to **(C D E)**, i.e., **CDDR**

of **X**. In other words, when ! precedes an assignment, it acts as a modifier to the ←, and has no effect whatsoever on the pattern itself, e.g., **(match X with ('A 'B))** and **(match X with ('A !FOO←'B))** match identically, and in the latter case, **FOO** will be set to **CDR** of **X**.

Note: ***←***PATTERN-ELEMENT* and **!*←***PATTERN-ELEMENT* are acceptable, e.g., **(match X with ($ 'A *←('B --) --))** translates as:

```
[PROG ($$2) (RETURN
 (AND (EQ (CAADR (SETQ $$2 (MEMB 'A X))) 'B)
   (CADR $$2]
```

12.8.5 Place-Markers

Variables of the form *#N*, *N* a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern, e.g., **(match X with (#1←$1 =(ADD1 #1)))** will match the list **(2 3)**. However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of **PATVARDEFAULT**, the first appearance of a defaulted place-marker is interpreted as though **PATVARDEFAULT** were ←. Thus the above pattern could have been written as **(match X with (1 =(ADD1 1)))**. Subsequent appearances of a place-marker are interpreted as though **PATVARDEFAULT** were **=**. For example, **(match X with (#1 #1 --))** is equivalent to **(match X with (#1←$1 = #1 --))**, and translates as **(AND (CDR X) (EQUAL (CAR X) (CADR X))**. (Note that **(EQUAL (CAR X) (CADR X))** would incorrectly match with **(NIL)**.)

12.8.6 Replacements

The construct *PATTERN-ELEMENT*←*FORM* specifies that if the match succeeds, the part of the data that matched is to be *replaced* with the value of *FORM*. For example, if **X =(A B C D E)**, **(match X with ($ 'C $1←Y $1))** will replace the third element of **X** with the value of **Y**. As with assignments, replacements are not performed until after it is determined that the entire match will be successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if **X** is **(A B C D E F)** and **FOO** is **(1 2 3)**, after the pattern **('A $←FOO 'D $)** is matched with

X, **X** will be **(A 1 2 3 D E F)**, and **FOO** will be **EQ** to **CDR** of **X**, i.e., **(1 2 3 D E F)**.

Note that **($ FOO←FIE $)** is ambiguous, since it is not clear whether **FOO** or **FIE** is the pattern element, i.e., whether ← specifies assignment or replacement. For example, if **PATVARDEFAULT** is **=**, this pattern can be interpreted as **($ FOO← = FIE $)**, meaning search for the value of **FIE**, and if found set **FOO** to it, or **($ = FOO←FIE $)** meaning search for the value of **FOO**, and if found, store the value of **FIE** into the corresponding position. In such cases, the user should disambiguate by not using the **PATVARDEFAULT** option, i.e., by specifying **'** or **=**.

Note: Replacements are normally done with **RPLACA** or **RPLACD**. The user can specify that /**RPLACA** and /**RPLACD** should be used, or **FRPLACA** and **FRPLACD**, by means of CLISP declarations (see page 21.12).

12.8.7 Reconstruction

The user can specify a value for a pattern match operation other than what is returned by the match by writing **(match** *FORM*$_1$ **with** *PATTERN* **= >** *FORM*$_2$**)**. For example, **(match X with (FOO←$ 'A --) = > (REVERSE FOO))** translates as:

```
[PROG ($$2)
  (RETURN
    (COND ((SETQ $$2 (MEMB 'A X))
           (SETQ FOO (LDIFF X $2))
           (REVERSE FOO]
```

Place-markers in the pattern can be referred to from within *FORM*, e.g., the above could also have been written as **(match X with (!#1 'A --) = > (REVERSE #1))**. If **->** is used in place of **= >**, the expression being matched is also *physically changed* to the value of *FORM*. For example, **(match X with (#1 'A !#2) -> (CONS #1 #2))** would remove the second element from **X**, if it were equal to **A**.

In general, **(match** *FORM*$_1$ **with** *PATTERN* **->** *FORM*$_2$**)** is translated so as to compute *FORM*$_2$ if the match is successful, and then smash its value into the first node of *FORM*$_1$. However, whenever possible, the translation does not actually require *FORM*$_2$ to be computed in its entirety, but instead the pattern match compiler uses *FORM*$_2$ as an indication of what should be done to *FORM*$_1$. For example, **(match X with (#1 'A !#2) -> (CONS #1 #2))** translates as **(AND (EQ (CADR X) 'A) (RPLACD X (CDDR X)))**.

12.8.8 Examples

Example: **(match X with (-- 'A --))**

-- matches any arbitrary segment. 'A matches only an A, and the second -- again matches an arbitrary segment; thus this translates to **(MEMB 'A X)**.

Example: **(match X with (-- 'A))**

Again, -- matches an arbitrary segment; however, since there is no -- after the 'A, A must be the last element of X. Thus this translates to: **(EQ (CAR (LAST X)) 'A)**.

Example: **(match X with ('A 'B -- 'C $3 --))**

CAR of X must be A, and CADR must be B, and there must be at least three elements after the first C, so the translation is:

```
(AND (EQ (CAR X) 'A)
  (EQ (CADR X) 'B)
  (CDDDR (MEMB 'C (CDDR X))))
```

Example: **(match X with (('A 'B) 'C Y←$1 $))**

Since ('A 'B) does not end in $ or --, (CDDAR X) must be NIL. The translation is:

```
(COND
 ((AND (EQ (CAAR X) 'A)
    (EQ (CADAR X) 'B)
    (NULL (CDDAR X))
    (EQ (CADR X) 'C)
    (CDDR X))
  (SETQ Y (CADDR X))
  T))
```

Example: **(match X with (#1 'A $ 'B 'C #1 $))**

#1 is implicitly assigned to the first element in the list. The $ searches for the first B following A. This B must be followed by a C, and the C by an expression equal to the first element. The translation is:

```
[PROG ($$2)
 (RETURN
  (AND (EQ (CADR X) 'A)
    (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
    (CDDR $$2)
    (EQUAL (CADDR $$2) (CAR X]
```

Example: **(match X with (#1 'A -- 'B 'C #1 $))**

Similar to the pattern above, except that -- specifies a search for *any* B followed by a C followed by the first element, so the translation is:

```
[AND (EQ (CADR X) 'A)
  (SOME (CDDR X)
```

```
(FUNCTION (LAMBDA ($$2 $$1)
  (AND (EQ $$2 'B)
     (EQ (CADR $$1) 'C)
     (CDDR $$1)
     (EQUAL (CADDR $$1) (CAR X]
```