

LispCourse #1: Getting InterLisp-D Up and Running

Topic 0: Bravery

Topic 1: Getting InterLisp-D Up and Running

A. InterLisp-D is a comprehensive environment

B. Basic differences between Dolphins, Dorados, & Dandelions

Alto Exec versus Installation Utilities

File Systems ž Partitions (2 vs 5 vs 19) versus Logical Volumes

C. Background Concepts

Memory ž core versus disk versus file-server

Virtual Memory ž pageing(swapping)

Lisp system contained in a virtual image

Vmems and Sysouts

D. Basic Procedure & Concepts

Set up local disk

Vmem file

Frequency: Once

Retrieve necessary Lisp files

Support files: microcode, run files, etc.

Lisp.Sysout (full.sysout; *.sysout)

Frequency: New releases

Start up Lisp

Clean Lisp (create new virtual image from sysout)

Restarting Lisp (use old virtual image)

Frequency: As necessary

Leaving Lisp

"(LOGOUT)"

"(LOGOUT T)"

Frequency: As necessary

E. Procedures ž Prefatory Remarks

Why are the getting started procedures such a mess?

Square pegs in round holes, insufficient resources, etc.

Assumption: You know how to boot the machines.

F. Procedures ž Dolphins and Dorados (Alto-based)

Local disk configuration ž Files on a partition

Set up local disk ž NewUserDisk.cm

Needs to be a blank partition ž Use NewOs in NetExec to erase

Retrieve NewUserDisk.cm using FTP

(currently on {eris}<lisp>harmony>cm>newuserdisk.cis)

"@NewUserDisk"

Retrieve necessary Lisp files

Not necessary first time after NewUserDisk.cm

Thereafter for new releases use: UpdateLisp.cm

Retrieve UpdateLisp.cm using FTP

(currently on {eris}<lisp>harmony>cm>updatelisp.cis)

"@UpdateLisp"

Start up Lisp

Clean Lisp

"Lisp <sysout file name>"

Examples:

"Lisp {eris}<lisp>harmony>basics>full.sysout"

"Lisp {dsk2}lisp.sysout"

Restarting Lisp

"Lisp"

Error msg: "Inconsistent Vmem"

Leaving Lisp

"(LOGOUT)" or "(LOGOUT T)" leaves you in Alto Exec

"Quit" and power-off machine

G. Procedures ž Dandelions

Note: There are several different utilities for setting up and running Lisp on a Dandelion!!!! The one described here is the most commonly used at PARC.

Local disk configuration ž Logical volumes, Vmem = Logical Volume

Utilities: Othello and Hello

Set up local disk & Retrieve necessary Lisp files

3-Boot the DLion to load Othello from the net

"login"

Partition the disk into logical volumes & retrieve files

"@[eris]<lisp>harmony>cm>Partition43Lisp.othello"

<Note: Alternative partitionings are available on [eris]<lisp>harmony>cm> for Star users or 10/29 Mbyte disks.>

Start up Lisp

Clean Lisp

1-Boot the DLion to load Hello from the local disk

"login"

"@[eris]<lisp>harmony>cm>InstallLisp.hello"

(or InstallFull.hello)

When Lisp comes up, you will be asked for the name of the init file. At PARC: "{eris}<lisp>harmony>basics>init.cis"
[Details on init files will come in later class.]

If this is the first startup after the disk has been partitioned, the LispFiles volume must be initialized by typing:
"DFSCREATEDIRECTORY[LispFiles]"

Restarting Lisp

0-Boot the DLion (goes directly into Lisp)

Flashing 217 in MP indicates "Inconsistent VMem"

Leaving Lisp

"(LOGOUT)" or "(LOGOUT T)" returns you to Hello

Turn off machine or get new lisp.

Alternative: Keeping clean sysouts on spare logical volumes

To install the sysout on the logical volume:

1-Boot the DLion to load Hello from the local disk

"login"

"open eris"

"lisp"

"<logical volume name>", e.g., "Lisp2"

To start up Lisp using this sysout on another logical volume:

1-Boot the DLion to load Hello from the local disk

"Copy"

"<source logical volume>", e.g. "Lisp2"

"<destination logical volume>", e.g., "Lisp"

"Boot"

"<destination logical volume>", e.g., "Lisp"

H. Interacting with InterLisp-D ž the top level tty window

I. REFERENCES

{eris}<lisp>harmony>doc>GettingStarted.tedit (.press & .ip)

{eris}<lisp>harmony>doc>hello.tedit (.press)

Mesa Users Guide (Chapters 35, 2, & 3) [about Othello]

Xerox 1108 User's Guide [this contains an alternative procedure for using Dandelions]

Mesa Users Guide (Chapters 35, 2, & 3) [about Othello]

Alto User's Handbook

LispCourse #8: Tailoring Parameters, the File Package, Init Files

Tailoring Interlisp to your needs and desires: (FLGS, Parameters, etc.)

Parameters & FLGs

There are literally thousands of atoms in Interlisp whose values are used by the system and/or by various packages to decide how to "behave". By setting the values of these atoms, you can tailor Interlisp to your particular needs and to your particular style of interaction.

For example, the system uses the value of the atom DEFAULTPRINTINGHOST to determine what printer to use to print your output. SETQing DEFAULTPRINTINGHOST to 'Quake will make your output go to Quake, while SETQing it to 'Espresso will direct your output to Espresso. In general, most packages in Interlisp use the value of DEFAULTPRINTINGHOST to determine where to send their printed output. [There is no guarantee of this. But when it comes to printer output, packages are very well behaved.]

DEFAULTPRINTINGHOST functions a *parameter* that you set to specify how you want your system to behave. Its value can be the name of any available printer, or it can be a list of the names of several printers of different types (e.g., a Press, FullPress and an InterPress printer [e.g., (Quake LispPrint: Stinger)]).

Parameters that take on values of T (non-NIL) or NIL are generally known as FLGs since in general their names end in "FLG". For example, PROMPT#FLG determines whether or not the number appears before the back-arrow prompt in your top-level tty window. If PROMPT#FLG is T, the number is there (e.g., the prompt will be "12_"). If PROMPT#FLG is NIL, no number will be printed (e.g., the prompt will be "_").

FLGs are no different from any other parameters. They are just parameters used to make a simple Yes-No decision. FLGs are to parameters what predicates are to functions. Note also that not all parameters that function like FLGs have names ending in "FLG". So it goes.

Note that DEFAULTPRINTINGHOST is a system-wide parameter. Its value is used by Interlisp and all of the various packages.

Default Settings

All parameters and FLGs have default settings. You don't have to fool with a parameter unless you don't like the default setting!

For example, PROMPT#FLG is set to T by the system. You don't even have to know about PROMPT#FLG unless you want to get rid of those pesky little numbers, in which case you have to find out about PROMPT#FLG and set it to NIL.

Parameters like DEFAULTPRINTINGHOST really have to be set differently for every group of Interlisp users since they use different printers. BUT as a user you generally don't have to worry about it (unless your printer is down and you want to change where your printout goes) because it will be set to the proper value in your Init file (see below!!!).

Package-specific Parameters and FLGs

Almost all packages have their own set of parameters and FLGs that can be set to slightly (or sometimes not so slightly) modify their behavior.

Example: DEdit has 3 parameters that determine its behavior.

EDITEMBEDTOKEN ž Value is used as the special "embed token" described last class. Default value (used in examples last class) is &.

DEditLinger ž if T, then when you exit DEdit, the DEdit window will remain on the screen to be used for the next DEdit. If NIL, the DEdit window will be closed when you exit and reopened next time you call DEdit. Note this parameter applies only to the top-level call to DEdit and not to recursive calls which always close their windows. Default value is T. [Note: this is a FLG without "FLG" in its name.]

DEDITTYPEINCOMS ž value is a list that defines the control character commands that you can type in to DEdit. Default value is a list that contains definitions for the Ctrl-F and Ctrl-S described last time. Also contains a Ctrl-Z command not described last time. [Note: this is a parameter for real hackers only.]

Second Example: Chat has 5 parameters. (Note: Chat is a package we haven't yet covered, so if you aren't familiar with it, don't worry. Just look at the kind of things the parameters allow you to tailor).

CLOSECHATWINDOWFLG ž If non-NIL, every Chat window is closed on exit. If NIL, the initial setting, then the primary Chat window is not closed. Default value is ????.

CHAT.FONT ž If non-NIL, the font that Chat windows are created with. If CHAT.FONT is NIL, Chat windows are created with (DEFAULTFONT 'DISPLAY) [To be covered in a later class!]. Default value is ????.

DEFAULTCHATHOST ž The host to which CHAT connects when it is called with no HOST argument. Default value is ????.

CHAT.ALLHOSTS ž A list of host names, as uppercase litatoms, that the user desires to Chat to. Chatting to a host not on the list adds it to the list. These names are placed in the menu that the background Chat command prompts with. Default value is ????.

CHAT.DISPLAYTYPE ž The type of display (a number) that Chat should tell the remote host the user is on. If Datamedia emulation is desired, this variable should be set to the number corresponding to the terminal type Datamedia for the remote host. If the remote host does not respond to the terminal type protocol in Pup Telnet, this variable is irrelevant. Default value is 10.

How to find out about available parameters and FLGs.

Package-specific parameters

Package-specific parameters are usually included in the package documentation.

For example, the DEdit parameters described above are described under the heading "DEdit Parameters" in the DEdit section of the Interlisp Reference Manual (see Section 20.1.4). The Chat parameters are described at the end of the Chat section of the Interlisp Reference Manual (see Section 20.5). They have no

heading, but are preceded by the sentence: "The following variables control aspects of Chat's behavior."

If you want to change the behavior of some particular package, look in the documentation for that package. You will almost certainly find a list of parameters. To find the documentation of a package is another matter. Some are documented in the IRM (Chat, DEdit, TEdit, etc.), the rest are documented in separate documents usually stored in the same directory as their LOAD files. You just have to poke around to find them.

System-wide parameters

Good luck!

Documentation for system-wide parameters is scattered all over the IRM. Moreover, there is no independent listing or description of them in existence ANYWHERE.

If you want to change some system behavior, your only chance is to poke around the IRM, looking for where that behavior is described. Sometimes along with the description there will be a description of some interesting parameters.

You might expect DEFAULTPRINTINGHOST to be documented in the INPUT/OUTPUT chapter (#6) of the IRM. After all most of the printing functions are described there. But in fact, DEFAULTPRINTINGHOST is documented in the Interlisp-D Sepcifics chapter (#18). This fact was easy to discover because I could look up DEFAULTPRINTINGHOST in the index. But if I wanted to find "the parameter that changes my default printer". Hah!

Learn to do the following:

Accept things as they are and don't mess with parameters.

Wander around the IRM looking for interesting parameters and FLGs.

Look at the Init files from lots of experienced users. They are a gold mine of parameters being set.

When you see someone whose system behaves differntly from yours, ask how they did that.

Ask your local Interlisp wizard lots of questions.

Dream up and implement a good, permanent solution to the problem of managing and documenting the thousands of system parameters.

Parameters are great, but only if you can find out about them.

Some More Interesting System-Wide Parameters and FLGs

INITIALSLST ž a list of the form $((\text{UserName1} \text{ FirstName1} \text{ Initials1})(\text{UserName2} \text{ FirstName2} \text{ Initials2}) \dots (\text{UserNameN} \text{ FirstNameN} \text{ InitialsN}))$. The system makes sure that if UserName_i is logs in, then various parts of the system will use FirstName_i as the users first name and Initials_i as the users initials. For example, my Init file sets this parameter to $((\text{Halasz Frank fgh:}))$. Thus the system says "Hi, Frank." when is comes up with a clean sysout. It also puts my initials "fgh:" in a comment in the beginning of every function that I edit. Note the **INITIALSLST** must be set in your Init file or you must run **"(SETINITIALS)"** after setting **INITIALSLST** to a new value so that the system recognizes the changes **INITIALSLST**.

LOGINHOST/DIR ž the host and directory that is considered to be your home directory by various programs. E.g., my Init file sets **LOGINHOST/DIR** to be $\{\text{PHYLUM}\}\langle\text{HALASZ}\rangle$.

DIRECTORIES ž a list of host/directories that the system will look on when it can't find a file. Used by the **LOAD** function and certain other functions, but not by **TEdit**, **COPYFILE** and many other functions. Ususally you just want to add on to the default value of this list. For example, **NoteCards** does the following when it comes up: **"(SETQ DIRECTOIRES (CONS ' {PHYLUM} <NOTECARDS>CURRENT> DIRECTORIES))"**. This just makes sure that the **NoteCards** directories are searched along with all the standard Lisp ones when the system can't find a file.

WINDOWTITLESHADE ž the shade that the right part (i.e., the part to the right of the title) of every window title bar will be. Value can be any shade (i.e., a number less than 65535) but the values of

BLACKSHADE, WHITESHADE and GRAYSHADE are particularly easy shades to use.

???? and many, many more I can't think of now.

The File Package

Introduction

When you `DEFINEQ` a function, the function definition is entered into the system's virtual memory.

When you `SETQ` an atom to a value, the atom-value pair is entered into the system's virtual memory.

When you apply a function or when you evaluate an atom, the system just looks up the function definition or atom in its virtual memory.

All is fine and dandy UNTIL your virtual memory is destroyed; for example, when you copy a clean sysout to your disk and restart Lisp or when your machine bombs leaving an inconsistent vmem.

When your vmem goes away, the effect of your `DEFINEQs` and `SETQs` goes away as well. When a new sysout is loaded (i.e., a "new" vmem is started), you have to redefine all the functions and reset all the variables.

You could type all those `DEFINEQs` and `SETQs` again. But there is an easier way: the File Package.

The File Package helps you write out on a file all the Lisp function calls necessary to redefine a given set of functions and/or to reset the values for a given set of variables.

Then when you load a new vmem (or for that matter, in another person's vmem), `LOADing` this file will cause these `DEFINEQs` and `SETQs` on the file to be evaluated just as if you had typed them in, thus redefining all the functions and resetting the values of all the variables.

So **LOADing** is just the process of evaluating in a bunch of **DEFINEQs** and **SETQs** from a file rather than from the user's type-in.

MAKEFILE and LOAD, with LISTFILES too.

There are two major functions in the File Package:

MAKEFILE ž makes a **LOADable** file. The first argument specifies the name of the file to be made. The making of a **LOADable** file is controlled by the files **COMS** list described in the next section.

LOAD ž reads in a **LOADable** file made by **MAKEFILE** and evaluates all the Lisp function calls thereon. The first argument is the name of the file.

A third, handy function is **LISTFILES** which prints a **LOADable** file on your **DEFAULTPRINTINGHOST** with a handy little index included. Note that **LISTFILES** is a special form (**NLambda** function).

The COMS list

Introduction

The **COMS** list contains a set of commands that the File Package uses to determine what functions, variables, and other Lisp objects you want to appear on a file made by **MAKEFILE**.

Each **LOADable** file (i.e., file made by **MAKEFILE**) has its own **COMS** list.

Terminology: The root part of a file name is the file name without its extension. For example, the root part of **EXAMPLES.LISP** is **EXAMPLES**.

The **COMS** for a file is the value of the atom constructed by adding **COMS** on to the root of its file name.

For example, if the name of a file is **EXAMPLES**, then the **COMS** for that file is the value of the atom **EXAMPLESCOMS**.

Note this limits your choice of file names a bit. You cannot have two **LOADable** files with the same root part. E.g., you can't have an

EXAMPLES.OLD and an EXAMPLES.NEW because they both would want to have the value of EXAMPLESCOMS as their COMS list and hence would interfere with each other.

Warning: This is one place where Interlisp is case sensitive. All loadable files must be named in ALL CAPS. The File Package does not handle non-CAPS well at all. Your file name must be EXAMPLES and you COMS must be EXAMPLESCOMS; Examples and ExamplesCOMS will not work!

The structure of a COMS list

A COMS list has the following form:

```
((FilePackageCommand1 Arg1.1 Arg1.2 ... Arg1.P)
 (FilePackageCommand2 Arg2.1 Arg2.2 ... Arg2.Q)
 ...
 (FilePackageCommandN ArgN.1 ArgN.2 ... ArgN.R))
```

Each clause in this list tells the FilePackage to write something on the file. What is written is determined by the *FilePackageCommandi* that is the CAR of the clause and by the *Args* that are in the CDR of the clause.

There are many possible *FilePackageCommands*.

The most important are:

FNS ž tells the File Package to write function definitions on the file. The FP essentially write a DEFINEQ for every function named in the rest of the list.

Example: (FNS CountList CountAtoms CloseWindowList) tells the FP to write DEFINEQs for the three functions onto the file.

VARs ž tells the file package to write the values of variables on the file. The FP essentially writes SETQs for every variable named in the *Arg* items in the rest of the list.

If a *Arg* is simply an **atom**, then the FP will write a SETQ that sets the atom to the value it had WHEN THE FILE WAS WRITTEN.

So if *Arg* is simply MyVariable and if MyVariable has the value 12 when the file is written, the FP will write (SETQ MyVariable 12) onto the file. When the file is LOADED, MyVariable will be set to 12. BUT if MyVariable had the value 77 when the file was written, the SETQ would be (SETQ MyVariable 77) , with the corresponding change when the file was LOADED.

If a *Arg* is a **list**, then the FP will CONS a SETQ onto this list and write the result down onto the file. For example, if *Arg* is (MyVariable 5). The FP will write (SETQ MyVariable 5) onto the file so that MyVariable will always be set to 5 when the file is LOADED.

For example, (VARS DEFAULTPRINTINGHOST (LOGINHOST/DIR '{PHYLUM}<HALASZ>) (FOUR (PLUS 2 2))) is a FP COMS clause.

ADDVARS ž like VARS except that each *Arg* is a list of two items, the CAR is a variable name and the CDR is a value. ADDVARS write the necessary Lisp code on the file so that when the file is LOADED, the value is added to the list that is already the value of the variable.

Example: (ADDVARS (DIRECTORIES {phylum}<notecards>current>)).

Then when the file is LOADED, the atom {phylum}<notecards>current> will be added to the list that is already the value of DIRECTORIES. If DIRECTORIES had the value ({PHYLUM}<HALASZ> {ERIS}<LISP>) before the LOAD, then after the LOAD it would have the value ({phylum}<notecards>current> {PHYLUM}<HALASZ> {ERIS}<LISP>).

FILES ž tells the file package to write load commands for some other files on this file. The FP essentially writes LOADs for every files named in the rest of the list. When the file is LOADED, it will cause these other files to be LOADED as well.

For example, (FILES NOTECARDS1 NOTECARDS2 NOTECARDS3 NOTECARDSREST) might be a clause in the COMS file for NOTECARDS.LSP. Then whenever NOTECARDS.LSP was LOADED, the files NOTECARDS1, NOTECARDS2, NOTECARDS3, and NOTECARDSREST would also be loaded.

P ž tells the FP to print each *Arg* item in the rest of the list out on the file.

Then when the file is LOADED, these items will be evaluated.

Example: (P (LAFITE 'ON)(PRINT "Hello, Tiger")(CLOSEW LOGOW)) tells the FP to write (LAFITE 'ON), (PRINT "Hello, Tiger"), and (CLOSEW LOGOW) on the file. Then when this file is LOADED, these function calls will be executed causing LAFITE to be turned on, "Hello, Tiger" to be printed in the tty window, and the Interlisp-D logo window to be closed.

* ž tells the FP that this is a comment. * can be used to put comments in your COMS lists so that they are more understandable.

Note that there can be as many FNS clauses as you like in a COMS list. They can be in any order as well. It usually helps to group your functions into FNS clauses in a way that makes some structural sense to you. Ditto for VARS, P, ADDVARS, *.

What does MAKEFILE do?

MAKEFILE looks at the COMS list for the file it is making and simply follows the instructions there.

Actually, the first thing it does is equivalent to (VARS *File*COMS). That is it writes out a SETQ so that the COMS for *File* is reset when it is LOADED. So the first thing you always see in a LOADable file is the SETQ for the file's COMS list.

Then, MAKEFILE follows the FP clauses in the COMS list in order. If the COMS says FNS, MAKEFILE writes out DEFINEQs. If the COMS says VARS, the FP writes out SETQs. And so on.

When its done, it returns the name of the file it just made.

How does LOAD work?

LOAD just reads in the file and evaluates each item (DEFINEQ, SETQ, etc.) in the file, just like "normal" Lisp reads the user's type-in and evaluates it.

How does the user work with the FP?

1. Define a few functions ž say, CarOfListItems, CountList and CloseWindowList.
2. SETQ a few variables ž say, AtomList, WhoCaresList and FranksAge
3. Create a COMS for this file, which we'll call EXAMPLE.LSP.

```
(SETQ EXAMPLECOMS
      '((FNS CarOfListItems CountList CloseWindowList)
        (VARS AtomList (WhoCaresList (LIST 'Who 'Cares))
              FranksAge)))
```

4. Evaluate "(MAKEFILE 'EXAMPLE.LSP)", creating the file EXAMPLE.LSP.
5. Define a new function, say ExampleFunction.
6. Evaluate "(DC EXAMPLE)" which will call DEdit on EXAMPLECOMS.

You can then edit the EXAMPLECOMS list. In particular, add a new clause (FNS ExampleFunction) or add ExampleFunction to the already existing FNS clause.

7. Redo Step 4 to make an updated version of EXAMPLE.LSP.
8. Get a new sysout, i.e., wipe out your vmem.
9. Try to evaluate (CountList '(A B C)). You will get a "undefined function" error because in your clean vmem CountList has not yet been defined.
10. Evaluate "(LOAD 'EXAMPLE.LSP)".
11. Try to evaluate (CountList '(A B C)), resulting in "3". CountList was defined during the LOAD.
12. Change CountList to return 2 times the list length.
13. Evaluate "(MAKEFILE 'EXAMPLE.LSP)" to update EXAMPLE.LSP with this change. No need to remake the EXAMPLECOMS because it was properly set during the LOAD.
14. Define a new function, say LastExample.
15. Evaluate "(DC EXAMPLE)" and edit the COMS to add LastExample to some FNS clause.
16. Evaluate "(MAKEFILE 'EXAMPLE.LSP)" to update EXAMPLE.LSP with this change.

17. You get the picture.

Example LOADable File

A LISTFILES of the EXAMPLE.LSP file (created by "(LISTFILES EXAMPLE.LSP)") is in the Appendix.

The first page is the handy index created by LISTFILES for your convenience in handling large files. The MAKEFILE output really starts on the second page.

IMPORTANT NOTE: There are no SETQs. That's because I've been lying a bit. The FP uses the functions RPAQ and RPAQQ instead of SETQ and SETQQ. Just translate in your head, the RPAQs and RPAQQs into SETQs and SETQQs. For all practical purposes they are the same.

Note that the first variable set in the file is in fact the EXAMPLECOMS.

The FP has ways to make life really easy

FILES?

The function (FILES?) will list in your tty window all functions, variables and other Lisp objects that you have defined but not saved using a MAKEFILE. It will then ask you if you want to say where these things go. If you say Yes, it will ask for each function and variable to what file it belongs. You give it a file name, and it will add it to the appropriate clause in the COMS for the file name.

You can even, specify a new file name and it will create the appropriate COMS list from scratch.

If you give NIL as the file name, it will not add the function or variable to any COMS and it will never ask you about it again.

If you simply type a <return>, it will not add it to any COMS, but will ask again the next time you call FILES?.

MAKEFILES

Does a FILES? to capture all the orphans.

Then for each file that has to be updated for any reason, does a MAKEFILE.

Note: Some users exist on FILES? and MAKEFILES alone and never touch a COMS directly. Other (like me) prefer to keep track of our own changes, use DEdit on the COMS, and then call MAKEFILE ourselves. The choice is yours to make.

Final Note on the FP

I have described about 20% of the FP. But its probably all you need to know for a while.

The FP is documented in glorious detail in Chapter 11 of the IRM.

Section 11.7 is probalby the most interesting because it describes the FilePackageCommands you can put in your COMS lists.

Sections 11.1 and 11.2 cover LOAD and MAKEFILE et al.

At Last, Init Files

An Init file is just another LOADable file. Nothing mysterious about it all. In fact, as LOADable files go its quite dull!

An Init file is a special LOADable file for the following reason:

When Lisp starts up from a clean sysout, the first thing it does is evaluate the function "(GREET)".

GREET immediately goes out and finds a file called INIT.LISP on your local disk (if it can't find it, it asks you to specify where it is).

GREET LOADs INIT.LISP.

Then GREET tries to find another Init file, one that is somehow associated with your login. It looks in a number of places like on your directory on your file server.

The first Init file it finds, it LOADs.

So, Init files are just LOADable files that Lisp always LOADs the first time it comes up from a clean sysout.

There are two Init files:

- 1) A **site-specific Init file** that is common to all users at a given site (e.g., ISL, KSA, etc)

This init file sets things like DEFAULTPRINTINGHOST, DIRECTORIES, FONTDIRECTORIES, CH.NET.HOSTS, etc. All those parameters that have to do with what printer to use, what file server to use, what your ethernet looks like, where to find the local copies of the Lisp files, AND where to look for the user specific init files.

At PARC, the site-specific init files are located on {eris}<lisp>harmony>basics>init.cis (for ISL) and init.ksa for KSA. The appropriate file should be copied to your local disk and renamed to be INIT.LISP.

- 2) A **user-specific Init file** that is setup by each user as he or she pleases

This init file is generally used to set all those parameters and FLGs to make the system behave "correctly", to make the screen look pretty, to load the files for all the packages that are frequently used, etc.

My Init file has four basic clauses:

A **VAR**S clause that sets parameters like INITIALSLST, WINDOWTITLESHADE, DeditLinger, CHAT.FONT, CLOSECHATWINDOWFLG, etc. to my liking.

An **ADDVAR**S clause something like: (ADDVAR (DIRECTORIES {PHYLUM}<HALASZ>LISP>)(DIRECTORIES {PHYLUM}<NOTECARDS>RELEASE1.1>))

A **FILE**S clause that loads CROCK, ARCHIVETOOL, SKETCH, BITMAPFNS, FILEBROWSER, and other such optional packages from either {eris}<lispusers> or from {eris}<lisp>harmony>library>.

A **P** clause something like: (P (CROCK (CREATEREGION 816 687 128 115))(CLOSEW LOGOW)(LAFITE 'ON)) which causes CROCK to start up, the Logo window to close and Lafite to start up when my Init file is LOADED.

At PARC, your Init file should be named with one of the following names:

{ERIS}<user>LISP>INIT.DCOM, {ERIS}<user>LISP>INIT,
 {ERIS}<user>INIT.DCOM,
 {ERIS}<user>INIT.LISP, {PHYLUM}<user>LISP>INIT.DCOM,
 {PHYLUM}<user>LISP>INIT, {PHYLUM}<user>INIT.DCOM,
 {PHYLUM}<user>INIT.LISP, {IVY}<user>LISP>INIT.DCOM,
 {IVY}<user>LISP>INIT, {IVY}<user>INIT.DCOM,
 {IVY}<user>INIT.LISP, where user should be replaced by your login name.

Commonly asked questions:

How do I change my init file?

Use "(DC INIT)" to DEdit the INITCOMS, then do a (MAKEFILE
'{PHYLUM}<user>LISP>INIT) or (MAKEFILE
'{IVY}<user>INIT.LISP) or whatever.

How do I start an Init file?

You have two choices:

Create an INITCOMS from scratch using (SETQ INITCOMS
'((FNS ...))), then edit it using DC, then do a MAKEFILE.

Copy someone else's INIT file to your directory, load Lisp, edit the
INITCOMS using DC to change all the "bad" parameters settings,
then do a MAKEFILE to update the INIT file with your changes.

What do I put in my Init file?

Whatever you like. Generally, you set parameters for the system and
specific packages you commonly use. You also load your commonly used
packages.

Don't keep your own functions in your init file. Put your functions in a
file called MYUTILITIES.LISP or some such, and then put a (FILES
MYUTILITIES.LISP) clause in your INITCOMS so this file gets
LOADed when your Init file gets LOADed.

Final Comments:

1. You don't need Init files. The system runs fine without them. Though, it can't
find printers, etc. because DEFAULTPRINTINGHOST, etc, haven't been set.
The system will run perfectly without your personal Init file. You might have to
load some packages by hand every time the system gets reloaded, but ...
2. If you want to take over an already loaded system from someone else, you can
call (GREET) yourself. This will "ungreet" the previous person and then redo the
GREET sequence with your Init file.

References

The IRM, Chapter 11

{ERIS}<LISP>HARMONY>DOC>INITFILES.TEDIT

The Init file documentation from the Sysdoc group.

Exercises

Define a few functions like CountAtoms and then save them on a file. Get a new sysout and try LOADING the file. Etc.

Make yourself a nice, interesting Init file.

LispCourse #9: Typing Into the TTY Window: TTYIN & the Programmer's Assistant

Introduction

"read-EVAL-print loop" versus "READ-eval-print loop"

Recall the following from class #2:

1. The user interacts with Lisp by typing into the *top-level TTY window* (a.k.a. the *Exec window*).
2. In the TTY window Lisp is running in a read-eval-print loop:
WHILE T
 Read user's typed input
 Evaluate user's input
 Print result of evaluation
3. This loop is really the read-EVAL-print loop because EVALUATION IS THE THING that does all the work in Lisp.

In many Lisps, "read-EVAL-print" is an accurate picture of what goes on. Reading the user input is a trivial operation, evaluation is the important "work-doing" part of the loop.

Interlisp is different. The evaluator is the most important part, **BUT** the read part of the read-eval-print loop in Interlisp is very complex and very powerful and in fact does a lot of work over and above what the Lisp evaluator does.

So, the topic for the next couple of classes is the READ-eval-print loop. Emphasis is on the READ part of the read-eval-print loop and all the help it gives us in interacting with Interlisp.

The complexity issue once again

The mechanism Interlisp uses to read and process your type-in (i.e., *the Lisp Exec*) is very complex. It represents the result of many years of non-coordinated development by many different sets of people.

The mechanism consists of several different packages.

Once again, each package is somewhat different, having different conventions, etc.

Once again, the trade-off between increased functionality and ease of use/learnability has been settled in favor of functionality.

BUT, if you accept things as they are and simply memorize the few things you deal with frequently, the Lisp Exec can help you A LOT in doing your work!!!!

An overview of how Interlisp processes user input (i.e., the Lisp Exec)

The first page of the Appendix contains a flow diagram of how Interlisp processes type-in in the Exec window.

The diagram illustrates the following components:

Most characters typed in by the user are input into a small text editor called the **TTYIN** editor. The TTYIN editor allows you to make changes to your type-in before it is actually processed by Lisp.

Note that some characters (e.g., Ctrl-D and Ctrl-E) don't go to TTYIN. These characters are special interrupt characters that can interrupt or abort Lisp at any time. Thus they are handled by a special **Interrupt mechanism**.

After the input in the TTYIN editor is "completed" [e.g., by typing a <RETURN> or by entering the last ")" in a list], the input is passed on to the **Programmer's Assistant**.

The P.A. is a general purpose "assistant" that keeps a history of your type-in, allows you to repeat or undo previous function calls, provides several short-hand methods for typing in Lisp function calls, and so on.

When the P.A. receives a complete user input, it determines the nature of the input.

If it is a P.A. command (e.g., a history reference), then it executes the command.

If it is P.A. short-hand for some Lisp expression, then it translates the expression to a standard Lisp expression and sends it on to the Lisp read mechanism.

If it is a standard Lisp expression, it just passes it on to the Lisp read mechanism.

The **Lisp reader**, accepts standard Lisp expressions and expands any Lisp short-hand they may have. For example, if the input contains a '(A B C), the Lisp reader will change this to (QUOTE (A B C)) as required by the evaluator.

The Lisp reader, then passes the standard Lisp expression to the **Lisp Evaluator**. If all is correct, then the Lisp evaluator does its work as we discovered in some detail a couple sessions ago.

If the evaluator discovers an error in the Lisp expression, it passes the error off to the **error handler** which tries to handle the error in one of several ways:

By getting **CLISP** to translate the syntactic sugar into "real" Lisp.

By getting **DWIM** to automatically correct the error.

By calling on the **Break Package** to open a break window so that the user can correct the error or abort the operation.

Note that in our earlier view, the Lisp evaluator was the only thing that did "work" in Lisp. That is, when we give Lisp a "command", we said it was the evaluator which carried out the command.

According to this diagram, there are several components that can be seen as carrying out "work", i.e., carrying out user commands. In particular, there are Programmer Assistant commands that are separate from Lisp function calls and that allow you to do certain kinds of "work".

Note that in reality it is the Lisp Evaluator that's doing all of the work. The P.A. is just a Lisp program. So, the process of interpreting and carrying out P.A. commands eventually reduces to Lisp being evaluated by the Lisp evaluator.

Today's topics: The main path through TTYIN and the P.A.

The diagram on the second page of the Appendix contains a simplified version of the complete diagram.

When all is well and there are no errors and no interrupt characters, this represents the processing Interlisp does to your type-in. The major components are the TTYIN editor and the Programmer's Assistant.

We'll cover these two parts today. Next time we'll cover the Lisp READER and the Interrupt mechanism as well as DWIM and CLISP.

The following time we'll cover the Error Handler and the Break Package.

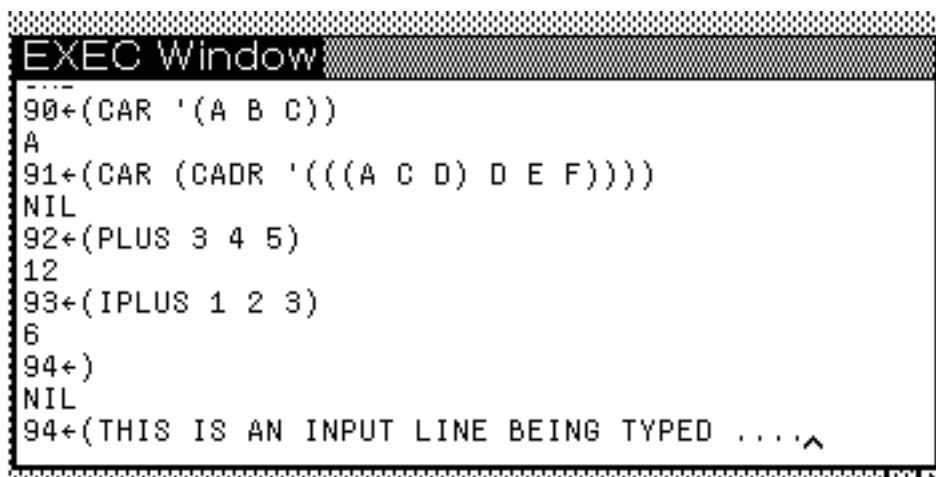
The TTYIN type-in editor

The Basics

The TTYIN editor is a simple text editor that uses both the mouse and key commands.

The editor operates in an *active region* that contains only the current type-in line(s) within the Lisp Exec window. Thus, you can only edit the current type-in and no other text appearing in the Lisp Exec window.

If "94_" is the current uncompleted input line:

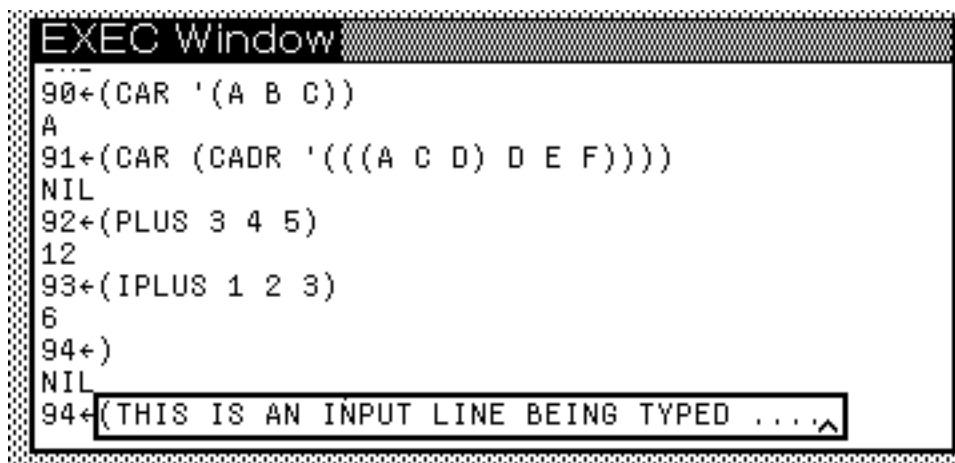


```

EXEC Window
---
90+(CAR '(A B C))
A
91+(CAR (CADR '(((A C D) D E F))))
NIL
92+(PLUS 3 4 5)
12
93+(IPLUS 1 2 3)
6
94+
NIL
94+(THIS IS AN INPUT LINE BEING TYPED ... ^

```

Then the boxed area is the region in which TTYIN is active:



```

EXEC Window
---
90+(CAR '(A B C))
A
91+(CAR (CADR '(((A C D) D E F))))
NIL
92+(PLUS 3 4 5)
12
93+(IPLUS 1 2 3)
6
94+
NIL
94+(THIS IS AN INPUT LINE BEING TYPED ... ^

```

TTYIN is always in insert mode: your type-in is inserted just before the *blinking caret* on the input line.

While you are typing in TTYIN helps you balance parentheses. Every time you type a ")" or a "]", the *blinking caret* moves to the matching "(" or "[". It remains there for 1 second or until you type a character and then returns to its previous location. Note that any characters you type when the caret is in this "balancing indicator" mode will go at the previous location of the caret and not at its current location near the balancing "(" or "[".

Example: The ")" has just been typed. The left arrow points to the caret in "balancing" mode. The right arrow points to the insertion point. After one second, the caret will return to the insertion point.

```

EXEC Window
31+(BITMAPCOPY (CAR DEFAULTCARET))
{BITMAP}#5,46762
32+(SETQ XX IT)
{BITMAP}#5,46762
33+(EDITBM IT]

34+(SETQ XX)
(XX reset)
NIL
35+(PLUS (PLUS (PLUS 2 3))

```

The editor remains in operation in the type-in active region as long as the type-in remains "uncompleted". When the input is completed, TTYIN sends it along to the Programmer's Assistant and you can no longer edit it.

TTYIN's Definition of "Complete" Input

Completed input is defined similarly to the DEdit type-in:

An input beginning with a "(" is complete when a balancing or over-balancing ")" is typed.

Examples:

(A B C) is complete

(A (B C) is not complete

A "]" balances any number of "(", until the most recent "[".

Examples:

(A (B (C D) is complete

(A [B (C D) is not complete

(A [B (C D)) is complete

A <RETURN> causes a new line to be started in the Exec window, but does not complete the current input.

```

EXEC Window
UNDEFINED CAR OF FORM
A
96+SET[A B]
(A reset)
B
97+SET(A B)
B
98+SET (A B)
B
99+(THIS IS AN INPUT LINE BEING TYPED
AND THIS IS THE SECOND LINE ... ^

```

An input beginning with an atom completes as follows:

If the first atom is immediately followed by a "(" or a "[", the input completes when this a "(" or "[" is balanced by a ")" or a "]"".

Examples:

SET[A B] is complete

SET(A B) is complete

SET (A B) is not complete

SET(A (B C) is not complete

The input always completes when an over-balancing ")" or "]" is typed.

Examples:

SETQ A B] is complete

SETQ A B) is complete

SET (A B)) is complete

SET(A (B C) is not complete

The input always completes when all parentheses are balanced and a <RETURN> is typed.

Examples:

SET (A B)<RETURN> is complete

SET (A B<RETURN> is not complete

PLUS 2 3<RETURN> is complete

(PLUS 2 3<RETURN> is not complete

If all parentheses are **NOT** balanced and a <RETURN> is typed, then input continues on the next line:

```

EXEC Window
20+SET (A B
]
B
29+PLUS 2 3
= (PLUS 2 3)
5
30+(PLUS 2 3
30+LIST (A B
C D E F^

```

Mouse-based editing in TTYIN

The most convenient way to edit text in TTYIN is with the mouse.

The mouse buttons work as follows:

LEFT ž used to change the place where text will be inserted when you type. When the LEFT button is held down, the TTYIN caret will follow the mouse cursor as it moves about the TTYIN active region. If the mouse cursor moves out of the active region, this tracking stops until the cursor moves back into the active region.

MIDDLE ž Same as LEFT, except that the caret moves only to word boundaries (i.e., just before or just after any word in the active region,

where words are separated by spaces, tabs, periods, commas, etc. as well as the various types of parentheses.)

RIGHT ž used to delete the text from the caret position to the mouse cursor position. As long as you hold down the **RIGHT** button with the cursor in the active region, any text between the caret and the cursor is complemented (i.e., reverse-videoed). When you let up on the **RIGHT** button, the complemented text (i.e., the text between the caret and cursor at that point in time) is deleted.

If you move out of the active region while holding down the **RIGHT** button, the text complementing is stopped until you move back into the active region with the button down. You can **abort** the delete by letting up on the **RIGHT** button while outside the active region.

Copy, Move and Delete Text

Copy, Move and Delete selections are also available in TTYIN.

To make these secondary selections you hold down the **SHIFT** and/or **CTRL** keys [see below] while making and (optionally) extending a selection with the mouse.

In a secondary selection the mouse buttons work as follows:

LEFT ž selects the indicated letter

MIDDLE ž selects the indicated word

RIGHT ž extends a selection just made with the **LEFT** or **MIDDLE** buttons. The selection is extended from the previous selection to the current cursor position.

In all cases, the selection takes effect when the mouse button is *released*. Moving within the active region with the mouse button held down, moves the selection. Moving out of the active region and letting up on the mouse button, aborts the selection.

The **SHIFT/CTRL** keys are used to determine whether the secondary selection is a copy, a move, or a delete.

Copy ǰ holding down the **SHIFT** key and making a secondary selection will copy the selected text and insert it at the blinking caret. During the secondary selection, the text to be copied is underlined with a dotted line.

Move ǰ holding down the **SHIFT and CTRL** keys while making a secondary selection will move the selected text to just after the blinking caret. During the secondary selection, the text to be moved is complemented.

Delete ǰ holding down the **CTRL** key while making a secondary selection will delete the selected text. During the secondary selection, the text to be deleted is complemented. Note: this is very similar to the RIGHT button primary selection described above.

Copy, Move, and Delete **do not** take effect until you let up on the **SHIFT** and **CTRL** keys. You are free to select and reselect while the **CTRL** and/or **SHIFT** keys are down.

To abort a Copy, Move, or Delete just move the mouse outside the active region while holding down the **LEFT** or **MIDDLE** buttons and then let up on the **CTRL** and **SHIFT** keys.

Important Note: Pressing the middle-blank key on Dolphins/Dorados or the **OPEN** key on the Dandelions will retrieve the last bit of deleted text and insert it at the blinking caret. Hence, these keys can be used to **UNDO** an mistaken deletion.

Key-based editing in TTYIN

TTYIN has a full-complement of key-based editing commands. These are mainly meant for use on Interlisp-10 and Interlisp-Vax where there are no mice and bit-mapped displays.

We will cover only those key commands that are handy for Interlisp-D users with mice. Refer to the IRM (Section 20.7) for a full-description of all of the available commands.

Key commands can be typed at any time while using TTYIN.

The useful commands are:

BS, Ctrl-A ž deletes the character just before the blinking caret.

Ctrl-W ž deletes text from blinking caret backwards to nearest beginning of a word. Word beginnings are usually marked by a space, period, or parenthesis.

Ctrl-Q ž deletes the line of text containing the caret. If this line is blank, Ctrl-Q will delete the previous line. So several Ctrl-Qs can be used to delete all of the lines in a multi-line type-in.

Ctrl-X ž moves the blinking caret to the end of your current input. If the parentheses in the input are balanced or over-balanced will "complete" the current input.

Middle-blank/OPEN ž

If the last command was a Ctrl-A, a BS, a Ctrl-Q or a Ctrl-W, undoes that command.

Otherwise, retrieves the last text deleted using the mouse and inserts it at the blinking caret.

ESC ž Tries to complete the word that the caret is in. For example, typing in "NC.T<ESC>" might result in "NC.TestFunction" because the only "known" word matching "NC.T ..." is "NC.TestFunction".

TTYIN searches a list of words (the value of USERWORDS) which contains all the words recently defined with a DEFINEQs or set with a SETQ, and so on. If TTYIN cannot find a completion, it places an ESC in the text (which appears on the terminal as a "\$"). If TTYIN finds more than one possible completion, it flashes the window.

Very specialized stuff -- Meta-key commands

Most of the more specialized commands in TTYIN are meta-key commands. To use them you must use the Meta-key as well as the commands key (or keys).

The Meta-key works in one of two ways:

1. Meta can work like a CTRL key in that you depress the Meta key and the command key simultaneously.
2. Meta can be a prefix key where you hit the Meta key first and then the command key.

The default Meta key is Top-Blank on Dolphins/Dorados and KEYBOARD on Dandelions. This is a *prefix* meta-key. To change this to a simultaneous Meta-key, see section 20.7.3 of the IRM.

Thus by default, the meta-key commands are activated by pressing Meta followed by the command key(s).

The following are interesting Meta-key commands:

Meta-U ž upper-cases the word containing the caret.

Meta-L ž lower-cases the word containing the caret.

Meta-C ž capitalizes the word containing the caret.

Meta-Ctrl-Y ž lets you talk to a recursive call to Lisp. You can do anything you want in this sub-Lisp (including do a further recursion by typing Meta-Ctrl-Y to the sub-Lisp). When you are done, type "OK<RETURN>". This will return you to the current input at the point you typed Meta-Ctrl-Y.

Concluding remarks for TTYIN

90% of my editing with TTYIN is done using the normal Left and Right mouse button operations, BS, and Ctrl-X.

Another 5% is covered by the ESC completion and CTRL-Q commands.

The rest of the stuff I use very little. It comes in handy when I need it, but I could easily live without it.

TTYIN Documentation

TTYIN is documented in Section 20.7 of the IRM. Sub-sections 20.7.1 through 20.7.3 are most relevant to the non-programming user.

Section 20.7.10 described the parameters and FLGs for TTYIN, most of which are fairly technical and none of which are especially useful.

The Programmer's Assistant

The Programmer's Assistant is a tool that helps you manage your interactions with Interlisp. The P.A. basically offers three services to the Interlisp user:

1. Alternative syntactic forms for some common, but clumsy Lisp syntax.
2. A history mechanism that keeps a record of the user's interactions with Lisp and allows the user to undo and/or redo interactions or sequences of interactions.
3. A set of special P.A. (and LISPX) commands that by-pass (to some extent) the normal Lisp evaluation mechanism, allowing the user to carry out various tasks for which there are no specific Lisp functions or for which the Lisp functions are particularly clumsy to use.

Commentary: Except for the history list and a bit of the alternative syntax, most of this stuff is seldom used. But it is there and you do occasionally have to interact with the P.A. or with someone using the P.A., so its best to have some familiarity with how it works.

The Good Stuff: The History Mechanism

The P.A. maintains a record of your inputs and the results of those inputs. An input and its results are called an **event**. The P.A. automatically maintains a record of the last 100 or so events. Through a series of P.A. commands, you have access to these 100 events. You can undo or redo any of the events.

All events have an event number. This is the number that is printed before each input in the Exec window, e.g., "12_" is event number 12.

EventSpecs: All history P.A. commands refer to an event using a EventSpec which can be one of the following:

N (i.e., a positive integer) ž refers to event number N.

žN (i.e., a negative integer) ž refers to N events prior to the current event being input. E.g., -1 is the previous event and -3 is three events back.

Pattern ž refers to the last event that matches the given pattern. The allowable patterns are as defined in the DEdit Find command and are described on page 17.13 of the IRM.

Examples:

ABC refers the last event that used ABC as an atom or function name.

SETQ refers to the last event using SETQ.

SET<ESC> refers to the last event using any of SET, SETQ or SETQQ, etc.

(LIST & &) refers to the last event that called LIST with two arguments.

Empty (i.e., nothing) ž refers to event -1, i.e., the last event.

Compound Event Specs: EventSpecs can be combined to refer to ranges of events. The combinations are:

EventSpec1 THRU EventSpec2 ž refers the events from *EventSpec1* through *EventSpec2* inclusive, e.g., "47 THRU 49" refers to event 47 through event 49.

EventSpec1 TO EventSpec2 ž refers to the events from *EventSpec1* to *EventSpec2* non-inclusive of *EventSpec2*, e.g., "47 TO 49" refers to event 47 through event 48.

ALL *EventSpec* ž refers to all events matching *EventSpec* rather than just the last one, e.g., "ALL SETQ" refers to all events with SETQ in them.

EventSpec1 AND EventSpec2 ž refers to *EventSpec1* and *EventSpec2*, e.g., "SETQ AND 47 AND 34" refers to the last event with SETQ, event 47, and to event 34.

P.A. History Commands: The following are P.A. commands that refer to the history list. To execute, a P.A. command you can just type it into the Exec window terminated by a <RETURN>. (See below for a discussion of P.A. commands in general.)

?? *EventSpec* ž prints the event or events referred to *EventSpec*. Each event is printed with the user's input followed by Lisp's response to this input. If any event ended in an error, the Exec window will be flashed when this event is printed. "??" alone will print the entire history list. "?? - 1" will print the last event.

UNDO *EventSpec* ž UNDOes the events specified by *EventSpec*. Not all events can be undone. However, the P.A. takes every effort to insure that events the user types in are in fact undoable. For example, SETQ and DEFINEQ are undoable. However, COPYFILE is not undoable since Lisp doesn't know how to undo a file copy.

Examples:

```
1_ (SETQ A 5)
5
2_ (SETQ A 10)
10
3_ A
10
4_ UNDO 2
SETQ undone.
5_ A
5
6_ UNDO UNDO
UNDO undone.
7_ A
10
```

REDO *EventSpec* ž REDOes the events specified by *EventSpec*. REDOing an event has the same effect as if you retyped the same input again.

Examples:

```
8_ (SETQ A '(A B C D E F))
(A B C D E F)
9_ (SETQ A (CDR A))
(B C D E F)
10_ REDO
(C D E F)
11_ UNDO
REDO undone.
12_ A
(B C D E F)
13_ REDO SETQ
(C D E F)
14_ REDO 9
```

(D E F)

FIX *EventSpec* ž If *EventSpec* refers to a single event, will reprint the input for that event in the Exec window. You can then use TTYIN to edit the input and then redo the event.

If *EventSpec* refers to more than one event, the P.A. will pop you into DEdit with a list of the events' inputs. You can then edit this list. When you exit DEdit normally (i.e., with OK or Exit), then the P.A. will redo all the events in the list with the modifications made in DEdit. If you exit DEdit with STOP, then the P.A. will do nothing.

USE *New* FOR *Old* IN *EventSpec* ž REDOes every event in *EventSpec* after substituting every *New* for every *Old* in the event. Note that *New* and *Old* can be sequences, in which case *New1* is substituted for *Old1*, *New2* is substituted for *Old2*, and so on. Note that both *Old* and *New* may contain wildcards and other special pattern match characters as specified in Section 17.4 of the IRM.

Example:

```

14_ (SETQ FIE 'AAA)
AAA
15_ (SETQ FOO 'BAR)
BAR
16_ USE BAZ FOR BAR IN -1
BAZ
17_ FOO
BAZ
18_ (SETQ FOO 'JAR)
JAR
19_ USE FIE FOR FOO IN 15
BAR
20_ FOO
JAR
21_ FIE
BAR
22_ USE FIZ BANG FOR FIE BAR IN SETQ
BANG
23_ (LIST FIE FOO FIZ)
(BAR JAR BANG)

```

Final Notes:

The History mechanism is real, real handy. Learn to use it effectively!!!!

HistMenu: There is a menu-based interface to the history mechanism available as a LispUser package on {eris}<lispusers>HistMenu.Dcom (&.press, .tty for documentation). I find this interface much more difficult to use than the P.A. command interface, but not everyone agrees. Load HistMenu and try for yourself. **BUT NOT UNTIL YOU BECOME THOROUGHLY FAMILIAR WITH THE HISTORY MECHANISM.**

Documentation: Can be found in Section 8.2 in the IRM. There's lots of details about the history mechanism there that we haven't even touched on here!

Parameters: See Section 8.3 of the IRM for a discussion of the parameters that effect the history mechanism.. In particular, the function (CHANGESLICE ...) can be used to alter the number of events recorded in your history list.

Other parts of the P.A.: Alternative syntax

The P.A. allows several variants on standard Lisp syntax. It accepts this syntax and then translates it into standard Lisp to be passed to the Lisp evaluator.

The syntax rules for the P.A. are the following:

Standard Lisp Format: If the input is a single atom or a standard Lisp expression beginning with a "(" or "[" and ending in a balancing ")" or "]", then this is standard Lisp syntax which the P.A. will pass directly to the Lisp evaluator (with the very important exceptions about P.A. commands noted below!!!).

EVAL-QUOTE Format: If the input begins with an atom immediately followed by a single list, then the atom is assumed to be a function to be applied to the quoted (i.e., unevaluated) elements of the list. This is commonly known as EVAL-QUOTE format, but as APPLY-format in the IRM. The EVAL-QUOTE expression is translated into its standard Lisp equivalent before being passed to the Lisp evaluator.

Examples:

LIST(A B) => (LIST (QUOTE A)(QUOTE B))

SET(A B) => (SETQ A (QUOTE B)) or to (SETQQ A B)

COPYFILE[FOO BAR] => (COPYFILE 'FOO 'BAR)

CONS[A (B C D)] => (CONS 'A '(B C D))

Other formats: If the input begins with an atom followed by a space and then zero or more other atoms and lists, then the P.A. uses the following rules:

If input is a single atom, assume it is a single atom and ignore the space.

If input is two atoms, then APPLY the function named by the first atom to NIL.

If input is an atom followed by a list, then assume it is EVAL-QUOTE format with an extra space.

If input has three or more atoms and lists, then wrap parentheses around the beginning and the end of the list.

Examples:

Atom<space> => Atom

Function Argument => (Function)

MINUS 7 => (MINUS) {resulting in non-numeric arg error}

LIST A => (LIST) {resulting in NIL}

LIST (A B) => LIST(AB) => (LIST 'A 'B)

PLUS 7 8 => (PLUS 7 8) {resulting in 15}

PLUS (MINUS 7) 8 => (PLUS (MINUS 7) 8)

Except for the EVAL-QUOTE format, syntax other than the standard Lisp syntax is seldom used. The rules are too arcane to be effectively used.

EVAL-QUOTE format is handy and is frequently used. It cuts down on the number of QUOTES and 's you have to type. BUT BE CAREFUL!!! EVAL-

QUOTE format can lead to unexpected results when used with NLambda functions that evaluate their arguments.

Example:

SET(A B) sets A to B as expected, but SETQ(A B) sets A to the value of B just as does (SETQ A B)!! This is because the function SETQ evaluates its second argument by itself.

Other parts of the P.A.: P.A. Commands and LISPX macros

The P.A. has a set of commands that the user can execute to do various things. Most of these commands deal with manipulating the history list and are covered in detail above. However, there are some non-history P.A. commands which will be described briefly here.

Examples of P.A. commands include *DIR*, *CONN* and *FB*.

The user or programmer can also create his or her own P.A. commands called LISPX macros. These LISPX macros are indistinguishable from the built-in P.A. commands. Therefore, both the built-in P.A. commands and the LISPX macros will be called P.A. commands here.

P.A. commands work as follows:

For all input, the P.A. checks to see if the first atom in the input matches a P.A. command.

If the first atom is a P.A. command, the P.A. command is executed using the rest of the input as its argument. No evaluation is carried out on the rest of the input.

If the first atom is not a P.A. command, the input is processed as described above and sent on to the Lisp evaluator.

NOTE: If an atom is a P.A. command, it will always be interpreted as a P.A. command when the first atom in an input. It will mask the value and function definitions of the atom from the user.

Example:

If I create a P.A. command called SETQ, then "(SETQ A B)" will always cause the *P.A. command* called SETQ to be carried out and never the function SETQ.

But note that "(NULL (SETQ A B))" will cause the *function SETQ* to be evaluated because SETQ is not the first atom in the input.

Bizzare!!!!

A P.A. command can be entered as either a series of atoms or as a standard Lisp expression. All that matters is that the first atom be the P.A. command.

Example:

DIR is a P.A. command.

DIR {phylum}<halasz> is equivalent to *(DIR {phylum}<halasz>)*

"Interesting" P.A. Commands

There are no truly interesting P.A. commands. Any task worth doing is better implemented as a full-fledged Lisp function!!!

But, here are some that aren't:

DIR ž lists the files matching the given file specification.

CONN ž "connects" to the given directory, e.g. "CONN {phylum}<halasz>".

FB ž starts a FileBrowser on the files described, e.g., "FB {DSK}".

PL ž prints the property list of the given atom, e.g., "PL NOTECARDS".

; ž ignores the rest of the line, allowing the user to type a comment that will show up, for example, in a dribble file.

Documentation on the Programmer's Assistant

The Programmer's Assistant is described in some detail in Chapter 8 of the IRM. Sections 8.1, 8.2, and 8.3 may contain some useful information, though much of it may be difficult to understand for beginners. Sections 8.4 and beyond: don't even bother looking there.

References

The aforementioned chapters and sections of the IRM.

Also see some of the references at the end of Section 1 of the old (1975) IRM. There are pointer to articles about the original design of the P.A., which in its time was quite a novel idea.

Exercises

Fool around with the TTYIN editor trying out some of the features.

Fool around with EVAL-QUOTE format.

Fool around with the history mechanism.

Do the old excercises you never did!

LispCourse #10: Miscellaneous Issues Regarding Type-in

READ macros

In general, the Lisp reader just takes Lisp expressions from the P.A. and passes them on to the Lisp evaluator.

However, the Lisp reader can be set to give special interpretation to certain characters or character sequences it reads in. When it receives a Lisp expression containing one of these special characters or character sequences, it carries out a prespecified action usually resulting in some change to the Lisp expression it received.

The character sequences that are special to the Lisp reader together with their associated actions are known as *READ macros*.

READ macros can be arbitrarily defined by any user. But doing so requires a fair understanding of programming. So we won't discuss it here.

However, there are three READ macros that are predefined in the standard Lisp reader. The READ macro characters are `"'"`, `"|'"` and `"Ctrl-Y"`. When these characters are typed in a Lisp expression, the reader carries out the following actions:

`' ž` places the immediately following (i.e., with no spaces) S-expression within a call to QUOTE. For example, `'A` becomes `(QUOTE A)` and `'(A B C)` becomes `(QUOTE (A B C))`. Note that this only happens if the `"'"` is preceded by a space or a parenthesis and followed by an S-expression. Otherwise, the `"'"` is not altered by the reader.

`|' ž` places the immediately following (i.e., with no spaces) S-expression within a call to BQUOTE. For example, `|'A` becomes `(BQUOTE A)` and `|'(A , B C)` becomes `(BQUOTE (A B C))`. Note that this only happens if the `"|'"` is preceded by a space or a parenthesis and followed by an S-expression. Otherwise, the `"|'"` is not altered by the reader.

Note: BQUOTE is like QUOTE except that it allows exceptions. Any element in a BQUOTEd list that is preceded by a `" ,"` IS evaluated. The rest of the elements are not evaluated (as in QUOTE). Example: `(BQUOTE (A , B C))` would evaluate to `(A 7 C)` when B had a value of 7.

If B had a value of (A B C), then (BQUOTE (Q , B)) would evaluate to (Q (A B C)).

Ctrl-Y \checkmark causes the immediately following S-expression to be evaluated before the whole expression is passed to the Lisp evaluator. For example, if NEATO has the value 7 then the expression (*LIST 2 3 ^YNEATO ^Y(PLUS44 3)*) will be changed to (*LIST 2 3 7 47*) BEFORE it is passed to the Lisp Evaluator.

Note: this functionality is seldom used by mere mortals.

Documentation: Read macros are documented in Section 6.6.3 of the IRM. But the documentation is aimed mainly at programmers

Final Note: Read macros need seldom be worried about. You use "" so often that it becomes second nature. "|" and "Ctrl-Y" are never used by the non-programmer.

But some documentation mentions Read macros and every once in a while you'll here a mention in the hall. Now you know what they are.

Redefining the keyboard

All of the keys on the Inerlisp-D keyboard are *soft* keys. The interpretation of each key on the keyboard is not fixed, but can be changed by the user.

For example, the "A" key defaults to producing an "a" when the SHIFT key is up and an "A" when the SHIFT key is down.

While, it is unlikely anyone would want to change the "A" key, users sometimes wish to change the interpretation of the non-alphanumeric keys such as the blank keys on the right of the Dolphin/Dorado keyboard or the row of editing keys on the top of the Dandelion keyboard.

KEYACTION is the function that can be used to change the interpretation of a key. Keyaction takes two arguments: a *KeyName* and an *Action*.

KeyName is the name of the key whose interpretation is to be changed. Most keys are named by the character (or any of the characters) that appears on them. For example, the "A" key is named "A" or "a", while the "1" key is named "1" or "!".

Special keys are named in the obvious way: For the Dolphin/Dorado they are: TAB, LF, DEL, BLANK-TOP, BLANK-MIDDLE, BLANK-BOTTOM, SPACE, LSHIFT, RSHIFT, CTRL, ESC.

For the Dandelion, the special keys are SKIP, NEXT, UNDO, MOVE, MARGINS, FIND, LARGER, SMALLER, etc.

Action is either NIL or the action that should be taken when the key is pushed down and when it is let up.

If *Action* is NIL, then KEYACTION just returns the current interpretation of the key.

Otherwise, *Action* is of the form (*DownAction* . *UpAction*), where *DownAction* is the specification of what should happen when the key is pressed down and *UpAction* is a specification of what should happen when the key is let up.

Each action specifications can be one of the following:

NIL ž specifies that no action should be taken.

(*Character ShiftedCharacter LockedShifted?*) ž *Character* and *ShiftedCharacter* are either single characters or ASCII character codes standing for characters. *Character* is the code to be transmitted when the key is pressed or let up without the SHIFT key down. *ShiftedCharacter* is the code to be transmitted when the SHIFT key is down.

LockedShifted can be either LOCKSHIFT or NOLOCKSHIFT, indicating whether the *Character* or the *ShiftedCharacter* is to be transmitted when the LOCK SHIFT is down. For example, alphabetic keys are usually LOCKSHIFT and numeric keys are usually NOLOCKSHIFT. When the LOCK SHIFT is down, the "1" key transmits a "1" and not a "!", but when the SHIFT is down it transmits a "!". The "A" key transmits a "A" when both the SHIFT and the LOCK SHIFT are down.

LOCKUP, LOCKDOWN, CTRLUP, CTRLDOWN, METAUP, METADOWN, 1SHIFTUP, 1SHIFTDOWN, 2SHIFTUP, 2SHIFTDOWN ž when one of these is used as an action specification it changes the internal variables in Lisp that determine whether characters will be transmitted as Shifted, Control, Meta, and or LockShifted characters.

For example, if a key is specified as having a down action of LOCKDOWN, the pressing this key will be like pressing down the LOCK key on most keyboards. If another key has a down action of LOCKUP, the *pressing down* this key will be like letting *up* on the LOCK key.

Examples:

To set the "A" key to its ordinary interpretation:

```
(KEYACTION 'A '((a A LOCKSHIFT) . NIL))
```

To set the "1" key to its ordinary interpretation:

```
(KEYACTION '1 '((49 ! NOLOCKSHIFT) . NIL))
```

To set the "A" key so that it works when the key is let up rather than when it is pressed down:

```
(KEYACTION 'A '(NIL . (a A LOCKSHIFT)))
```

To set the "A" key so that it works BOTH when the key is let up and when it is pressed down:

```
(KEYACTION 'A '((a A LOCKSHIFT) . (a A LOCKSHIFT)))
```

To set the "A" key work only when pressed down, but to transmit a "B" instead of an "A":

```
(KEYACTION 'A '((b B LOCKSHIFT) . NIL))
```

On a Dandelion to make the LOCK key be the CTRL key:

```
(KEYACTION 'LOCK '(CTRLDOWN . CTRLUP))
```

To make the BS key transmit a BS (Ctrl-A) when unshifted and a BackWord (Ctrl-W) when shifted, both to take effect on the down press:

```
(KEYACTION 'BS '((1 23 NOLOCKSHIFT) . NIL))
```


Documentation: KEYACTION is documented on page 18.8 of the IRM.

CHARCODE is a function that returns the ASCII code of a character. This is useful in figuring out the *Character* and *ShiftedCharacter* arguments in KEYACTION.

CHARCODE takes one argument, a character. Since it is an NLAMBDA function, this character need not be quoted. CTRL characters are indicated by a "^" prefix, as in "^A". META characters are indicated by a "#" prefix as in "#A" or "#^A". Certain special characters have their own names such as CR, LF, SPACE, ESC, BS, TAB, and DEL.

Examples:

```
(CHARCODE BS) returns 8
(CHARCODE A) returns 65
(CHARCODE a ) returns 97
(CHARCODE ^A) returns 1
(CHARCODE #A) returns 193
(CHARCODE #^A) returns 129.
```

Documentation can be found on page 2.12 of the IRM.

The TTY Process

At any given time, there are several things going on in your Interlisp-D environment. For example, you may be editing a file in a TEdit window while you are copying another file in your Exec window. Also the clock window is constantly updating the time and Lafite is constantly checking if you have new mail.

Roughly speaking, each thing that is going on in your environment at one time is called a *process*. The Interlisp-D environment is said to allow *multiple processes*, i.e., many things going on simultaneously.

A process usually is associated with one or more windows. For example, each TEdit is a process and is associated with a TEdit window. There are, however, process with several windows as well as processes without any windows.

Understanding multiple processes, how to manage them and how to use them effectively is the topic for another whole session.

However, we need to know about one special process called the *TTY process*.

At any given time there may be many processes running in your environment, but there is only one keyboard. Therefore, there is only one process that can receive input from the keyboard at any given time. Interlisp handles this problem by allowing only one process at a time to "own" the keyboard.

The process which "owns" the keyboard at a given moment is known as the *TTY process*. When the user types at the keyboard, the typed input gets sent to the TTY process for processing. If the TTY process is the Lisp exec, then the input gets evaluated. If the TTY process is a TEdit, then the input gets entered into the file being edited.

The TTY process can be moved from process to process in many ways under both user and program control.

Clicking with the mouse in a window usually makes the process associated with that window become the TTY process. For example, if you have two TEdits running. If you click in the first TEdit window, your type in will be dispatched to that TEdit. However, if you then click in the second TEdit window subsequent type-in will be dispatched to the second TEdit. If you then click in the Exec window, subsequent type-in will go to the Lisp Exec.

A window whose process is the TTY process usually contains a blinking caret that indicates where the type-in will be put. Usually, windows that are not associated with the TTY process have a caret that is not blinking. Therefore, the blinking caret generally indicates which window contains the TTY process.

The concept of a TTY process should become second nature to you as you use the Interlisp environment.

Interrupt Characters

Some characters, known as *interrupt characters*, bypass the normal Lisp type-in processing altogether. Immediately after these characters are typed-in, they are dispatched to the interrupt mechanism. The interrupt mechanism then interrupts the ongoing Lisp processing. How processing is interrupted is determined by what specific interrupt character was typed.

Interrupt characters generally affect only the current TTY process. Moreover, which interrupt characters are activated at a given time is determined by the TTY process at that time. Many programs (e.g., TEdit) turn off all or certain interrupt characters while they are the TTY process. Thus interrupt characters may differ depending on what programs are running and what programs have the TTY process.

There is a default set of interrupt characters that are generally in effect when you are typing into the Lisp Exec window. These characters are the following:

Ctrl-B ž causes the TTY process to go into a break, opening a break window.

Ctrl-C ž causes Lisp to enter RAID or TELERAID, a debugger where no sane Lisp user wishes to tread.

Ctrl-D ž immediately aborts the current TTY process.

Ctrl-E ž causes a soft abort of the TTY process at the next sensible time. The abort is soft because the aborted process can abort the abort if it has the mechanism to handle the situation.

Ctrl-H ž Pops up a menu listing all currently running processes. Selecting a process from this menu cause that process to go into a break.

Ctrl-T ž prints some statistics about the TTY process in the TTY processes window.

INTERRUPTCHAR is a function that can be used to alter the interrupt status of a given character. **INTERRUPTCHAR** takes two arguments: *CharacterCode* and *InterruptType*.

CharacterCode is the ASCII code for the character whose interrupt status is being altered. The ASCII code can be obtained using **CHARCODE**. For example, Ctrl-C is (**CHARCODE** ^C) is 3.

InterruptType is an atom stating what the new interrupt status should be. The possible values are:

NIL ž turn off this character's interrupt status altogether. The character will no longer be processed by the interrupt mechanism.

RESET ž make this character act like Ctrl-D ordinarily does.

ERROR ž make this character act like Ctrl-E ordinarily does.

HELP ž make this character act like Ctrl-H ordinarily does.

BREAK ž make this character act like Ctrl-B ordinarily does.

RAID ž make this character act like Ctrl-C ordinarily does.

CONTROL-T ž make this character act like Ctrl-T ordinarily does.

Finally, if *InterruptType* is T, no change is made to the character's interrupt status, but its current status is returned as the value of the call to `INTERRUPTCHAR`.

`INTERRUPTCHAR` is normally used to turn off unpleasant interrupt characters like Ctrl-C or to change the location of interrupts assigned to characters that are often hit by mistake. For example, Ctrl-H is BackSpace on many other systems such as the Vax. I often hit Ctrl-H by mistake when I meant BackSpace, especially after an extended session using the Vax. Therefore, I have the Ctrl-H function moved over to Ctrl-N.

For example, my Init file contains the following three calls to `INTERRUPTCHAR`:

```
(INTERRUPTCHAR (CHARCODE ^C) NIL) ž turn off Ctrl-C
```

```
(INTERRUPTCHAR (CHARCODE ^H) NIL) ž turn off Ctrl-H
```

```
(INTERRUPTCHAR (CHARCODE ^N) 'HELP) ž make Ctrl-N be  
old Ctrl-H
```

Documentation: interrupt characters and `INTERRUPTCHAR` are discussed in Sections 9.6, 18.1, and 18.20.6.2 of the IRM. Most of the documentation is pretty technical.

Final Note: Don't fool with interrupt characters. Turn off the odious ones in your Init file and then let them be. Don't turn off Ctrl-D or Ctrl-E as it would then be hard to abort a program running amok.

References

See Documentation notes under the various topics above.

Exercises

Work on old homeworks.

LispCourse #11: Error Processing & DWIM

Completions and Corrections

1. KEYACTION and the "A"

I very stupidly used the "A" key in many of my examples of KEYACTION last time. Don't try out my examples on the "A" key. Use the "B" key. The reason is that if you alter the "A" to produce, say, a "C", then you can never type in "KEYACTION" to change the "A" key back since "KEYACTION" will come out "KEYCCTION". Beware!

2. The variable *IT* in the P.A.

The P.A. maintains a variable called *IT* that is always bound to the value of the last expression. You can use *IT* in the current expression to refer to the value returned by the last expression.

For example:

```
1_ (CAR '((A B C)(D E F))
(A B C)
2_ IT
(A B C)
3_ (CDR IT)
(B C)
4_ (CDR IT)
(C)
```

A more realistic example:

```
5_ (COPYFILE '{DSK}ABC '{DSK}NEW)
{DSK}NEW;1
6_ (TEDIT IT)
{PROCESS}#1,1232
```

3. The % quote in the Lisp READER

The Lisp READER uses special characters to delimit atoms and lists. In particular, the "(" and ")" characters delimit lists and spaces delimit atoms. Ordinarily *(ABC)* would be considered an list containing one atom. *(ABC DEF)* would be a list containing two atoms.

A "%" preceding any special character can be used to suppress the special interpretation of that character. When preceded by a "%", special characters are considered ordinary characters.

For example, `%(ABC%)` would be considered a single atom and not a list because the special meaning of the ")" and "(" characters are suppressed by the preceding "%".

Similarly, `(ABC% DEF)` is a list containing a SINGLE atom because the "%" suppresses the delimiting function of the space between "ABC" and "DEF".

The "%" should be thought of as a QUOTE that works on characters rather than Lisp S-expressions.

Since "%" is itself a special character, it has to be quoted if you want to use it in an atom. For example, `(A %% B)` is a list containing three atoms, the middle of which is the atom %.

An Overview of Error Processing

The diagram in the Appendix illustrates what happens when an error occurs in the Lisp evaluator.

Errors occur when the Evaluator tries to evaluate an expression containing an unbound atom (i.e., an atom with no value), an undefined function (i.e., a form whose CAR is not the name of a function), an illegal argument to a functions, etc.

The major steps in the processing of errors are the following:

DWIM (and CLISP) ž unbound atom and undefined function errors are passed to DWIM, which attempts to automatically "correct" the error. DWIM assumes that the error is either a CLISP expression or a typo.

CLISP is a special syntax for Lisp that is "easier to use" (e.g., CLISP allows infix notation). If the "error" was actually a CLISP expression, then DWIM translates the expression into standard Lisp and returns the translated expression to the Evaluator. (Note: CLISP is not an error, just a special syntax that is implemented using the error mechanism. A very poor piece of systems design!!!!)

If the error appears to be a typo, DWIM "corrects" it and returns the corrected expression to the Lisp evaluator.

If it appears that the error is neither a typo nor a CLISP expression, then DWIM just passes on to the Error Mechanism.

Error Mechanism ž *u.b.a.* and *u.d.f.* errors not corrected by DWIM and all other errors are passed to the Error mechanism.

Each error is assigned an error number that identifies what kind of error it is. There are about 50 such error numbers.

The Error Mechanism, then determines whether to enter a Break. This determination depends on a number of factors including how many function have already been called and how long the computation has been going on.

If the error mechanism decides not to enter a Break, it prints an error message on the TTY window and then aborts the evaluation.

Breaks ž when appropriate, the Error Mechanism passes control to the Break Handler.

The Break Handler enters a "break", usually by opening a Break Window.

The Break Handler prints an error message and then passes control to the Break Exec.

In the Break Exec, the user can evaluate any expression just as in the Lisp Exec. Special commands for inspecting the state of the evaluation causing the error are also available.

To end the Break, the user can either repair the error and then return to the Lisp evaluator OR abort the evaluation in progress and return to the Lisp Exec.

DWIM (Automatic Error Correction)

Introduction

DWIM stands for "Do What I Mean".

DWIM contains two major components: an automatic typo corrector and a special set of Lisp-like syntax called CLISP. CLISP will be talked about in a later section. Only the automatic error corrector will be described here.

The DWIM error correction is a valiant attempt at making a user interface that is robust to simple user errors. It looks at an u.b.a. or a u.d.f. AND at the current context and tries to figure out what the user actually meant.

DWIM embodies an implicit model of the user and the types of errors he is likely to make. DWIM also embodies lots of assumptions about the way the Interlisp system works. Unfortunately, neither DWIM's model of the user nor DWIM's assumptions about the Interlisp environment are very good.

You can't turn DWIM off. Too much of the system has been made dependent on DWIM. So you have to learn to deal with DWIM and its idiosyncracies.

Spelling Correction

Basic idea:

The major function of the DWIM error corrector is spelling correction. When given a u.b.a or u.d.f. error, DWIM compares the incorrect atom (i.e., the unbound atom or the CAR of the u.d.f. form) against a list of atoms that it knows about. If one of the atoms on the list is "close" to the error atom, DWIM replaces the error atom by the "correct" atom.

DWIM uses a number of heuristics to determine the "closeness" of two atoms. Basically, closeness decreases with the number of letters that are different between the two atoms. It increases as a function of the number of letters in the longer atom.

Examples:

1. "CONX" is closer to "COND" than to "CORE"
2. "PRETTYPRNT " is closer to "PRETTYPRINT"
than "EQX" is to EQP"

DWIM ignores single transpositions (e.g., SD for DS) and doubled letters (e.g., SS for S). DWIM considers "CONS" and "CONNS" to be maximally close (i.e., identical) because they differ only by a doubled letter. Also "CONS" and CNOS" are maximally close since they differ by a transposition.

DWIM's spelling corrector works as follows:

It compares the error atom to each atom on its lists of atoms and if it can it chooses one.

The choice rules are:

If one matches with maximum closeness, then it chooses that atom. For example, if the error atom is CONSS and CONS is on the known atoms list.

At the end of the list, if DWIM will choose the **one** atom that had the maximum closeness measure, providing its closeness measure is over some threshold.

If there is no **unique** atom with a maximum closeness measure, DWIM won't choose any atom on the list. This can happen if there are no close atoms are if the maximum closeness is shared by two or more atoms on the list.

Lists of known atoms:

DWIM uses different lists of known atoms for different kinds of errors. Basically, it keeps a list of known function names for undefined function errors and it keeps a list of known variables for unbound atom errors.

These lists are automatically maintained by DWIM and by the Programmer's Assistant.

Some atoms are permanently on these spelling lists. Others are temporary.

The temporary atoms are placed on the list when they are used in an expression in which uses one of DEFINEQ, SETQ, COND, DF,

If a temporary atom gets used to correct an error, then it becomes permanent. Otherwise, it falls off the list after 30 or so new temporary atoms are added to the list.

Basically, the list of known atoms corresponds to the last 30 or so atoms or function names that you referred to.

Note that these rules hold basically for user type-in.

DWIM also operates on errors occurring within a function. However, if an error occurs within a function, DWIM acts slightly differently. In particular, it uses the other atoms within the function definition to determine how to spelling correct an unbound atom.

Other DWIM corrections

DWIM makes a few other corrections besides simple spelling corrections. These corrections include the following:

' followed by a space and an S-expression ž This would result in an u.b.a. error on the '. DWIM will "correct" this expression to eliminate the space after the '. For example: `(CONS 'A ' (A B C))` would be corrected to `(CONS 'A '(A B C))` by DWIM. Similarly, `(LIST 'A ' B)` would be corrected to `(LIST 'A 'B)`.

Misplaced T clause in COND expression ž DWIM will attempt to correct various misplacements of a T clause in a COND expression. It uses various heuristics to do so. For example:

`(COND ((NUMBERP 'A) 'X)((T 'Y)))` would get corrected to

`(COND ((NUMBERP 'A) 'X)(T 'Y))`

`(COND ((NUMBERP 'A) 'X))(T 'Y)` would get corrected to

`(COND ((NUMBERP 'A) 'X)(T 'Y))`

And other fairly esoteric stuff ... ž See the DWIM documentation for all the Bells and whistles.

The DWIM user interface

DWIM is called automatically when an appropriate error occurs in the Lisp evaluator. However, DWIM sometimes interacts with the user.

In particular, DWIM has two modes: *CAUTIOUS* and *TRUSTING*.

In **TRUSTING** mode, DWIM makes most corrections without asking the user's permission. It simply prints a record of the corrections it is making in the TTY window.

Example:

5_ `(COSN 4 (LIST 5))`

```
=CONS
```

```
(4 5)
```

The "=CONS' line is a message from DWIM saying that it did a spelling correction that resulted in an atom being corrected to CONS.

```
6_ (DEFINEQ (JUNK (LAMBDA NIL (ITIMSE 4 3))))
```

```
(JUNK)
```

```
7_ (JUNK)
```

```
ITIMSE {in JUNK} -> ITIMES
```

```
12
```

The "ITIMSE {in JUNK} -> ITIMES" line is a message from DWIM saying that while executing JUNK it changed "ITIMSE" to "ITIMES"

In **CAUTIOUS** mode, DWIM makes asks the user's permission before making most corrections.

DWIM will print out the correction message followed by a "?".

If the users types "Y", then DWIM will make the correction.

If the user types "N", then DWIM will not mnake the correction and return to the Error Mechanism.

If the user types a space or <RETURN>, then DWIM will just wait until the user types a "Y" or "N".

If the user types nothing, DWIM will wait 10 seconds and then assume a default answer. The default anser may be "Y" or "N" depending on the type of correction being done.

Example:

```
6_ (DEFINEQ (JUNK (LAMBDA NIL (ITIMSE 4 3))))
```

```
(JUNK)
```

```
7_ (JUNK)
```

```
ITIMSE {in JUNK} ->
```

At this point DWIM waits for an answer of Y or N or for 10 seconds. In this case, the default answer after 10 seconds would be "Y".

When processing type-in DWIM is always in TRUSTING mode. The assumption is that you can always undo what DWIM has done.

When processing a function, DWIM can be in either CAUTIOUS or TRUSTING mode. The default is TRUSTING.

To change DWIM modes, use the function **DWIM**. **DWIM** takes one argument, *Mode*.

If *Mode* is the atom C, then DWIM is set to CAUTIOUS mode.

If *Mode* is the atom T, then DWIM is set to TRUSTING mode.

If *Mode* is NIL, then DWIM is turned off.

Beware: Lots of stuff will stop working when DWIM is turned off. It is probably a very bad idea to turn DWIM off, since you can never tell what depends on DWIM to work correctly!!!

Some DWIM parameters

DWIMWAIT ž the number of seconds DWIM will wait for a response from the user before assuming the default answer. Initially, 10 seconds.

FIXSPELLDEFAULT ž the default answer thatr DWIM uses after DWIMWAIT seconds is up during a spelling correction in CAUTIOUS mode. Initially, y.

FIXSPELLREL ž the minimum closeness measure needed to accept a known atom as a correction for an error atom. If 100, then only maximum matches will be accepted. Note that maximum matches include atoms that differ by single transpositions and doublings. Initially set to 70.

ADDSPELLFLG ž If NIL, suppresses the addition of items to the known atoms lists. If T, enables the additions. Initially, T.

NOSPELLFLG ž If T suppresses *all* spelling correction in DWIM. If other non-NIL value, suppresses spelling corrections in functions but not in type-in. If NIL, spelling correction is done. Initially, NIL.

RUNONFLG ž If T, DWIM tries to correct run-on typos. E.g., (IPLUS 8A) might be corrected to (IPLUS 8 A). If NIL, no run-on correction is done. Initially, NIL because run-on corrections are not very good.

SPELLINGS1, SPELLINGS2, SPELLINGS3, USERWORDS ž these are the lists of known atoms that DWIM and the P.A. maintain. You probably can't do much except look at these.

#SPELLINGS1, #SPELLINGS2, #SPELLINGS3, #USERWORDS ž these parameters have integer values that determine the length of the known atoms lists. If you want to increase the "history" of known atoms, increase the values of the parameters. Initially, 20.

DWIM Documentation

DWIM is documented in Chapter 15 of the IRM. The chapter appears to be significantly out-of-date!. It is very detailed and is probably hard to understand for the average user.

The introduction to Chapter 15 and Section 15.1 are good overall descriptions.

Sections 15.4 and 15.6 are more detailed descriptions that may be of interest to the non-programmer.

Caution is the bottom line with DWIM

DWIM doesn't work very well as a user interface. It is very non-intuitive and causes most users many problems than it solves. Tread with caution when interacting with DWIM.

Examples: [This is a transcript of a short session with DWIM]

```
6_ (CNOS 4 5)
  UNDEFINED CAR OF FORM
  CNOS
7_ (CONS 4 5)
  (4 . 5)
8_ (CNOS 4 5)
  =CONS
  (4 . 5)
9_ (CONNS 4 5)
  =CONN
  {PHYLUM}<4>
10_ UNDO
  CONN undone.
11_ (COSN 4 5)
  =CONS
  (4 . 5)
12_ (CON 4 5)
```

```
=CONN
{PHYLUM}<4>
13_ (CONDD (T 'X))
=CONN
ILLEGAL ARG
(T (QUOTE X))
14_ (COND (T 'X))
X
15_ (CONDD (T 'X))
=CONN
ILLEGAL ARG
(T (QUOTE X))
16_ (CODN (T 'X))
=COND
X
17_ (CONX 4 5)
=CONN
{PHYLUM}<4>
```

LispCourse #12: CLISP

CLISP

Introduction

CLISP (Conversational LISP) is a package that implements an alternative syntax for Interlisp expressions. CLISP is intended to make Interlisp "easier to read and write, especially for beginners".

An example of CLISP is the expression `A_5`. According to standard Interlisp, this is simply an atom whose name is 3 characters long. However, according to CLISP it is alternative syntax for `(SETQ A 5)`.

Another example might be: `A*5+(6/2)`. According to standard Interlisp, this is simply an atom whose name is 4 characters long followed by a list containing a single atom. However, according to CLISP it is alternative syntax for `((PLUS (TIMES A 5) (QUOTIENT 6 2)))`.

CLISP is part of DWIM. It works as follows:

Expressions that the Lisp interpreter cannot interpret due to u.b.a (unbound atom) or u.d.f. (undefined function) errors are passed to DWIM.

DWIM checks to see if the "error" is in fact a legal CLISP expression. If so, it translates the CLISP to standard Lisp and returns the standard expression to the evaluator.

Otherwise, DWIM tries to automatically correct the error. DWIM knows about CLISP as well as standard Lisp. Therefore, it will try to "correct" expression that it thinks are CLISP expressions with typos. The "automatic correction" of CLISP works just like the "automatic correction" of standard Lisp described last time.

If the expression isn't CLISP and can't be corrected, it is passed to the error handler.

For example, when the user types `"A_5"`, a u.b.a error occurs because `A_5` is an atom with no value. The expression `"A_5"` is passed to DWIM/CLISP which

recognizes it as a CLISP expression. CLISP translates the expression into "(SETQ A 5)" which is then evaluated. *The result is the same as if (SETQ A 5) had been typed in.*

Note this all is a bit wacky!!! From the user's point of view, CLISP expressions are simply Lisp expressions written in an alternative syntax. They are in no way "errors". From the system's point of view a legal CLISP expression is an error that is "corrected" by the CLISP translator in DWIM.

CLISP has four basic components:

1. **CLISP character operators (including infix operators):** CLISP provides an alternative syntax based on a series of special characters. For example, CLISP provides infix arithmetic expressions such as "A+B", where the "+" is a CLISP character operator. CLISP uses the "+" as a clue to translate this expression into (*PLUS A B*).
2. **IF-THEN-ELSE statements:** CLISP provides a "more intuitive" syntax for the COND statement that is similar to the IF-THEN-ELSE statements in PASCAL and FORTRAN. CLISP just translates these statements into their equivalent COND forms.
3. **Iterative statements:** The FOR and WHILE loops discussed in Session #5 are implemented in CLISP. CLISP translates these iterative statements into a series of standard Lisp statements that carry out the iteration.
4. **Record Package:** The Record Package is a package that implements various data structures in addition to lists. We will cover the Record Package in detail in the programming section of this course.

We will not discuss the Record Package until much later in the course.

We have already covered the Iterative statements (i.e., FOR and WHILE statements) in detail. The fact that they are CLISP rather than standard Lisp has little effect on the user. In most interactions with the system, the user will see only FOR or WHILE statements and not their translations into standard Lisp. Exceptions to this rule do exist, however. So you should remember that FOR and WHILE loops are in fact CLISP and not standard Lisp expressions.

The CLISP character operators and the IF-THEN-ELSE statements are discussed in detail below. The fact that these are CLISP rather than standard Lisp is very evident to the user.

During the translation from CLISP to Lisp, the Lisp expression actually replaces the CLISP statement. All future accesses to the expression return the translation (i.e., Lisp) rather than the original (i.e., CLISP). For example, the typed-in expression `A_5` is replaced by the Lisp expression `(SETQ A 5)`. `A_5` is forgotten. In particular, the entry on the history list is `(SETQ A 5)` and NOT `A_5`. `"FIX SETQ"` will work but `"FIX A_5"` will not.

CLISP Character Operators

CLISP interprets several characters as operators to be applied to the surrounding S-expressions. Most of the CLISP operators are *infix* operators such as the "+" in `A+B`. There are some *prefix* operators such as the "~" in the expression `~(NUMBERP X)`.

In a CLISP expression, an operator can be surrounded by spaces or it can be in the middle of an atom. To CLISP, `"A+B"` is equivalent to `"A + B"`.

CLISP uses these operators as the basis for translating CLISP expressions into the appropriate Lisp function calls.

Arithmetic Operators:

`+, -, *, /, ^` ǰ these are the CLISP infix arithmetic operators. They are translated into the appropriate calls to PLUS, DIFFERENCE, TIMES, QUOTIENT, and EXPT.

Examples:

`4/2` translates to `(QUOTIENT 4 2)`

`(4+5)*3` translates to `(TIMES (PLUS 4 5) 3)`

`(A+(SETQ A 5))*(SETQ A 7)` translates to
`(TIMES (PLUS A (SETQ A 5)) (SETQ A 7))`

Parentheses can be used to insure that the operators are interpreted in the correct order. For example: $5*4+6$ can be either $(5*4)+6$ versus $5*(4+6)$.

In the absence of parentheses, normal rules of precedence are used. Thus, \wedge is higher than $*$ and $/$ which in turn are higher than \ddot{y} and $+$. Within a precedence level, precedence is given to the leftmost operator.

Examples:

$4*2+5$ translates to $(PLUS (TIMES 4 2) 5)$
[rather than to $(TIMES 4 (PLUS 2 5))$]

$4+2-3$ translates to $(DIFFERENCE (PLUS 4 2) 3)$
[rather than to $(PLUS 4 (DIFFERENCE 2 3))$]

$4*x^2$ translates to $(TIMES 4 (EXPT X 2))$
[rather than to $(EXPT (TIMES 4 X) 2)$]

\ddot{y} (i.e., **unary minus**) \check{z} the minus sign can also be used as a prefix unary minus to indicate arithmetic negation. Since the minus sign is both a binary (infix) and a unary (prefix) operator the following rule is in effect: the minus sign is interpreted as a binary minus except when it is the first item in a list or it follows another operator. Examples: $\ddot{y}A$, $(\ddot{y}A)$, $(B*\ddot{y}A)$.

Logical Operators:

$=$, **GT**, **LT**, **GE**, **LE** \check{z} are infix operators for EQUAL, GREATERP, LESSP, "Greater than or equal to", and "Less than or equal to". These are translated to a call to the appropriate predicate or predicate composition.

Examples:

$A GT B$ translates to $(GREATERP A B)$

$A=B$ translates to $(EQUAL A B)$

$A LE B$ translates to $(OR (EQUAL A B) (LESSP A B))$

Note that except for the $=$, all of these operators must be surrounded by spaces. For example, $A GT B$ cannot be written $AGTB$. However, $A = B$ can be written $A=B$.

AND, OR, MEMBER, EQUAL ž are infix operators standing for the Lisp functions of the same name. They are translated accordingly.

Examples:

$A \text{ EQUAL } B$ translates to $(\text{EQUAL } A \ B)$

$A \text{ OR } B$ translates to $(\text{OR } A \ B)$

$A \text{ MEMBER } B$ translates to $(\text{MEMBER } A \ B)$

\sim (**unary**) ž is prefix operator meaning NOT (or NULL).

Examples:

$\sim(\text{MEMBER } A \ '(1 \ 2 \ 3))$ translates to

$(\text{NULL } (\text{MEMBER } A \ '(1 \ 2 \ 3)))$

$\sim(A \ \text{OR } B)$ translates to $(\text{NULL } (\text{OR } A \ B))$

Note: Parentheses can be used to insure proper interpretation of logical combinations: $A \ \text{OR } B \ \text{AND } C \ \text{EQUAL } D$ can be written as $(A \ \text{OR } B) \ \text{AND } (C \ \text{EQUAL } D)$ or as $(A \ \text{OR } (B \ \text{AND } (C \ \text{EQUAL } D)))$

In the absence of parentheses, the precedence rules for the logical operators are as follows:

=

LT, GT, LE, GE, EQUAL, MEMBER

AND

OR.

Otherwise, precedence goes from left to right.

All of the logical operators have a lower precedence than the arithmetic operators.

Examples:

$A=B \ \text{OR } C=D$ translates to $(\text{OR } (\text{EQUAL } A \ B)(\text{EQUAL } C \ D))$

$A \ \text{EQUAL } B + C$ translates to $(\text{EQUAL } A \ (\text{PLUS } B \ C))$

$A \ \text{AND } B \ \text{OR } C$ translates to $(\text{OR } (\text{AND } A \ B) \ C)$

Other Operators:

: \checkmark is an infix operator that extracts elements from a list. $X:N$ stands for the Nth element of the list X. For example, $X:2$ stands for $(CADR X)$ while $X:4$ stands for $(CAR (CDDDR X))$.

The **:** operator has a higher precedence than any of the arithmetic or logical operators.

A negative N indicates the Nth element from the end of the list. For example $X:-1$ stands for the last element in the list.

The **:** operator can be composed as in $X:2:3$ which stands for *the third element of the second element of X* or $(CADDR (CADR X))$.

:: \checkmark is an infix operator that extracts tails of lists. For example: $X::1$ is the $(CDR X)$, $X::3$ is the $(CDDDR X)$, and $X::-1$ is the $(LAST X)$.

_ \checkmark is an infix operator that indicates assignment. For our purposes, assignment means SETQ. For example, X_Y translates to $(SETQ X Y)$.

[Note: The **_** operator can also be used in conjunction with the **:** operator to alter the composition of a list. For example, $X:2_5$ means replace the second element of X with 5. However, we have not yet covered how to say this in straight Lisp!].

<, > \checkmark are special operators in CLISP used to construct lists. A balanced pair of angle brackets indicates that a list is to be constructed containing everything between the "<" and the ">". For example, $\langle A B C \rangle$ translates to $(LIST A B C)$, while $\langle 1 2 3 \langle 4 5 6 \rangle \rangle$ translates to $(LIST 1 2 3 (LIST 4 5 6))$.

IF-THEN-ELSE expressions

CLISP provides an IF-THEN-ELSE statement similar to that found in PASCAL and FORTRAN.

The form of the IF-THEN-ELSE expression is:

(IF A THEN B ELSEIF C THEN D ELSEIF E THEN F ... ELSE G)

This expression is directly translated into:

(COND

(A B)
 (C D)
 (E F)
 ...
 (T G))

In "English":

If A is non-NIL, then evaluate B and exit
 Otherwise, if C is non-NIL, then evaluate D and exit
 Otherwise, if E is non-NIL, then evaluate F and exit
 ...
 Otherwise, evaluate G and exit.

The IF, THEN, ELSEIF and ELSE keywords have the lowest precedence of any of the CLISP character operators.

Example:

(IF A = B + C THEN C + D ELSE E - F)

translates to

*(COND
 ((EQUAL A (PLUS B C)) (PLUS C D))
 (T (DIFFERENCE E F)))*

Using CLISP - Advantage and Disadvantages

The advantage of CLISP is that it allows you to type-in complex expressions using a syntax that is more concise and supposedly more intuitive than the standard Lisp syntax.

The following examples make the case:

$A^2+B^2+C^2$

instead of *(PLUS (EXPT A 2)(EXPT B 2)(EXPT C 2))*

$(IF A^2+B^2+C^2=C+D-E*2 THEN A_C+E ELSE C_X:2)$

instead of

(COND

```
((EQ
      (IPLUS (EXPT A 2) (EXPT B 2) (EXPT C 2))
      (IDIFFERENCE (IPLUS C D) (ITIMES E 2)))
      (SETQ A (IPLUS C E)))
      (T (SETQ C (CADR X))))
```

The **disadvantages** of CLISP are many! First CLISP is another whole set of rules for the user to learn. The rules are inconsistent with the rules of Lisp. For example, CLISP violates the notion that every Lisp expression starts with the name of a function to be applied to the arguments which follow. Thus, CLISP has opted for a syntax that is locally optimized (or assumed to be optimized) for certain special cases at the expense of consistency across the system as a whole. It is not at all clear that this was a good trade-off to make!!

CLISP is not well integrated into the Interlisp environment.

When CLISP expressions are translated into Lisp, the Lisp expressions replace the CLISP expressions. Thus if you look at an entry on the history list or if you DEdit a function, the CLISP you typed-in will be gone and a very different expression will be in its place. This can be very, very confusing!

Example:

```
65_ (COND (T NIL))
NIL
66_X_(IF ~(LISTP X) THEN A_C+D ELSE X:3)
3
67_REDO IF
IF ?
68_REDO COND
8
...
72_(DEFINEQ (TEST (LAMBDA (X) (IF ~(LISTP X) THEN
A_C+D ELSE X:3) )))
(TEST)
73_PP TEST
```

```

(TEST
  (LAMBDA (X) **COMMENT**
    (IF ~(LISTP X)
      THEN A_C+D
      ELSE X:3)))
(TEST)
74_(TEST 1)
8
75_PP TEST
(TEST
  (LAMBDA (X) **COMMENT**
    (COND
      ((NLISTP X)
        (SETQ A (IPLUS C D)))
      (T (CADDR X))))))
(TEST)

```

A second example of problems with CLISP arises when DWIM error "correction" interacts with CLISP. The following is a simple example of a DWIM/CLISP interaction that is not at all intuitive:

```

86_ (SETQ PATIENT-RECORD 99)
99
87_ (SETQ PATIENT 7)
7
88_ (SETQ RECORD 2)
2
89_ PATENT-RECORD
=PATIENT
5

```


[Why should DWIM correct this to (DIFFERENCE PATIENT RECORD) rather than to the atom PATIENT-RECORD ????]

Moral: CLISP is nice, but watch out for all its traps. CLISP can be very handy when typing complex expressions. But its utility is limited by the fact that it is just a coating of syntactic sugar spread on top of Interlisp that is both inconsistent with the rest of Interlisp language AND not well integrated into the eInterlisp environment.

DWIMIFY and CLSPIFY

DWIMIFY is a function that applies DWIM to an expression returning the expression with all of the "errors" corrected. This forces the translation of CLISP expressions as well as other DWIM corrections.

DWIMIFY takes one argument which is either the name of a function or an expression. DWIMIFY runs DWIM on the function or expression, making all the necessary changes and returns the translated/corrected expression.

Example:

```
99_(DWIMIFY '(IF A=B THEN C+D ELSE E+F))
E+F TREAT AS CLISP ? yyes
(COND
  ((EQ A B)
   (IPLUS C D))
  (T (IPLUS E F)))
(COND ((EQ A B) (IPLUS C D)) (T (IPLUS E F)))
```

CLSPIFY is a function that takes a standard Lisp expression and translates it into CLISP using as many CLISP operators as possible. CLSPIFY is a sort of inverse to DWIMIFY.

CLSPIFY takes one argument which is either the name of a function or an expression. CLSPIFY translates the function or expression to CLISP

and returns the translation. If X is a function name, the function is redefined to be the CLISPIFYed translation.

Example:

```
4_(CLISPIFY
      (QUOTE
        (COND ((LISTP X) (SETQ A 5))
              (T (SETQ B 6))))))
(if (LISTP X) then A_5 else B_6)
```

DWIMIFY and CLISPIFY are available in DEdit as subcommands of the EDITCOM command. Select the expression you want DWIMIFYed or CLISPIFYed then click on EDITCOM in the DEdit menu with the mouse button. Choose DW (DWIMIFY) or CL (CLISPIFY) from the submenu that appears. The selected expression will be replaced by its DWIMIFYed or CLISPIFYed equivalent.

Tailoring CLISP

There are several parameters and functions that alter the behavior of CLISP. Some of the more interesting ones are the following:

CLISPFLG ž If NIL all CLISP infix and prefix operators are disabled but IF-THEN-ELSE and Iterative expressions remain in force. If the value is TYPE-IN, then CLISP is in effect only on user type-in and not on the body of defined functions. If the value is T, CLISP is in effect on all evaluated expressions. Initial value is T.

(CLDISABLE X) ž disables the CLISP operator X. For example, (CLDISABLE '-') disables the - CLISP character operator while (CLDISABLE 'IF) disables IF expressions.

CLSPIFTRANFLG ž If T, the original IF statements are left alone during CLISP translation to Lisp. If NIL, the original CLISP is replaced by the Lisp translations. Initially, NIL.

CLISPIFYPRETTYFLG ž If ALL, causes PP and MAKEFILE to CLISPIFY functions before printing them. If a list, causes all functions on the list to be CLISPIFYed before printing. If NIL, nothing is CLISPIFYed before printing, i.e., functions are printed as is.

CLISP Documentation

CLISP is documented in Chapter 16 of the IRM. Sections 16.1 thru 16.8 contain material of general interest. Section 16.9 is for hackers only. Description of tailoring parameters and functions appears mostly in Sections 16.6 thru 16.8.

IF-THEN-ELSE and Iterative expressions are covered in Chapter 4 of the IRM.

The Record Package is covered in Chapter 3.

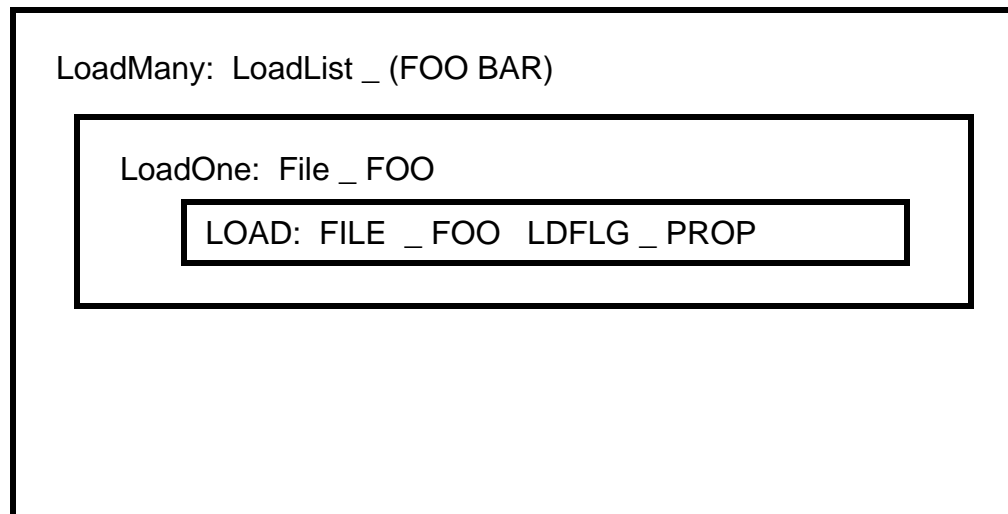
LispCourse #13: Error Handling and the Break Package

Background Concept: The Stack

Consider the following set of functions:

```
(DEFINEQ
  (LoadMany (LAMBDA (LoadList)
    (COND
      (LoadList
        (LoadOne (CAR LoadList))
        (LoadMany (CDR LoadList))))))
  (LoadOne (LAMBDA (File)
    (LOAD File 'PROP))))
```

Trace the following function call down to the first call to LOAD: (LoadMany '(FOO BAR))



When the Lisp evaluator starts to process the call to LOAD, it is keeping track of three function calls: the call to LoadMany, the call to LoadOne, and the call to LOAD.

The Lisp evaluator needs to keep track of lots of information about these three function calls. For example for each function call the evaluator must record the exact function called, the values of the arguments in the function call, where in the function body it currently is, and so on.

Lisp keeps this kind of information in a data structure called **the stack**.

The stack consists a set of stack frames. Each stack frame contains the information about one function call. The stack frames are ordered from most recent at the top of the stack to oldest at the bottom of the stack.

When a new function call is made, a new frame is added to the top of the stack. When a function call completes, its frame is removed from the top of the stack. Thus the stack grows and shrinks as function calls are made and then completed.

For example, at the start of the LOAD function call the Lisp stack would look something like:

```
LOAD:
  FILE _ FOO
  LDFLG _ PROP
LoadOne:
  File _ FOO
LoadMany:
  LoadList _ (FOO BAR)
```

After FOO is loaded and LoadOne completes, the Lisp stack would be:

```
LoadMany:
  LoadList _ (FOO BAR)
```

LoadMany then recurses to load BAR, so at the point that LOAD is called again the stack would be:

```
LOAD:
  FILE _ BAR
  LDFLG _ PROP
LoadOne:
  File _ BAR
LoadMany:
  LoadList _ (BAR)
LoadMany:
  LoadList _ (FOO BAR)
```

Note that at each point, the stack contains an ordered list of the function calls that are in progress. Each function call on the stack is waiting for function call just above it to complete and return a value. The top of the stack contains information about the function call currently being processed.

In short, at any given time the stack contains a record of the current state of a computation in progress.

When an error occurs, the stack contains valuable information about the state the computation was in when the error occurred.

Errors: Break or Abort?

When the Lisp evaluator encounters an error that DWIM/CLISP cannot "correct", it does one of two things:

- 1) it **aborts** the computation in progress. Unless the aborted program specifies otherwise, control returns to the Lisp exec. (Programs can be written so that they "handle" the abort by "fixing" or ignoring the error.)
- 2) it suspends the computation in progress and enters a **break**. From the break, the user can "fix" the error and resume (or abort) the computation.

The decision whether to Abort or Break is based on the heuristic that complex computations should be broken while simple computations should be aborted. This heuristic rests on the assumption that simple computations are easier redone rather than fixed and resumed.

"Complex" in this case is based on two factors, "depth" of the computation and time spent so far in the computation.

Depth is basically the number of number of function calls on the stack when the error occurs. If this number passes a given threshold (i.e., the value of `HELPDEPTH`), the computation will be broken.

Time is measured by how long the computation has been in progress. If this time passes a given threshold (i.e., the value of `HELPTIME`), then the computation will be broken.

If neither the depth nor the time passes threshold, then the computation will be aborted.

Some programs choose to handle their own aborts using the error processing machinery provided by Interlisp. The computation of "complexity" for these programs is slightly different. The effect is that breaks are less likely to occur and aborts are more likely to occur. When the abort does occur it is up to the program to decide what to do.

When an error causes an abort, the program can try to "fix" the error or decide to ignore it. If it does either of these, the computation will proceed as if no error had occurred (though the program will probably print out some sort of error message to let you know that the error did occur.) The program can also decide to force a break or it can force a "true" abort and return to the Lisp exec.

Standard packages like TEdit and DEdit almost always process their own aborts. They try to ignore or fix all but the most drastic errors so that the user isn't constantly bombarded by breaks and/or aborts.

When a computation is aborted, an error message is always printed in its TTY window before control is given to the Lisp exec. This message usually includes the type of error and the offending object. For example: "Non-numeric arg: NIL" indicates that one of the arguments to some function was NIL when it was supposed to be some numeric value.

There are some 50 standard types of error in Interlisp. We will review some of these later.

Examples:

12_(PLUS T)

NON-NUMERIC ARG
T

13_(QQQQQ 4)

UNDEFINED CAR OF FORM
QQQQQ

14_(SETQ NIL 5)

ATTEMPT TO SET NIL OR T
5

Breaks: the Break Exec and Break Windows

When a Break occurs, the computation is suspended and a *Break window* is opened on the screen. The title of this window is a message describing the error that caused the break. This error message is also printed in the window.

A *Break exec* is started in the Break window. A Break exec is very similar to the Lisp exec except it has an additional set of commands that can be used to "handle" the break.

The Break package is primarily intended as a debugging tool for programmers. Rather than aborting a computation when an error occurs, Interlisp suspends the computation and enters a break. The break gives the user an opportunity to examine the suspended computation, make changes that "fix" the error, and then restart the computation from any point. From a programmer's point of view this is a fantastic opportunity.

Unfortunately, to make full use of breaks you need to have considerable expertise in Interlisp programming. The average user doesn't have sufficient expertise and therefore can't really take advantage of the power of breaks.

But, when an error occurs a break usually occurs. So the all users have to learn to deal with breaks as best they can. We will cover "dealing with breaks" today and pick up the "effective use of breaks" later in the programming part of the course.

What is a break and what is it good for?

When a break occurs the ongoing computation is suspended at the point at which the error occurred. The break provides the tools for examining the state of the computation including its stack, all of the variables its using, the definitions of the functions it calls, etc. Functions can be edited, the values of variables reset, previous events undone, and so on. The computation can then be restarted at any point or it can be aborted completely.

Basically, a break is an opportunity for the user to do what she can to recover from errors, thereby saving any work that has been done by the computaion before the error occurred. The decision to abort the computation if the work can't be saved or isn't worth saving is left in the hands of the user rather than the system.

Examples:

If during a COPYFILE you run out of space on your account on the file server, the system will enter a break with a message saying "File system resources exceeded". From the break, you can delete some files on your account to free up some space. You can then type "OK" to restart the COPYFILE. If you can't find any files to delete and just want to stop the COPYFILE, you can type "^" to abort the COPYFILE and return to the Lisp exec.

If you type "(LOAD 'FOOBAR)" to the Lisp exec and FOOBAR doesn't exist, the system will (sometimes) enter a break with saying "File not found FOOBAR". If you really meant to load FUZBAL, you can type "(SETQ FILE 'FUZBAL)" followed by "OK". This will "correct" the typo you made and restart the computation, causing the file FUZBAL to be loaded.

The Break Exec

The Break exec running inside the Break window is a read-eval-print loop just like the Lisp exec. The type-in prompt is a ":" rather than a "_" to distinguish the Break exec from the Lisp exec.

The history event numbers that precede the prompt are shared between the Lisp exec and the Break exec, i.e., there is only one history list that serves all execs, Lisp and Break alike.

The Break exec contains all of the functionality of the Lisp exec including TTYIN and the Programmer's Assistant. You can evaluate any Lisp expression, CLISP expression, P.A. command, etc. just as you can in the Lisp exec.

The Break exec contains a number of additional commands for managing the Break, for repairing the computation, and for restarting/aborting the computation. These commands are like the P.A. commands in that you simply type them into exec followed by a <RETURN>. No parentheses etc. are necessary.

BRKEXP: When you enter a break, the value of the variable BRKEXP is the S-expression that caused the error in the evaluator. If you can't figure out what to do from the error message printed in the break window, you can always examine the value of BRKEXP.

BRKEXP can also be used as the expression to evaluate when leaving the break. Several of the break exec commands involve modifying or re-evaluating the BRKEXP expression.

Example:

In the LOAD example above, the value of BRKEXP was *(OPENFILE FILE ACCESS RECOG BYTESIZE)*. This suggests that the variable *FILE* is the "cause" of the error and should be set to the correct file name. Hence to fix the error, you SETQ *FILE* to the correct name. You then use the OK break command to exit the break and reevaluate the BRKEXP expression.

```
41_(LOAD 'FOOBAR)
FILE NOT FOUND
FOOBAR
(OPENSTREAM broken)
42:BRKEXP
(OPENFILE FILE ACCESS RECOG BYTESIZE)
43:FILE
FOOBAR
44:(SETQ FILE 'FUZBAL]
FUZBAL
45:FILE
FUZBAL
46:OK
{DSK}FUZBAL
47_
```

Break Exec Commands: The following are special commands that can be used to manage breaks from the break exec.

Exiting from a break:

When exiting from a break, you return some value. This is used in by the evaluator as the value of the expression that caused the error. For

example, the expression (*SETQ A (PLUS I T)*) will cause a "non-numeric arg" break while evaluating the (*PLUS I T*) expression. The value returned from this break is used in place of the value of the (*PLUS I T*) expression and hence is the value that A will be set to.

The following functions exit from a break, returning some value.

GO ž Evaluates the BRKEXP expression, prints the result in the break window, and exits the break returning the evaluation result.

OK ž Like GO, except doesn't print the result of the evaluation.

RETURN *Form* ž Evaluates *Form* and then exits from the break returning the evaluation result.

The following function, exit from a break by aborting and hence doesn't return any value:

^ ž exits the break by causing an abort. This will cause a return to the Lisp exec unless aborts are explicitly processed by the broken program as described earlier.

Other Commands:

Many of the break exec commands rely on the value of the variable LASTPOS. LASTPOS points to an entry on the stack, i.e., to a function call somewhere on the stack. Many functions allow you to see various characteristics of the LASTPOS function call. You can move the value of LASTPOS in order to examine various function calls on the stack. LASTPOS starts out at the expression causing the break.

EVAL ž evaluates the BRKEXP expression but does not exit from the break. !VALUE is set to the result of the evaluation. Can be used to see what the break would return, if GO or OK were used.

@ Atom ž moves LASTPOS to the most recent call to the function named by *Atom*. To search for the second most recent call to the function named by atom use "*@ Atom Atom*" and so on.

?= ž prints out the values of the arguments to the function at LASTPOS. For example, in the LOAD break described above with LASTPOS pointing to the OPENFILE function call, **?=** would print out the values of FILE, ACCESS, RECOG and BYTESIZE.

BT DUMMYFRAMEPž prints out the names of the function calls on the stack starting from the most recent to the oldest (called a *backtrace*). The DUMMYFRAMEP is optional, but gets a more concise backtrace with less junk in it.

REVERT ž removes from the stack all function calls from the error back through the function call specified by LASTPOS. It then causes a break in the function call specified by LASTPOS. Thus REVERT allows you to back the computation up to some previous point before the error was encountered.

EDIT ž calls DEdit on the function containing the BRKEXP expression. For example: if in the definition of the function FOO there is an expression like *(PLUS 1 T)*, a break will occur with the value of BRKEXP being *(PLUS 1 T)*. In this break, EDIT will call DEdit on the function FOO because FOO is the function that contains the incorrect statement *(PLUS 1 T)*.

Examples of Break Exec in use

```
99_(COPYFILE (QUOTE {DSK2}ABC)
             (QUOTE {DSK2}XYZ))
```

```
FILE NOT FOUND
{DSK2}ABC
```

```
(OPENSTREAM broken)
100:BRKEXP
(OPENSTREAM FILE ACCESS RECOG BYTESIZE PARAMETERS)
```

```

1:BT DUMMYFRAMEP
  OPENSTREAM
  ERRORSET
  COPYFILE
  EVAL
  LISPX
  ERRORSET
  EVALQT
  ERRORSET
  T
2:?=
  *FILE* = {DSK2}ABC
  *ACCESS* = INPUT
  *RECOG* = NIL
  *BYTESIZE* = NIL
  *PARAMETERS* = ((SEQUENTIAL T))
3:(COPYFILE '{DSK2}AAA '{DSK2}ABC]
  {DSK2}ABC.;1
4:OK
  OPENSTREAM
  {DSK2}XYZ.;1
5_

```

```

-----
64_(DEFINEQ
  (EXAMPLE (LAMBDA (A B C)
    (PLUS
      (EXAMPLEX A)
      (EXAMPLEX B)
      (EXAMPLEX C))))))
(EXAMPLE)
65_(DEFINEQ
  (EXAMPLEX (LAMBDA (X)
    (EXAMPLEY X 1))))

```

```
(EXAMPLEX)
66_(DEFINEQ
      (EXAMPLEY (LAMBDA (P Q)
                  (PLUS P Q))))
```

```
(EXAMPLEY)
67_(EXAMPLE 2 3 4)
12
...
77_(EXAMPLE 1 2 T)
```

NON-NUMERIC ARG

T

(PLUS broken)

80:BT DUMMYFRAMEP

PLUS

EXAMPLEY

EXAMPLEX

EXAMPLE

EVAL

LISPX

ERRORSET

EVALQT

ERRORSET

T

81:?=

**ARG1* = T*

**ARG2* = 1*

82:@ EXAMPLEY

EXAMPLEY

83:?=

$P = T$

$Q = 1$

84:@ EXAMPLEX

EXAMPLEX

85:?=

$X = T$

86:REVERT

EXAMPLEX

(EXAMPLEX broken)

87: BT DUMMYFRAMEP

EXAMPLEX

EXAMPLE

EVAL

LISPX

ERRORSET

EVALQT

ERRORSET

T

88:?=

$X = T$

89:X

T

90:(SETQ X 55)

55

91:X

55

92:BRKEXP

(PROGN (fgh: "13-Mar-85 22:54") (EXAMPLEY X 1))*

93:EVAL

EXAMPLEX evaluated


```
94:!VALUE
```

```
56
```

```
95:OK
```

```
EXAMPLEX
```

```
61
```

```
96_
```

```
-----
```

Break Windows

When a break occurs, the system opens a break window and starts up the break exec within this window. All interaction with the break exec takes place in this window, just as all interaction with the Lisp exec takes place in the TTY window.

Break windows provide two other functions besides exec type-in:

1. menu-based access to many of the break exec commands
2. visual displays of the stack and stack frames

Menu-based access to the break exec:

Clicking on the middle mouse button anywhere in the break window will bring up a menu consisting of 9 items: *!EVAL*, *EVAL*, *EDIT*, *revert*, *^*, *OK*, *BT*, *BT!*, *?=*.



Choosing *EVAL*, *EDIT*, *revert*, *^*, *OK*, or *?=* from the menu simply carries out the corresponding break exec command. For example, choosing the *OK* entry will exit the break after evaluating the BRKEXP expression.

Choosing the *BT* command from this menu will bring up a small window (called the *backtrace window*) attached to the left or right edge of the break window. The stack backtrace will be printed in this window, one item (i.e., function call) per line. The window is scrollable if the whole backtrace doesn't fit in the window.

The screenshot shows a window titled "{DSK2}ABC - FILE NOT FOUND break: 1". The main area contains the following text:

```
FILE NOT FOUND
{DSK2}ABC
(OPENSTREAM broken)
20:
```

On the right side, there is a menu with the following items:

```
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPM
ERRORSET
EVALQT
ERRORSET
_
```

The *!EVAL* and *BT!* are simply specialized versions of the *EVAL* and *BT* commands.

The backtrace window and the frame inspector:

Each item (i.e., function call) in the backtrace window is individually selectable by clicking on the left or middle mouse buttons while the cursor is over the item. The selected item will be highlighted with a gray background.

Selecting an item in the backtrace window has two effects:

1. LASTPOS is set to the corresponding stack entry. This is the menu-based equivalent of the @ command in the break exec.
2. a stack frame inspector is opened on the corresponding stack entry. This stack frame inspector contains the name of the function call and the functions arguments and their values.

```

COPYFILE Frame
COPYFILE
FROMFILE      {DSK2}ABC
TOFILE        {DSK2}XYZ
LISPXHIST     ((&) ← *LISPXPRINT* ("
               " "FILE NOT FOUND" "
               " {DSK2}ABC --))
RESETY        NIL

{DSK2}ABC - FILE NOT FOUND  break: 1
FILE NOT FOUND
{DSK2}ABC

(OPENSTREAM broken)
20:
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
_

```

The stack frame inspector is a standard inspector window. It has lots of functionality which will not be covered here. However, there are several operations in the inspector that may be useful. In particular:

To call DEdit the function in the frame:

First select the function name at the top of the inspector window by placing the cursor over the name and clicking the left mouse button.

Then click the middle mouse button anywhere in the inspector window. This will bring up a menu with three choices. Choose the middle one: *DisplayEdit*. This will bring up an editor on the function in the inspector window.

To change the value of one of the arguments:

First select the argument to be reset by placing the cursor over the argument name and clicking the left mouse button.

Then click the middle mouse button anywhere in the inspector window. This will bring up a menu with one item. Choose this item: *SET*.

This will bring up type-in window just above the inspector window. Type the new value for the argument into this window. NOTE that the expression you type in will be evaluated before the argument is set. Hence to set the argument to A, you have to type in (QUOTE A) and so on.

```

Enter the new (FROMFILE 1) for #1,13434/COPYFILE
The expression read will be EVALuated.
>
COPYFILE Frame
COPYFILE
FROMFILE {DSK2}ABC
TOFILE {DSK2}XYZ
LISPXHIST ((&) ← *LISPXPRINT* ("
" "FILE NOT FOUND" "
" {DSK2}ABC --))
RESETY NIL

{DSK2}ABC - FILE NOT FOUND break: 1
FILE NOT FOUND
{DSK2}ABC
(OPENSTREAM broken)
26:
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
_

```

The new setting for the argument will be displayed immediately in the inspector window.

```

COPYFILE Frame
COPYFILE
FROMFILE      {DSK2}QRZ
TOFILE        {DSK2}XYZ
LISPXHIST     ((&) + *LISPXPRINT* ("
              " "FILE NOT FOUND" "
              " {DSK2}ABC --))
RESEY        NIL

{DSK2}ABC - FILE NOT FOUND  break: 1
FILE NOT FOUND
{DSK2}ABC
(OPENSTREAM broken)
26:
ERRORSET
BREAK1
EVALA
OPENSTREAM
ERRORSET
COPYFILE
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET

```

Breaks within Breaks

When working inside a break frequently evaluates an expression containing an error. If this error causes a break, a second break window will be opened and a break exec started in this window.

Exiting from this second break window using either a normal exit (OK, GO, RETURN) or an abort (^) will simply return to the next higher level break exec, i.e., the break in which the the break being exited occurred.

Breaks within breaks are like DEdits. You can exit from a higher level break only by exiting from the next lower level break first.

If an error occurs in the break exec that would normally cause an abort rather than another break, then the abort error message is simply printed in the break window and control returns to the break exec.

Tailoring the Break Package

The following parameters can be used to tailor various aspects of the error handler, the break exec and break windows.

HELPDEPTH ž an integer used as a threshold for determining whether to break or abort. When an error occurs, if the stack depth is greater than HELPDEPTH then the computation is broken, otherwise its aborted. Initially, 7. A setting of 1 will insure that a break always occurs.

HELPTIME ž an integer indicating the number of milliseconds that a computation has to be in progress for a break to be used instead of an abort when an error occurs. Initially, 1000.

HELPFLAG ž if NIL, no breaks will occur and all errors will cause an abort. If BREAK!, a break will occur for every error. If T, the break or abort decision will be made based on HELPDEPTH and HELPTIME as described above. Initially, T.

Note HELPFLAG must be set with SETTOPVAL as in (SETTOPVAL 'HELPFLAG 'BREAK!). SETQ cannot be used to set this variable.

AUTOBACKTRACEFLG ž if non-NIL, a backtrace window is automatically opened along with every break window. If NIL, then the BT menu command must be used to open the backtrace window. Initially, NIL.

CLOSEBREAKWINDOWFLG ž if NIL, then a break window will remain on the screen when the break is exited. The window must then be closed by hand. If T, then each break window is closed as the break is exited. Initially, T.

Break Documentation

Breaks and the Break Exec are documented in Sections 9.1 thru 9.3 of the IRM. Break Windows are documented in Section 20.3 of the IRM. All of these sections contain a mix of user and programmer material.

The Standard Interlisp Errors

There are currently fifty-plus standard types of errors in the Interlisp system. When an abort or a break occurs, most errors will print the offending expression following the error message for that type of error, e.g., NON-NUMERIC ARG NIL is very common.

All of the standard Interlisp errors and error messages are described (very) briefly in Section 9.8 of the IRM. This section is reproduced in the Appendix.

Error #17 is a sort of generic error. Many packages and system function use the function called ERROR! to indicate an error not on the standard error types list. ERROR! will indicate an generic error and then print a more specific error message given to it by the specific package or function in which the error occurred.

Exercises

Set HELPDEPTH to 1. Then start entering Lisp expression containing errors. When these expressions cause errors, browse around in the break trying to do various things.

Define a few functions with errors and then execute them. Browse around in the resulting break. Try to correct the error and restart the computation.

LispCourse #14: Multiple Processes and the Process Status Window

Processes and Multiple Processes

Consider how you start up a DEdit:

You type "(DF FOO)" in the Lisp Exec window. This opens a DEdit window separate from the Exec window. You can then carry out your editing actions in this DEdit window.

But note: as long as the DEdit window is active, the Exec window is inactive. If you try to type into the Exec window, the input goes into the DEdit input buffer.

Only after you exit the DEdit, does a value (i.e., the name of the edited object) get returned in the Exec window.

After the value is returned, the Exec window becomes active again and the DEdit window (if its still on the screen) becomes inactive.

Conclusion: the Exec and DEdit use seprate windows but can not run simultaneously. They must "take turns".

Contrast this behavior with that of TEdit:

You type "(TEDIT 'FOO)" in the Lisp Exec window. This opens a TEdit window separate from the Exec window. You can then carry out your editing actions in this TEdit window.

But note: TEdit reads in the file FOO (which make take some time if the file server is slow) and then immediately returns a value (usually a "nonsense" value!).

The Exec window remains active. After the value is returned, you can switch back and forth between the Exec and TEdit windows, simultaneously doing edits in the TEdit window while starting another TEdit, COPYFILE, doing a DEdit, etc. in the Exec window.

Conclusion: the Exec and the TEdit use separate windows and can be run simultaneously.

In technical terms, TEdit runs in a separate process from the Exec while DEdit runs in the same process as the Exec.

A *process* in computer jargon is a sequence of activities or tasks that must be carried out one after another.

Interlisp-D supports *multiple processes* -- that is you can be carrying out several sequences of activities simultaneously.

Within each of these processes or activity sequences, the activities are carried out in a strict sequential order.

Across processes, several such activity sequences can be going on simultaneously.

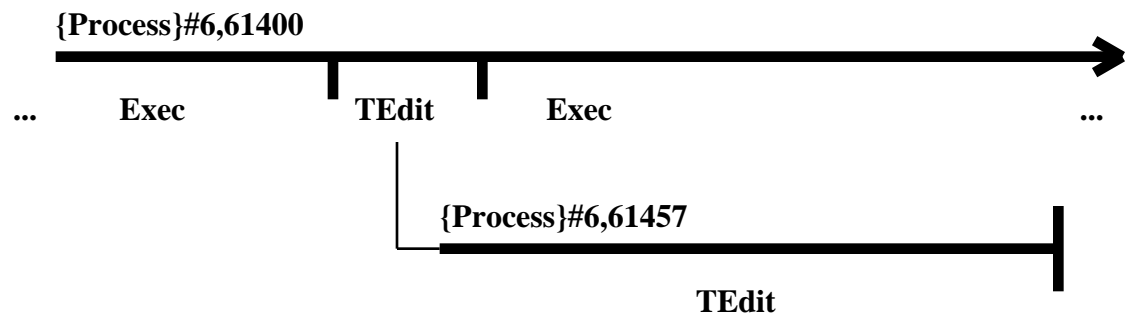
For complicated reasons, DEdit has been designed to run in the same activity sequence (i.e., process) as the Exec. When DEdit starts, the Exec goes into suspension until DEdit is finished.

Diagrammatically:

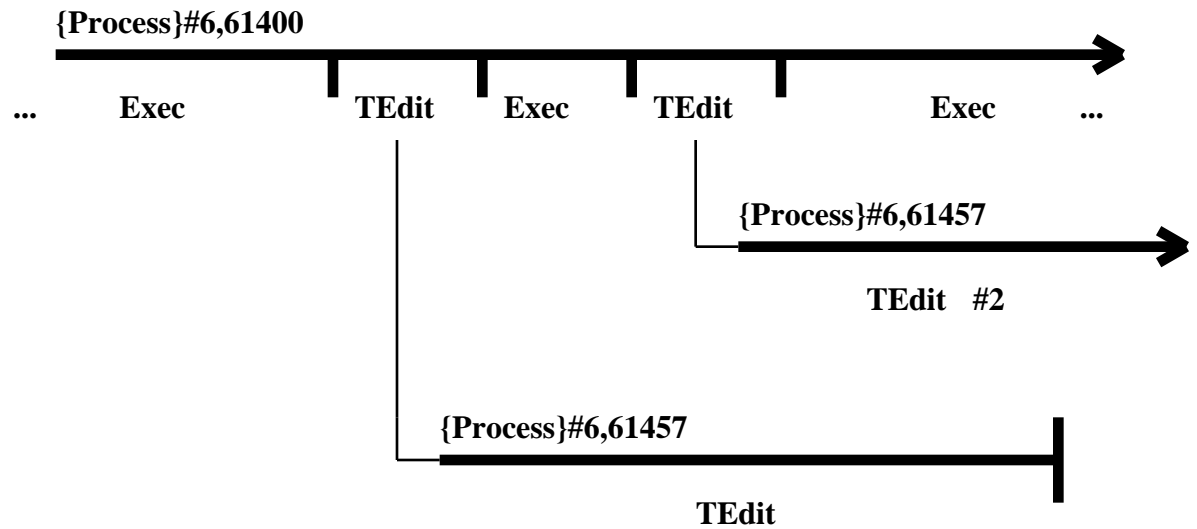


TEdit has been designed to run in a separate activity sequence (i.e., process) than the Exec. When TEdit starts, it starts up a new process. The Exec process then proceeds simultaneously with the Exec process.

Diagrammatically:



Or if you start up two TEdits:



The process of starting a new process from an old one is (sometimes) called *forking a process*.

Processes Scheduling: Time-sharing and all that

Time-sharing

Interlisp-D supports multiple processes in the sense that it (sometimes) appears that you can do multiple activities at once.

But, your D-machine can only do ONE thing at a time.

What actually happens is that Interlisp runs a little bit of one process, then a little bit of another process, then a little bit of a third process, and so on until the first process gets its turn again.

It all works because most of the time in any one process is spent *waiting*: waiting for the user to type the next character, waiting for the file server to respond, waiting for the disk to position to the right page, etc.

While the first process is waiting, the rest of the processes can run. While the rest of the processes are waiting, the first process can run.

This nifty little trick is called *time-sharing*.

The Scheduler

There is a part of Interlisp called the *scheduler* that determines which process gets its turn to run next.

The scheduler works as follows:

It runs through each process in the system giving each process control (i.e., permission to run) in turn. When a process gets control, it runs until it gives up that control. When a process gives up control, the control is passed to the next process in line.

This is called a *cooperative scheduler* because each process has to take the effort to give up control when it is waiting for something or when it feels like it has hogged-up too much time. The act of giving up control is called *blocking*.

There are systems with *preemptive schedulers* that force processes to give up control at regular intervals, so that every process gets a little bit each and every second. Processes need not worry about blocking.

One feature of a cooperative scheduler is that some processes don't play by the rules: they hog as much time as they like without blocking. When this happens, other processes don't get their turns to run.

For example, try starting up a TEdit. Then go back to the Exec window and start a LISTFILES on a fairly large Lisp file. Go back a starting editing in TEdit. At some point TEdit will just stop working for about 2 to 3 minutes. This is because LISTFILES goes into a part where it doesn't block for along period of time - it gets control and just hangs on to it until its work is finished, not allowing the TEdit and the other processes to have their turn to run.

E.F.S. -- An Experiment:

Try the following:

Start a TEdit.

Back in the Exec window, type "(FOR I FROM 1 do (PROMPTPRINT I))"

Try to use TEdit.

To kill the FOR: point in the Exec window, then type Ctrl-E.

Now try the following: (Note: (BLOCK) causes a process to block!!)

Start a TEdit.

Back in the Exec window, type "(FOR I FROM 1 do (PROMPTPRINT I) (BLOCK))"

Try to use TEdit.

To kill the FOR: point in the Exec window, then type Ctrl-E.

Cautionary Note: control hogging processes are just one reason for delays in Interlisp. Slow file servers are another. Don't automatically assume that if a process is slow, there is another process hogging the run time!

Job Control: Starting, Killing, Restarting, Suspending, & Waking Processes

You can do a variety of things to help you manage the processes running in your environment. The most important of these are the following:

Starting a process (& Process Names)

Most processes are started by programs forking a process to carry out a task.

Examples:

1. The function TEDIT opens a TEdit window and starts a TEdit process in that window.
2. The function call (LAFITE 'ON) starts a process that runs in the Lafite Status window and checks your mail every now and then.
3. The function CROCK starts a process that updates a clock on the screen once a minute.

You can also start a process by yourself from the Exec. The function ADD.PROCESS forks a new process. ADD.PROCESS takes as its first argument the form to be evaluated in the new process.

Example:

```
10_ ( ADD.PROCESS '(COPYFILE 'A 'B))
      {PROCESS}#3,65600
```

11_

Normally, COPYFILE runs as part of the Exec process, i.e., it doesn't automatically fork a process to do its work. Using ADD.PROCESS will start a new process and do a COPYFILE within that process. The COPYFILE will just run until its done. While its running, you can progress with whatever you want in the Exec, including starting another COPYFILE.

Process Names: Every process has a name. The name is usually the CAR of the form in the call to ADD.PROCESS. Thus the process created in the example above would have the name *COPYFILE*. If there is already a process with that name, a number is tacked onto the end of the name. For example, *COPYFILE#2*.

You can also give the process a name using (ADD.PROCESS *Form* 'NAME *ProcessName*). For example, (ADD.PROCESS '(COPYFILE 'A 'B) 'NAME 'AtoB) would start a process named "AtoB".

Killing a process

Most processes just evaluate a single function call as in the COPYFILE example above. These processes normally just run until the evaluation is completed. Then they just disappear.

Sometimes a process takes to long, gets hung up, or goes into a break. In this case, you may want to **kill** the process, i.e., stop the evaluation in progress and force the process to disappear.

Example:

You start a LISTFILES to print a file. This starts a process to send the files to the printer. But the printer is down. The process will just sit there printing in the PromptWindow *Quake not responding*. This can drive you crazy. So you kill the process started by LISTFILES and redo the LISTFILES when you know the printer is up again.

Killing is usually done using the Process Status window (described below). But can also be done by calling DEL.PROCESS from the Exec with the process name as an argument. E.g., (DEL.PROCESS 'COPYFILE)

Restarting a process

Some processes can be restarted from the beginning at any time before they finish. Restarting a process is just like killing it and then redoing the `ADD.PROCESS` that started the process to begin with.

Example:

You start a `HARDCOPY` in your TEdit window that never seems to finish because the file servers are slow and TEdit can't find the right fonts or some such nonsense. You just want to kill the `HARDCOPY` command and get on with your editing. Solution: Restart the TEdit process.

Note that you could kill the TEdit process. But then you'd have to go through the trouble of restarting TEdit in order to get back to editing. Moreover, if you hadn't put your edits, restarting TEdit may lose your edits. Therefore, restarting is much safer.

Restarting is usually done using the Process Status window (described below). But can also be done by calling `RESTART.PROCESS` from the Exec with the process name as an argument. E.g., `(RESTART.PROCESS 'TEdit#4)`

Suspending a process

Suspending a process stops a running process, but doesn't make it disappear. It stays around and inactive until it is either awoken (i.e., unsuspending) or killed.

Suspending a process is usually done by programs. But it can be useful in other ways.

Example:

You try to print something out with a LISTFILES. But the printer is down and the LISTFILES keeps printing *Quake not responding* in the Prompt window. You can stop this annoying printout by suspending the LISTFILES process until you know the printer is back up.

Suspending is usually done using the Process Status window (described below). But can also be done by calling SUSPEND.PROCESS from the Exec with the process name as an argument. E.g., (*SUSPEND.PROCESS* '*FILELISTING*').

Waking a process

A suspended process can be started at the exact point at which it was suspended by **waking** it.

Note that waking a process is a little like restarting a process. BUT a suspended process must be *woken*, it cannot be *restarted*. Restarting a suspended process, restarts the process from the beginning but does not unsuspend it. Therefore, doing a restart followed by a wake is basically the same as doing a wake followed by a restart. The wake unsusponds, the restart makes things start from the beginning again.

Example:

The printer is now fixed and you want your LISTFILES to pick up where it left off. You just wake the process you suspended earlier.

Waking a process requires that you return a value from the suspension. For processes suspended by the user, tis value is usually NIL.

Waking is usually done using the Process Status window (described below). But can also be done by calling `WAKE.PROCESS` from the Exec with the process name as an argument. E.g., (`WAKE.PROCESS 'FILELISTING`).

Processes and Breaks

Every process has its own, separate stack.

When an error occurs and a break is entered, the break occurs *within the process* causing the error. The Break Exec that is started is run in that process and the stack that you can examine using the Break Exec is the stack of that process.

Thus you can have as many Break Execs running as there are processes to break.

Example:

Your TEdit breaks due to some wierd error. This causes a break in the TEdit process, but will not effect any other running process. For example, if you have a COPYFILE going at the same time. The COPYFILE will progress as usual since the break has no effect on its process. If the COPYFILE then has an error, it will enter its own, independent Break window.

You can force a process to enter a break at any time from the Process Status window (described below). You cannot do this (easily) from the Exec.

You can also just observe the stack of a running process from the Process Status window. You cannot do this (easily) from the Exec.

The Standard Processes

Most of the time there will be 5 to 10 processes running in your system. Most of these processes are special system processes that handle things like the Ethernet, the mouse, the Lisp exec, file servers and so on.

Below is a list of the processes currently running on my system:

```

ERIS#LEAF
Tedit#3
Tedit#2
Tedit
MOUSE
LAFITEMAILWATCH
CROCK.PROCESS
PHYLUM#LEAF
\3MBWATCHER
EXEC
\NSGATELISTENER
\PUPGATELISTENER
\TIMER.PROCESS
BACKGROUND

```

The processes that I was directly responsible for were the three TEdit processes, the CROCK.PROCESS that is running my clock, and the LAFITEMAILWATCH process that is checking my mail every now and again.

The rest are all system processes carrying out some sort of system monitoring task. In general, I don't worry much about these system processes.

The TTY Process

There may be many processes running that require input from a keyboard. However, there is only one keyboard. In order to determine where the type-in from that keyboard goes, there is a notion called the *TTY process*.

The TTY process is a special designation given to only one process at a time. The process with this designation is the one that receives the keyboard input. The rest of the processes requiring keyboard input have to wait until they become the TTY process before getting any type-in.

For example, when you have multiple TEdits running, all but the one you are currently typing into are just sitting there waiting to become the TTY process.

If the process that is the TTY process has a window (as is almost always the case), then that window contains a blinking caret indicating where the typed input will appear.

When the process loses the TTY process designation, then the caret may remain, but it no longer blinks. Thus, the blinking caret almost always designates the window whose process is the TTY process.

If a process has a window, you can usually make that process be the TTY process by clicking a left or middle mouse button in its window. Basically, you point into the window you want to type into, then you type into that window.

You can also move the TTY process around using the Process Status window as described below.

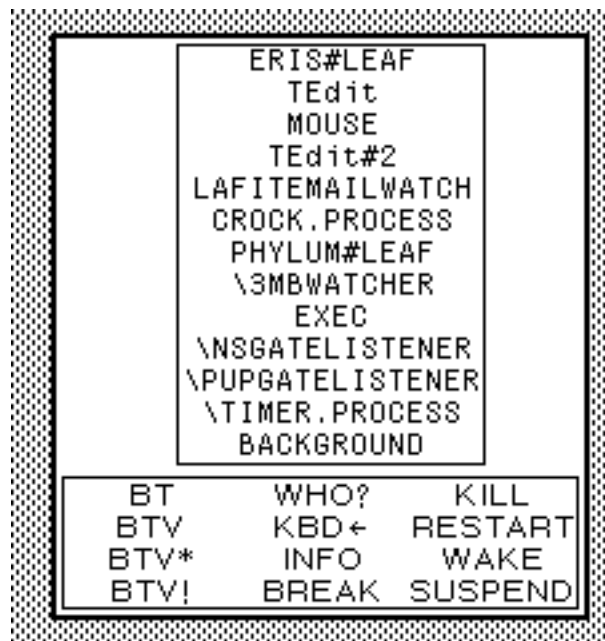
Finally, many programs move the TTY process around. For example, when you start a new TEdit, it grabs the TTY process for itself. When a TEdit quits, it gives the TTY process to the Lisp Exec.

The Process Status Window

The Process Status window provides an easy-to-use mouse/window-based interface to process management.

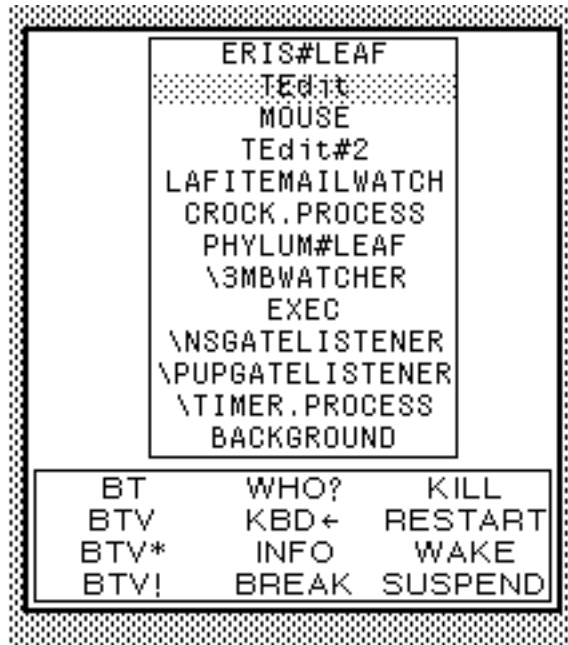
To open a PSW, select the PSW command from the background menu OR use the function call (PROCESS.STATUS.WINDOW) in the exec. Either way, you'll be asked to place the window on the screen.

The window will look like the following:



The upper box in this window contains a list of the names of all the processes currently in the system, both running and suspended.

You can *select* any one of these processes by clicking over the process name. This will shade this process name.



The lower box is a menu of commands that you can apply to the selected process.

The commands are: (starting from the right column)

KILL ž kills the selected process

RESTART ž restarts the selected process

WAKE ž wakes the selected process if its suspended. You will be given a menu of values that can be returned. Choosing NIL from this menu will generally work.

SUSPEND ž suspends the selected process

WHO? ž moves the selection to the process that has the TTY process.

This is very handy for telling multiple processes running the same program apart. For example, if there are 4 TEdits running, there will be

processes named TEdit, TEdit#2, TEdit#3, and TEdit#4. To find out which TEdit you want to operate on you can bug inside that TEdit's window. This will pass the TTY process to that TEdit. Then choose the WHO? command in the PSW. This will select the TEdit corresponding to the window you clicked in.

KBD_ ž gives the TTY process to the selected process.

Example: If TEdit has the TTY process, then selecting the EXEC process and using the KBD_ command will move the TTY Process (i.e., the blinking cursor) to the Exec window.

INFO ž returns any information the selected process wants to give. I have never seen a process that wanted to give any information, but ...

BREAK ž forces the selected process into a break, thus opening a break window, starting a break exec, etc.

This is handy to examine the state of a computation that has run amok:
Break the process and then muck around the stack using the Break Exec.

BT (BTV, ...) ž prints a backtrace of the stack of the selected process in an attached window just above (below) the PSW. BT prints standard backtrace with just the names of the functions called. BTV and the rest print backtraces with some more information about each frame.

```

Tedit
MOUSE
Tedit#2
LAFITEMAILWATCH
CROCK.PROCESS
PHYLUM#LEAF
\3MBWATCHER
EXEC
\NSGATELISTENER
\PUPGATELISTENER
\TIMER.PROCESS
BACKGROUND

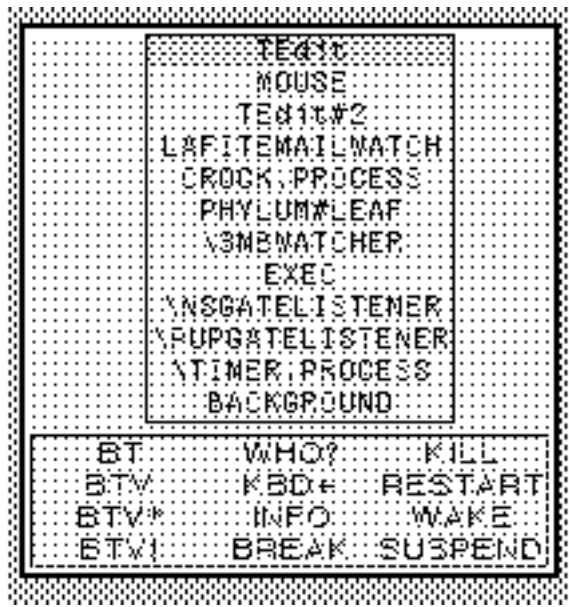
BT      WHO?      KILL
BTV     KBD←     RESTART
BTV*    INFO     WAKE
BTV!    BREAK    SUSPEND

Process backtrace
DISMISS
\TEDIT.COMMAND.LOOP
\TEDIT2
\EVALFORM
ERRORSET
\MAKE.PROCESS0
T

```

This is a handy way to find out if a process is progressing or if its stuck at some point: click BT many times in succession. If the stack is constantly changing, then things are progressing. If the stack is unchanging with `AWAIT.EVENT` or some such function at the top, then the process is probably stuck waiting for a file server to respond or some other external event to occur.

Whenever a process starts or finishes, the PSW gets shaded over. This indicates that the information is no longer current.



To update the information, just click the left or middle mouse buttons anywhere in the PSW. This will redisplay the PSW with the current information.

Multiple Exec Processes

In default mode, Interlisp has only one Exec process. There are, however, a number of Lisp Users packages that allow for multiple Exec windows and multiple Exec processes to run simultaneously.

One such package is called EXEC.

Load {eris}<LispUsers>Exec.DCOM. This will add a "EXEC" choice to the background menu.

Choosing EXEC from the background menu will prompt you for a region and then open a new Exec window in that region running an independent Exec process. You can use this new Exec simultaneously with and in exactly the same manner as the original Exec process.

For example, you can run a long COPYFILE in the original Exec window. While its running, you can carry out your normal tasks (e.g., starting TEdits, doing DIRs, etc.) using the second Exec.

You can open as many Exec windows and start as many Exec processes as you like.

To terminate a given Exec, evaluate the function call (*EXEC.QUIT*) in its Exec window.

Note: There is one way in which the multiple Execs are not independent: they share a common history list. Each event in every Exec updates the event number in all running Execs. From any Exec window you can redo or undo events originally carried out in any other Exec window.

This shared history list can sometimes be a great help. But it can also be problematic if you forget the shared nature of the list. In particular, nevent numbers printed in the Exec windows are not always up-to-date because they may change constantly due to events in other Exec windows. You very often tend to undo or redo the wrong event number because you rely on incorrect information printed in the Exec window and don't check the history list for the correct event number.

The Mouse Process and (SPAWN.MOUSE)

The process that sits in the background and watches your mouse movements and mouse button clicks is called the *MOUSE process*.

When you click inside a menu or a window, the mouse process is responsible for getting a decision about what to do from the menu or window, and then evaluating the correct functions necessary to do it. Thus much work in Interlisp gets done within the mouse process as button presses lead to the evaluation of various functions.

For example, when you choose TEDIT from the background menu, the MOUSE process evaluates the function call (*TEDIT*) just as if you had typed it into an Exec window.

In general, you shouldn't have to worry to much about the mouse process unless you are programming.

Sometimes, however, something goes wrong and the mouse process gets stuck. For example, a break can occur in a function being evaluated in the mouse process. Or a function can go into an infinite loop when being evaluated in the mouse process.

When this happens, you can't use your mouse for anything. You can often, however, still type in to an Exec or a Break Exec.

When this happens, the function call (SPAWN.MOUSE) comes in very handy. SPAWN.MOUSE will start up a new mouse process, replacing the old.

The old function keeps going and finishes whatever it was doing when SPAWN.MOUSE was called, except that it is renamed OLDMOUSE.

The new mouse process takes over control of the mouse and hence all of the mouse functions are restored.

Example:

You choose an item from a Lafite Browser menu. This immediately causes a break. However, in the Break window you discover that the mouse doesn't work at all. In fact, no where on the screen does the mouse have any effect.

If the flashing cursor is in the Break window or in a Exec window, then you could type (*SPAWN.MOUSE*). This would free up the mouse by starting a new MOUSE process. The break in the Lafite menu command would remain unaffected, although its process would be renamed from MOUSE to OLDMOUSE.

References

Processes are documented in Section 18.20 of the IRM.

Subsections 18.20.1, 18.20.2, & 18.20.5 are generally relevant for non-programming users.

Subsection 18.20.6 covers the TTY process.

Subsection 18.20.7 covers the MOUSE process and SPAWN.MOUSE.

Subsection 18.20.8 covers the Process Status window.

LispCourse #15: Files and Directories ž Part 1

Basic Concepts

What is a file?

A file is a "place" where you keep information.

A file is the basic unit of permanent external storage in Interlisp.

A file is a bunch of information or data that should be manipulated in one chunk.

A file is the basic organizational construct for storing data in Interlisp.

???

File Names

Every file has a name. In general, file names have two parts: a *name* and an *extension*. Both the name and the extension are litatoms with no spaces or other unusual characters such as Tabs, ":", ".", ",", etc.

In the file name, the name and the extension are separated by a period ("."). For example, *FILE.EXT* is a file name. So is *NewData3-3-85.TED*. In this latter example, *NewData3-3-85* is the name part and *TED* is the extension part.

The extension part is optional. *XYZZY* is a perfectly good file name.

The name part of the file name is usually used to indicate what the file contains. The extension part is usually indicative of the flavor of the file. (See discussion of flavors below.)

Note that Interlisp is NOT case sensitive with respect to file names. The file *FOO.BAR* and the file *Foo.Bar* are the SAME file according to Interlisp.

Flavors of files

From one point of view, all files are alike: they are an ordered stream of bits (1s and 0s) that must be stored somewhere for later retrieval.

From another point of view, there are many different *flavors* of files.

The pattern of 1s and 0s in a file is really a code that stands for some meaningful data or information.

In order to extract the meaningful information in a file, we need to know how to interpret the format of the bits in that file.

Flavors of files arise because different files contain different kinds of meaningful information and represent that information using different codes and patterns of bits.

Examples:

TEdit files contain formatted text.

Bravo files also contain formatted text

(but using a different "code" than TEdit files)

Lafite mail files contain a sequence of mail messages

DCOM files contain compiled Lisp programs

SYSOUT files contain Interlisp virtual memories

Each of these types of file contains a different kind of information and/or represents information using a different code.

You alternate between these two viewpoints when dealing with files in Interlisp, depending on whether you are dealing with the *file itself* or with the *information in the file*.

A function like COPYFILE deals with the *file itself* and therefore treats a file like a stream of bits to be copied from one location to another. Thus COPYFILE is very general and can operate on files of any flavor.

Functions like TEdit, LOAD, and LAFITE make use of the *information within a file* and therefore work only with files of certain flavors. They will totally misinterpret files of the wrong flavor. Or worse, they often crash when processing files of the wrong flavor.

The flavor of a file is (in Interlisp) *intensionally* defined. There is no way (other than naming conventions) to specifically declare a file to be of a particular flavor.

The system itself treats all files as bit streams. This bit stream will be interpreted as meaningful information or as garbage depending on what function you use to access it.

For example, you can access a DCOM file using TEdit, but it will appear as (mostly) garbage in the TEdit window. Similarly, you can read in a TEdit file using LOAD, but in all probability the LOAD will bomb in a very short time.

It's up to each user to use TEdit on files that are meant to be TEdit files and to use LOAD on files that are meant to be Lisp files.

The system provides only minimal help in helping you with this task!! The most you can expect is that a function will detect and inform you that you have asked it to work on a file it can't interpret.

To overcome the lack of the distinction between flavors of files, the *convention* has been established that the extension of each file name (see discussion of file names below) gives some indication of the file's flavor.

Examples:

TEdit files end in .TED or .TEDIT

DCOM files end in .DCOM

SYSOUT files end in .SYSOUT

Lafite files end in .MAIL

Devices

The stream of bits that makes up a file has to be stored on some "*physical*" device such as a floppy, a disk, a file server, etc.

Interlisp supports file storage on many different kinds of devices.

Interlisp also tries to make file storage on all of the different devices behave as similarly as possible.

But different devices have different characteristics, both hardware and software, and Interlisp cannot always cover up these differences.

So. How you use files in Interlisp depends a little bit on what device the file is stored on.

Moreover, as you move a file from device to device, its characteristics may change slightly.

I will try to point out these device-dependencies as we go along.

The devices supported by Interlisp-D are the following:

Local Disk

DLion local disk ž 1 or more logical volumes used for Lisp files

Dolphin local disk ž 2 Alto partitions

Dorado local disk ž 5 or 19 Alto partitions

File Servers

IFS ž e.g., Phylum, Eris, etc.

NS ž e.g., LispFiles:, StarFiles:

Vax (Unix) ž e.g., PARC-CSLI, PARC-VAXC

Floppy Disks (DLion only)

Core Devices (simulated devices in the Lisp virtual memory)

Device Names:

All devices have a name. When used to refer to a file, the name of the device is enclosed in curly-brackets. The device name conventions are as follows:

Local Disk

DLion local disk ž {DSK}

Dolphin local disk ž {DSK}, {DSK1}, {DSK2}

Dorado local disk ž {DSK}, {DSK i }

File Servers

IFS ž the name of the server, e.g., {Phylum}, {Eris}, etc.

NS ž the NS network address of the server or its abbreviation as in {LispFiles:PARC:Xerox} or {LispFiles:} for short. Note the abbreviated name always ends in a ":".

Vax (Unix) ž the name of the machine, e.g., {PARC-CSLI}, {PARC-VAXC}

Floppy Disks ž {FLOPPY}

Core Devices ž more than one core device can be created using the function COREDEVICE. The name of the core device is set by this function. See p. 18.13 of the IRM. The device {CORE} is already created in every system.

Directories

Different devices organize the files they contain in different ways.

Some devices use a **flat** organization, i.e., the files are just stored on the device one after the other. On some devices the files are ordered (e.g., alphabetically). On some devices the files are randomly ordered.

Example:

```
{DSK}
ACCOUNTANT.RUN;1
ALTOD0MC.EB;1
ALTOD1MC.EB;1
CHAT.RUN;1
Com.cm;1
DiskDescriptor.;1
DoradoLisp.MB;1
DORADOLISPMC.EB;1
Dumper.boot;1
EMPRESS.RUN;1
Executive.Run;1
Fonts.Widths;1
FOOTNOTES.TEDIT;2
FOOTNOTES.TEDIT;1
Ftp.log;1
Ftp.Run;1
INIT.LISP;1
LISP.RUN;1
LISP.SYMS;1
LISP.VIRTUALMEM;1
Rem.Cm;1
Swat.;1
Swatee.;1
Sys.Boot;1
SYS.ERRORS;1
SysDir.;1
SysFont.Al;1
```

User.Cm;1

However, most devices use **directories** and **subdirectories** to organize the files they contain into logical groupings:

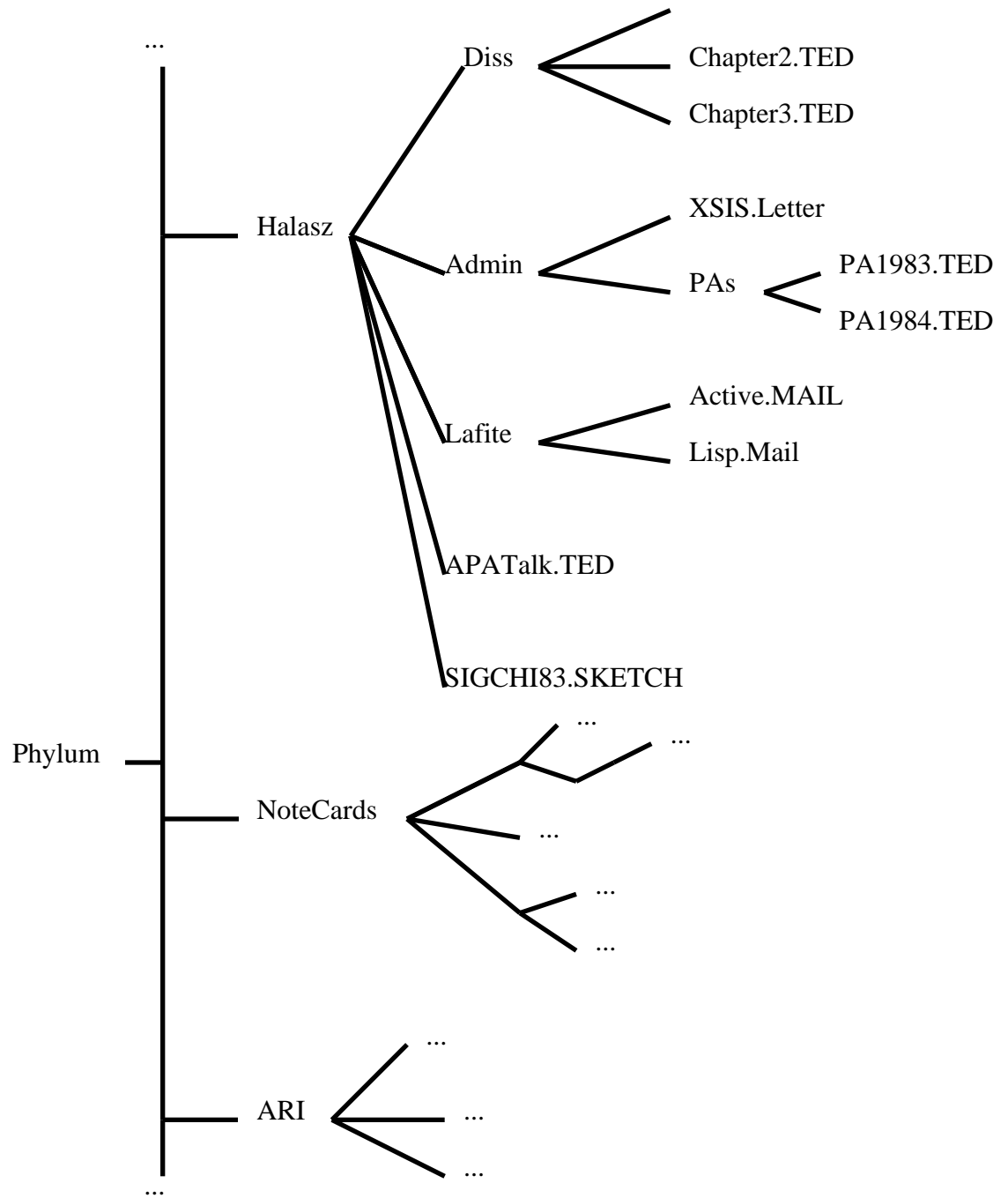
Basically a **directory** is just a *list* of 1 or more files that are stored together in the same logical "place" on the device.

A **subdirectory** is a *list* of 1 or more files from a single directory that are stored together in the same logical "place" within that *directory*.

Every directory and sub-directory has a name, which is usually a short atom.

Directories and subdirectories organize the files on a device into a tree structure. The device is divided into a set of directories. Each directory is sub-divided into zero or more sub-directories. Each sub-directory is in turn divided into zero or more sub-sub-directories. Files can be placed in any directory or sub-directory in this tree structure.

Diagrammatically:



Path Names

The directory or sub-directory that a file is located in can be uniquely specified by a path that starts at the top-level directory and includes each sub-directory in turn until the file is reached in the tree structure.

When referring to this path name, the directory name is enclosed in angle brackets, while each sub-directory is terminated by an end angle bracket.

Examples:

PA1984.TED is in <Halasz>Admin>PAs>

APA.Talk is in <Halasz>

Lisp.mail is in <Halasz>Lafite>

Creating Directories/Sub-directories:

On some devices directories and sub-directories have to be explicitly created before any files can be entered into them. The method for creating directories and sub-directories differs between devices. On some devices, e.g., IFSs, it requires an administrator with special privileges to create a new directory.

Directories that are created explicitly remain even if they have no files in them, e.g., after the last file in the directory/sub-directory has been deleted.

On devices that don't require directories and/or sub-directories to be explicitly created, you can create directories and/or sub-directories simply by creating a file that has the new directory/sub-directory as part of its name. The directory and/or sub-directory will be created when the file is created.

Directories/sub-directories that are created implicitly by naming a file disappear after the last file with the directory/sub-directory in its name is deleted.

The various devices and their directory types are as follows:

Local Disk

DLion ž directories are logical volumes on disk created during initial disk partitioning; sub-directories are created implicitly by naming.

Dolphin/Dorado ž each Alto partition is a separate device (named DSK1 thru DSK n). No directories are supported within these devices, i.e., each device has a FLAT structure.

File Servers

IFS ž directories are created explicitly by an IFS administrator, subdirectories are created implicitly by naming.

NS ž directories (called file drawers) are created explicitly by an administrator, sub-directories are created explicitly by naming. Note, however, deleting the last file from a sub-directory DOES NOT delete the sub-directory.

Vax (Unix) ž directories are created explicitly by an administrator, sub-directories are created explicitly by the user. To create a sub-directory, you must log into the Vax and use the Unix mkdir command.

Floppy Disks ž support either a flat file structure or a hierarchical (directory/sub-directory) file structure or both. Both directories and sub-directories are created implicitly by naming.

Core Devices ž support either a flat file structure or a hierarchical (directory/sub-directory) file structure or both. Both directories and sub-directories are created implicitly by naming.

Versions

Interlisp supports file **versions**. That is, Interlisp allows two files with the same name to be on the same device in the same directory/sub-directory.

When this happens, the files are considered to be different *versions* of the same file.

Every file has a **version number**. When a file of a given name in a given device/directory is first created, it is given a version number of 1. When a new version is created, it is assigned a version number one higher than the highest version already in existence.

In Interlisp, the version number is specified after the file name, preceded by a semi-colon. For example, *TESTFILE.TED;3* and *TESTFILE.TED;4* are two versions of the same file. *NewFile;6* and *NewFile;8* are two versions of another file.

Different versions of the "same" file are actually totally separate files and could be treated as such. Version1 of a file could be a TEdit file and version 2 a DCOM file.

However, this would violate the *intent* of versioning. Different versions of a file are supposed to be different versions (e.g., updates) of the "same" information. Lots of little things in Interlisp support this notion.

Examples:

When you name a file without a version number, Interlisp always thinks you mean the latest version of the file. To get to an earlier version of a file, you have to specify the exact version number. (Exception is in deleting a file, where Interlisp always assumes you mean the lowest version number unless otherwise specified.)

When you PUT a file, TEdit just writes a new version of the file. The old (unedited) version still exists.

When you do a MAKEFILE, the system just writes a new version of the file. The old (i.e., previous) version still exists and can still be LOADED by specifying its version number in the LOAD command.

Full File Names

A file is uniquely specified by a full file name containing the name of the device its on, its path name, its file name and its version.

For example, *{phylum}<halasz>lisp>init.lisp;4* is a full file name.
{DSK}FOO.TED;3 and *{FLOPPY}<FOO>BAZ;55* are also full file names.

Ultimately ALL file references in Interlisp MUST specify a full file name. If this wasn't the case, there would be ambiguity as to which file was being referenced.

However, Interlisp allows elliptical references to files that allow you to use much less than the full file name when referring to a file.

For example, if a version number is not specified, Interlisp assumes you mean the highest version number.

Also, Interlisp will use your connected directory (see the CONN command below) if you don't specify a device and directory.

Thus, the name *FOO.BAR* may be an allowable abbreviation of *{phylum}<halasz>lisp>foo.bar;5* if you are connected to *{phylum}<halasz>lisp>* and 5 is the highest version of FOO.BAR in *{phylum}<halasz>lisp>*.

Note: it almost NEVER hurts to specify the full file name of a file if you know it.

Reminder Note: Interlisp is not case-sensitive with regard to file names.

File Attributes

In addition to their name, all files have some attributes. The attributes attached to a file depend somewhat on the device the file is on. However, in general a file has the following attributes:

SIZE ž the size of the file in disk pages (512 bytes each)

LENGTH ž the length of the file in bytes (~ 512 times its size).

AUTHOR ž login name of person who created this file

CREATIONDATE ž the date and time of the creation of the file.

READDATE ž the date and time when this file was last read.

WRITEDATE ž the date and time when this file was last written to.

TYPE ž text or binary (to help along some older computers and file servers for which this was an important distinction)

BYTESIZE ž in Interlisp-D always 8, but some older computers and file servers allow other sizes.

The attributes of a file can be seen using the FILEBROWSER or the DIR command. Both of these are described below.

You can also access the attributes of a file one at a time using the function GETFILEINFO which has two arguments: the (full) file name and the name of the attribute you want to see.

Example:

```
10_ (GETFILEINFO ' <halasz>lisp>init 'SIZE)
```

```
15
```

```
11_
```

In general, you do not change the attributes of a file directly. The attributes change as you work with the file, e.g., as you add information to the file, its size increases.

File Protection

Some devices allow an additional set of attributes for a file: *the protection code*. The protection code of a file determines who can read or write the file.

Protection schemes vary from device to device.

On the Local Disk, Floppy and Core devices there are no protection codes for files. Basically, anyone can read or write onto any file on these devices.

There is some protection for Dolphin/Dorado local disks since whole Alto partitions are passworded. You cannot read from or write to any file on an Alto partition (i.e., on {DSK*i*} for any *i*) unless you can supply the password for that partition. However, once you give the correct password, all the files on that partitin can be accessed at will.

On the IFS file servers, a file has three separate protections:

a *read* protection ž who can read from this file

a *write* protection žwho can write on this file

an *append* protection ž who can append something to the end of this file

Each of these is a list of people or distribution lists who have the permission carry out the specified action. The list is either a person's grapevine name (e.g., Halasz.pa), a grapevine distribution list (e.g., USR^.pa), the atom *owner* (indicating the AUTHOR of the file), or the atom *USRegistries^.Internet* (basically indicating anyone with access to the Xerox network).

For example, {*phylum*}<*halasz*>*lisp*>*init* has the protection:

R: USRegistries^.Internet Owner; W: Owner; A: Owner

This means that any one in the Xerox world can read this file but only I (as owner of the file) can write or append to it.

But, {*ERIS*}<*LispCore*>*next*>*full.sysout* has the protection:

R: Lispcore^.pa LispCoreAccess^.pa Owner; W: Lispcore^.pa Owner; A: Lispcore^.pa Owner

This means that any one on the distribution lists called `Lispcore^.pa` and `LispCoreAccess^.pa` can read this file but only people on `Lispcore^.pa` (and its owner whoever it may be) can write or append to it.

To list or change the protection of a file on an IFS, you have to CHAT to the IFS.

To see the protection codes of a file type:

List *FileName*, <RETURN>

The IFS will respond with a @@ prompt.

Type:

Protection<RETURN><RETURN>

This will list the file with its protection codes as above.

To change the protection codes type:

Change Protection *FileName*<RETURN>

The IFS will respond with the @@ prompt.

Type:

Write *PersonOrDistList*<RETURN><RETURN>

This will add the designated person/group to the WRITE permission list of the file. Use *Read* or *Append* instead of *Write* to change the Read and Append protections. If the Read, Write, or Append is preceded by a *No*, then the designated group/person is removed from the permission list. (E.g., *No Read LispCore^.pa*).

The Vax and NS file servers have similar, but different protection schemes. See you file server administartor for information on the relevant protection schemes and how to use them.

References

Files are covered in Chapter 6 and Sections 18.16 and 18.17 of the IRM. User relevant material is scattered throughout these sections, so you'll just have to skim for what you're interested in.

The file protection scheme for the IFS is covered in the "How to use the IFS" memo on {indigo}<ifs>howtouse.bravo (or .press).

LispCourse #16: Files and Directories ž Part 2

Update on Processes

When dealing with processes, I neglected to mention the following situation: For each formatting menu that is used in TEdit, a new TEdit process is started. Thus if you have one TEdit running with the Paragraph-Looks Menu open above that window, you will see 2 Tedit processes running in your PSW. The process called *TEdit* will refer to the TEdit itself, and the process called *TEDIT#2* will refer to the process operating the Paragraph Menu. This situation can sometimes be confusing!!

```
ERIS#LEAF
RTP#2
RTP
TEDIT#2
PHYLUM#LEAF
\3MBWATCHER
MOUSE
LAFITEMAILWATCH
CROCK.PROCESS
TEdit
EXEC
\NSGATELISTENER
\PUPGATELISTENER
\TIMER.PROCESS
BACKGROUND

BT      WHO?      KILL
BTV     KBD←     RESTART
BTV*    INFO      WAKE
BTV!    BREAK     SUSPEND
```

Dealing with Directories

The *Connected* Directory

At any given time you are *connected* to some device and directory. Connecting to a device and directory means that that device/directory is used as the default whenever a file name is specified with no device and/or directory.

Whenever a file name is specified without a device and path name, the device/pathname is assumed to be the current *connected directory*.

Essentially, the connected directory is appended to the beginning of every file name that you type in without a device and path name.

If a file name is specified with a path name but without a device, then the device part of the current connected directory is used as the device.

Examples:

Connected Directory: *{Phylum}<halasz>lisp>*

File Name: *init == {Phylum}<halasz>lisp>init*

File Name: *new>init == {Phylum}<halasz>lisp>new>init*

File Name: *<jones>lisp>init == {Phylum}<jones>lisp>init*

File Name: *{Eris}<jones>lisp>init == {Eris}<jones>lisp>init*

Form: *(DELFILe 'ABC) ==*

(DELFILe '{Phylum}<halasz>lisp>ABC)

Form: *(DELFILe '{Eris}<jones>lisp>init) ==*

(DELFILe '{Eris}<jones>lisp>init)

LOGINHOST/DIR

The value of the variable LOGINHOST/DIR is the default connected directory. GREET sets your connected directory to this value (e.g., whenever you load a new sysout). CONN and CNDIR also use this variable (see below).

Usually LOGINHOST/DIR is set in your INIT file to your "home" directory. If not set in your INIT file, it will default to {DSK}.

Connecting to a directory:

There are two equivalent ways to change the current connected directory:

CONN *device/pathname* ž a P.A. command to change the currently connect directory. *device/pathname* is the directory to connect to.

(CNDIR *device/pathname*) ž a function to change the current connected directory. *device/pathname* is the directory to connect to. CNDIR returns the full path name of the directory being connected to.

Examples:

```
10_ CONN {phylum}<halasz>
```

```
{phylum}<halasz>
```

```
11_ (CNDIR '{phylum}<halasz>lisp>)
```

```
{phylum}<halasz>lisp>
```

```
12_ (CNDIR '<jones>lisp>)
```

```
{phylum}<jones>lisp>
```

Notes:

1. If *device/pathname* is NIL, the value of the variable LOGINHOST/DIR will be used.

2. If *pathname* is NIL but there is a device specified, then for any device that supports directories, the directory will be set to the user's login name.

Examples:

```
CONN {phylum} => CONN {phylum}<halasz>
```

```
CONN {dsk} => CONN {dsk}
```

3. CONN is UNDOable, but CNDIR is not.

4. As described above, certain devices require that a directory be created before you can connect to it. Other devices further require that all of the sub-directories in a path name exist before you can connect to the directory specified by that path name.

Examples:

```
CONN {phylum}<foo> == error
```

```
CONN {phylum}<halasz>foo == okay
```

```
CONN {FLOPPY}<FOO> == okay
```

Asking "what is the current connected directory?"

The function **DIRECTORYNAME** can be used to find out about LOGINHOST/DIR and about the current connect directory.

(DIRECTORYNAME) returns the value of LOGINHOST/DIR

(DIRECTORYNAME T) returns the current connected directory.

The DIRECTORIES List

Whenever you specify a file name that can't be found, DWIM tries to "correct" the spelling of the name.

If there is no device/pathname specified and the file does not exist in the currently connected directory, then DWIM will consult the value of the variable DIRECTORIES.

DIRECTORIES is a *ordered* list of device/pathnames for DWIM to look on for any file that it cannot find. DWIM will "temporarily connect" to each device/pathname on this list until it finds one that contains the file name it is looking for.

Example:

```
14_ (SETQ DIRECTORIES '({phylum}<halasz> {phylum}<halasz>lisp>
{phylum}<notecards> {eris}<lispusers>))
```

```
({phylum}<halasz> {phylum}<halasz>lisp> {phylum}<notecards>
{eris}<lispusers>)
```

```
15_ CONN {eris}<lisp>
```

```
{eris}<lisp>
```

```
16_ (TEDIT 'INIT)
```

```
={PHYLUM}<HALASZ>LISP>INIT
```

```
{process}#6,24304
```

[Since there was no file called INIT on the connected directory, {eris}<lisp>, DWIM tried to find the file. Using the DIRECTORIES list, it first looked for the file {phylum}<halasz>init. Since that file doesn't exist, it looked for the file {PHYLUM}<HALASZ>LISP>INIT, which does exist. So it used that as the "corrected" file name.

```
17_ (TEDIT '{ERIS}<lisp>INIT]
```

```
FILE NOT FOUND
```

```
{ERIS}<lisp>INIT
```

[DWIM did not use the DIRECTORIES list here because the Device/Pathname was already specified.]

DIRECTORIES should be set in your INIT file to a list of the directories where you typically keep files of general interest. The DIRECTORIES list is initialized in the system INIT file to include directories like *{eris}<lispusers>* and *{eris}<lisp>harmony>library>*. Therefore, your INIT file should use the ADDVARS file package command.

My INIT file contains the following clause:

```
(ADDVARS
  (DIRECTORIES
    {DSK}
    {DSK2}
    {PHYLUM}<HALASZ>LISP>
    {PHYLUM}<HALASZ>
    {PHYLUM}<NOTECARDS>RELEASE1.2>LIBRARY>
    {PHYLUM}<NOTECARDS>RELEASE1.1>))
```

Finding out what files are in a directory ž DIR, FILDIR etc.

One of the most common operations in using files is finding out what files you have in a given directory. For example, "what files do I have on {phylum}<halasz>?".

The functions FILDIR, DIR, and DIRECTORY can be used to get this information.

(FILDIR *FileNamePattern*) ž a function that returns a list of all of the full file names matching the the *FileNamePattern*. The *FileNamePattern* can be any part of a full file name. It may contain the * and ? wildcards, standing for *any number of any character* and *any one character*, respectively.

The parts of the full file name are defaulted as follows:

If the device name is left out, the device name in the current connected directory will be used.

If the device/pathname is left out, then the current connect directory will be used.

If no file name is specified, then the name is assumed to be *.*.*.

If the extension is left off, then the extension and version is assumed to be *.*. For example, *FOO* == *FOO.*.**

If the version is left off, then the version is assumed to be *. For example, *FOO.BAR* == *FOO.BAR.**

Examples:

(FILDIR) == all files in the connected directory

(FILDIR '.*.DCOM)* == all files in the connected directory that have a DCOM extension

(FILDIR '{ERIS}<LISP>LIBRARY>) == all files on {eris}<lisp>library>

(FILDIR '{phylum}<notecards>library>.DCOM)* == all files in {phylum}<notecards>library> that have a DCOM extension.

*(FILDIR '{phylum}<halasz>*outline*)* == all files on {phylum}<halasz> and its subdirectories that contain the string "outline".

Result is:

```
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE.FORM;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE01.TED;7
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE01.TED;8
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE01.TED;9
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE02.TED;2
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE02.TED;3
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE02.TED;4
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE03.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE04.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE04.TED;2
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE05.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE06.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE07.TED;13
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE07.TED;14
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE07.TED;15
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE08.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE09.TED;1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE10.TED;3
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE10.TED;4
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE10.TED;5
```

```

{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE11 . TED ; 7
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE11 . TED ; 8
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE11 . TED ; 9
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE12 . TED ; 9
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE12 . TED ; 10
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE12 . TED ; 11
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE13 . TED ; 12
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE13 . TED ; 13
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE13 . TED ; 14
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE14 . TED ; 1
{ PHYLUM } <HALASZ>LISPCOURSE>OUTLINE15 . TED ; 1

```

DIR *FileNamePattern* Commands ž a P.A. command that returns a list of all of the full file names matching the the *FileNamePattern*. The *FileNamePattern* is exactly as in FILDIR.

The *Commands* consists of one or more atoms that specify the additional attributes that should be printed for each file AND/OR an action that should be carried out on each file listed. The possible atoms are:

READDATE ž print the date/time each file was last read

WRITEDATE ž print the date/time each file was last written on

CREATIONDATE ž print the date/time each file was created

SIZE ž print the size (in pages) of each file

LENGTH ž print the length (in bytes) of each file

AUTHOR ž print the author of each file

DELETE ž delete each file

PROMPT *msg* ž print *msg* then wait for the user to type "Y" or "N". If the user types "N" skip the rest of the commands for the current file.

Examples:

Print the names and creation dates of all the DCOM files in the connected directory: *DIR *.DCOM CREATIONDATE*

Result:

```

{ PHYLUM } <notecards>RELEASE1 . 2 >LIBRARY>
INTERVALTEST . DCOM ; 1          25-Mar-85  0:10:21 PST
NCCHAIN . DCOM ; 1              2-Jan-85   3:00:34 PST
NCCLUSTER . DCOM ; 1           12-Feb-85  3:11:14 PST

```

```

NCCLUSTER.DCOM;2          25-Mar-85  0:13:02 PST
NCFILECARD.DCOM;9        12-Feb-85 23:32:47 PST
NCFILECARD.DCOM;10       12-Feb-85 23:47:14 PST
NCFILECARD.DCOM;11       12-Mar-85 18:49:23 PST
NCFILESUBSTANCE.DCOM;1   12-Feb-85 23:33:04 PST
NCFILESUBSTANCE.DCOM;2   12-Feb-85 23:48:39 PST
NCFILESUBSTANCE.DCOM;3   12-Mar-85 18:48:46 PST
NCKEYS.DCOM;1            7-Feb-85  20:33:25 PST
NCKEYS.DCOM;2            22-Mar-85 11:06:40 PST
NCSCREEN.DCOM;1          11-Feb-85 14:38:00 PST
NCSCREEN.DCOM;2          25-Mar-85  0:15:05 PST

```

Print the names, creation dates and authors of all the files in
 {phylum}<notecards>release1.1>library>:

DIR {phylum}<notecards>release1.1>library> CREATIONDATE AUTHOR

Result:

```

  {PHYLUM}<notecards>RELEASE1.1>LIBRARY>
NCCHAIN.;6                2-Dec-84 13:23:32 PST trigg.PA
NCCHAIN.;7                10-Dec-84 17:37:45 PST trigg.PA
NCCHAIN.;8                3-Jan-85  3:00:17 PST trigg.PA
NCCHAIN.DCOM;4            3-Jan-85  3:00:34 PST trigg.PA
NCCLUSTER.;11            3-Jan-85  3:01:00 PST trigg.PA
NCCLUSTER.;12            3-Jan-85 19:58:09 PST trigg.PA
NCCLUSTER.;13            3-Jan-85 23:52:03 PST trigg.PA
NCCLUSTER.;14            12-Feb-85  3:10:55 PST trigg.PA
NCCLUSTER.DCOM;9         3-Jan-85 23:52:55 PST trigg.PA
NCCLUSTER.DCOM;10        12-Feb-85  3:11:14 PST trigg.PA
NCKEYS.;3                4-Feb-85 13:08:12 PST Halasz.PA
NCKEYS.;4                4-Feb-85 13:10:46 PST Halasz.PA
NCKEYS.;5                4-Feb-85 13:12:35 PST Halasz.PA
NCKEYS.;6                4-Feb-85 19:32:46 PST Halasz.PA
NCKEYS.;7                4-Feb-85 22:16:11 PST Halasz.PA
NCKEYS.DCOM;3            4-Feb-85 19:33:05 PST Halasz.PA
NCKEYS.DCOM;4            4-Feb-85 22:16:34 PST Halasz.PA
NCKEYS.TED;2             4-Feb-85 18:05:06 PST Halasz.PA
NCKEYS.TED;3             4-Feb-85 18:14:39 PST Halasz.PA
NCKEYS.TED;4             4-Feb-85 18:15:39 PST Halasz.PA
NCKEYS.TED;5             4-Feb-85 18:24:10 PST Halasz.PA
NCKEYS.TED;6             4-Feb-85 18:26:46 PST Halasz.PA
NCSCREEN.;3              3-Jan-85  3:02:39 PST trigg.PA
NCSCREEN.;4              3-Jan-85 20:00:18 PST trigg.PA
NCSCREEN.;5              4-Jan-85  2:40:02 PST trigg.PA
NCSCREEN.;6              11-Feb-85 14:37:29 PST Trigg.PA
NCSCREEN.DCOM;4          4-Jan-85  2:40:59 PST trigg.PA

```



```

NCSCREEN.DCOM;5    11-Feb-85 14:38:00 PST Trigg.PA
VIDEOTAPE.;1      11-Feb-85 16:24:01 PST Halasz.PA

```

DELETE all of the files on the local disk's partition 5: *DIR {DSK5}*

DELETE

Result:

```

      {DSK5}
ALTOD1MC.EB;1 c      deleted
Com.cm;1             deleted
DiskDescriptor.;1   deleted
DORADOLISPMC.EB;1  deleted
Dumper.boot;1       deleted
Executive.Run;1     deleted
FONTS.WIDTHS;1     deleted
HOLD.NOTEFIL;1     deleted
INIT.LISP;1         deleted
LISP.RUN;1          deleted
LISP.SYMS;1         deleted
LISP.VIRTUALMEM;1  deleted
REM.CM;1            deleted
Swat.;1             deleted
Swatee.;1          deleted
Sys.Boot;1         deleted
SYS.ERRORS;1       deleted
SysDir.;1          deleted
SysFont.Al;1       deleted
User.Cm;1           deleted

```

Go thru all the files on {DSK5} and ask the user if they should be deleted:

DIR {DSK5} PROMPT "Delete? " DELETE

Result:

```

      {DSK5}
ALTOD1MC.EB;1      Delete? Yes      deleted
Com.cm;1           Delete? Yes      deleted
DiskDescriptor.;1  Delete? No
DORADOLISPMC.EB;1 Delete? No
Dumper.boot;1     Delete? No
Executive.Run;1   Delete? Yes      deleted
FONTS.WIDTHS;1   Delete? Yes      deleted
HOLD.NOTEFIL;1   Delete? Yes      deleted
INIT.LISP;1       Delete? Yes      deleted
LISP.RUN;1        Delete? Yes      deleted
LISP.SYMS;1       Delete? Yes      deleted

```

LISP.VIRTUALMEM;1	Delete?	<i>No</i>	
REM.CM;1	Delete?	<i>Yes</i>	deleted
Swat.;1	Delete?	<i>Yes</i>	deleted
Swatee.;1	Delete?	<i>Yes</i>	deleted
Sys.Boot;1	Delete?	<i>Yes</i>	deleted
SYS.ERRORS;1	Delete?	<i>Yes</i>	deleted
SysDir.;1	Delete?	<i>Yes</i>	deleted
SysFont.Al;1	Delete?	<i>Yes</i>	deleted
User.Cm;1	Delete?	<i>Yes</i>	deleted

Note: The *Italics* indicate the users response.

(DIRECTORY *FileNamePattern Commands*) ž is a function that acts just like the DIR P.A. command. The only difference is that DIRECTORY doesn't automatically list the file names. There is an additional command (**P**) to do this.

Example: (*DIRECTORY '{PHYLUM}<HALASZ> 'P 'AUTHOR*) is equivalent to *DIR {PHYLUM}<HALASZ> AUTHOR*

Documentation of FILDIR, DIR, and DIRECTORY ž can be found in Section 14.3 of the IRM.

Manipulating Files

Basic Lisp File Manipulation Functions

There are a number of functions that allow you to manipulated files in Interlisp, e.g., to copy, move, and delete files.

These functions work on the file *per se* and not on the *information in the file*. Therefore, they apply to all flavors of files.

Each function takes one or two file name arguments. These arguments can *not* contain wildcard characters. They can however, leave off the device/pathname of the full file name. In this case, the connected directory and the DIRECTORIES list will be used as described above. If the version number is not specified, all function assume the HIGHEST version is intended (except DELFILE, which assumes the lowest version).

(DELFILE *FileName*) ž deletes *FileName* (i.e., gets rid of it forever and ever!!!)

Example: (DELFILE ' <HALASZ>LISP>INIT)

(COPYFILE *FromFile ToFile*) ž makes a copy of *FromFile* and places on *ToFile*. If a file named *ToFile* already exists, the copied file becomes the next higher version number.

Example: (COPYFILE ' <HALASZ>LISP>INIT
' {ERIS} <JONES>LISP>INIT)

(RENAMEFILE *OldName NewName*) ž renames the file *OldName* to be *NewName*. If a file named *NewName* already exists, the copied file becomes the next higher version number. On some devices, RENAMEFILE actually does a rename (and hence is relatively fast). On other devices, RENAMEFILE actually does a COPYFILE from *OldName* to *NewName* and then a DELFILE of *OldName* (and hence is relatively slow). RENAMEFILE from one device to another device always does a COPYFILE followed by a DELFILE.

Example: (RENAMEFILE ' {phylum} <halasz>lisp>init;5 ' <halasz>init)

Example: (RENAMEFILE ' {phylum} <halasz>lisp>init
' {eris} <jones>init)

(**SEE *FileName***) [Note: **SEE** is an NLAMBDA function] ž prints *FileName* to the TTY window so that you can examine it. SEE does no formatting, it just dumps it on your screen character by character.

Example: (SEE '{phylum}<halasz>lisp>test)

Result printed in TTY window:

```
69_SEE <HALASZ>LISP>TEST
(FILECREATED "21-Feb-85 15:50:09" {PHYLUM}<HALASZ>LISP>TEST.;3 1096
  changes to: (FILEPKGCOMS ANNOUNCEDFILES)
    (VARS TESTCOMS)
  previous date: "21-Feb-85 15:48:38" {PHYLUM}<HALASZ>LISP>TEST.;2)
**COMMENT**
(PRETTYCOMPRINT TESTCOMS)
(RPAQQ TESTCOMS ((ANNOUNCEDFILES ONE TWO THREE)
  (FILEPKGCOMS ANNOUNCEDFILES)))
(PRINT "Loading ONE")
(FILELOAD ONE)
(PRINT "Loading TWO")
(FILELOAD TWO)
(PRINT "Loading THREE")
(FILELOAD THREE)
(PUTDEF (QUOTE ANNOUNCEDFILES) (QUOTE FILEPKGCOMS) (QUOTE
  ((COM
    MACRO
    (X
    (P
    *
    (FOR File in (QUOTE X)
      join
      (LIST (BQUOTE (PRINT ,
        (CONCAT
          "Loading "
          File)))
        (BQUOTE (FILESLOAD , File))))))
  CONTENTS
  (LAMBDA (COM NAME TYPE)
    (AND (EQ TYPE (QUOTE ANNOUNCEDFILES))
      (SUBSET (INFILECOMTAIL COM)
        (FUNCTION LITATOM))))))
(PUTPROPS TEST COPYRIGHT ("Xerox Corporation" 1985)
(DECLARE: DONTCOPY
(FILEMAP (NIL)))
STOP
NIL
70_
```

(**LISTFILES *FileName1 FileName2 ...***) [Note: **LISTFILES** is an NLAMBDA function] ž prints hardcopies of *FileName1*, *FileName2*, etc. on the DEFAULTPRINTINGHOST. LISTFILES is not strictly independent of file flavors. LISTFILES actually determines what flavor of file each *FileName_i* is and then calls the appropriate hardcopy function for that flavor of file.

COPYFILES LispUsers Package

The COPYFILES LispUsers package makes it easy to copy or move groups of files from one place to another using wildcard facilities like those in DIR and FILDIR.

(COPYFILES *source destination options*) ž Copies the files designated by *source* to the place designated by *destination*. *source* is a pattern such as given to DIRECTORY or DIR; it can also be a list of file names. *destination* is either a directory name, or a file-name pattern, with a 1-1 match of "*"s in *to* to "*"s in *source*. (The number of *'s in each *source* pattern needs to match the number of *'s in each *destination* pattern.) The argument *options* is a (list of) options (if you have only one and its an atom, you can supply it as an atom), as follows:

You can specify whether COPYFILES should ask before each transfer. Default is not to ask.

ASK ž ask each time before moving/copying a file (default is to not ask).

(ASK N) ž Ask, with default to No after DWIMWAIT seconds.

(ASK Y) ž Ask, with default to Yes after DWIMWAIT seconds.

COPYFILES normally uses COPYFILE to create a new file. It also usually only copies the "highest version", and creates a new version at the destination. Alternatively, you can specify any of the following:

RENAME or *MOVE* ž use RENAMEFILE instead of COPYFILE, i.e., the source is deleted afterwards.

ALLVERSIONS ž Copy all versions, and preserve version numbers.

REPLACE ž If a file by the same name exists on the destination, overwrite it (don't create a new version)

After COPYFILES gets done, it can be instructed to delete some files afterward:

PURGE ž This involves a separate pass (afterwards): any file on the *destination* which doesn't have a counterpart on the *source* is deleted.

PURGESOURCE ž converse of PURGE (and used by it): if the file is on the source and not on the destination, delete it.

Examples:

(COPYFILES ' {ERIS} <MASINTER> *.MAIL
' {PHYLUM} <MASINTER> OLD- *.MAIL) will copy the any mail file on {Eris} <Masinter> to {Phylum} <Masinter>, renaming FOO.MAIL to OLD-FOO.MAIL.

(COPYFILES ' {ERIS} <MASINTER> *.MAIL
' {PHYLUM} <MASINTER> OLD- *.MAIL 'RENAME) will use RENAMEFILE instead.

(COPYFILES ' ({DSK} TEST {DSK} WEST) ' {PHYLUM} <MYDIR>) will move the files TEST and WEST from {DSK} to {PHYLUM} <MYDIR>.

COPYFILES ({DSK} *. ; {FLOPPY}) will copy all files on {DSK} with no extension to {FLOPPY}.

(COPYFILES ' {ERIS} <LISPUSERS> ' {PHYLUM} <LISPUSERS>
' (PURGE)) will make {Phylum} <LISPUSERS> look like {ERIS} <LISPUSERS>; bringing over any file that isn't already on Phylum and then deleting the ones that were on {PHYLUM} and aren't on {ERIS} any more.

Still to come about files and directories:

Using the FileBrowser

CHATing to an IFS

Dealing with Devices

Using Floppies on the Dlion

Archiving Files

Opening and closing Files

References

Connected directories are covered in Section 18.16.6 of the IRM.

The DIRECTORIES list is covered in Section 18.16.6 and page 15.20 of the IRM.

DIR, FILDIR and DIRECTORY are covered in Section 14.3 of the IRM.

The file manipulation functions are documented on pages 6.3 and 18.10-11 of the IRM.

COPYFILES is documented on {eris}<lisp>harmony>library>CopyFiles.TEdit.

LispCourse #17: Files and Directories ž Part 3

The FileBrowser Package

The FileBrowser is a window/mouse-based interface to the manipulation of files.

To start a file browser, use the function call: `(FB FileNamePattern)` where *FileNamePattern* is a wildcarded file name of the form allowed by FILDIR and DIR (See LispCourse #16, p. 6).

Example: `(FB '{phylum}<halasz>lisp>*.dcom)` will open a file browser on all of the DCOM files in the `{phylum}<halasz>lisp>` directory. FB will first prompt you for the region in which to place the window.

The result:

Edit		Delete		Undelete	
Expunge		Load		Compile	
Copy		Rename		See	
Update		Hardcopy			
{phylum}<halasz>lisp>*.DCOM;* browser					
Name	Length	Author	Created		
ARCHIVETOOL.DCOM;4	7597	Halasz.PA	6-Feb-85	1:25:13	PST
COPIER.DCOM;1	67283	Halasz.PA	22-Mar-84	15:26:12	PST
FPI.DCOM;4	16065	Halasz.PA	17-Oct-83	20:06:06	PDT
INTERUNIX.DCOM;9	8415	halasz.PA	17-Nov-83	23:06:38	PST
TEDITKEYFIXES.DCOM;2	1874	halasz.PA	14-Feb-85	21:08:23	PST
TOUCH.DCOM;3	1523	Halasz.PA	15-Dec-84	13:51:14	PST
Info Options					
Length	ByteSize	Pages	Type		
Created	Written	Read	Author		

The FileBrowser display consists of the following 5 divisions from top to bottom:

- 1) Prompt window ž the place where the FileBrowser prints messages and gets input from you.
- 2) Command menu ž these are commands that can be applied to the selected file(s).

- 3) Title bar ž shows the FileNamePattern for this FileBrowser.
- 4) Files listing ž a scrollable listing of the names and attributes of files matching FileNamePattern.
- 5) Info Options menu ž a menu of the file attributes to be displayed in the Files listing.

The Files Listing and Selecting Files

The files listing can be scrolled like any window; just roll out the left side of a window to get the scroll bar.

One or more files in the files listing can be selected at any given time. To select a file, point to the left end of any line and click a mouse button. When the cursor nears the left edge of the window, it turns into a right pointing arrow (-->).

The mouse buttons are interpreted as follows:

LEFT ž selects the file and unselects all other files (i.e., makes the file the **ONLY** selection).

MIDDLE ž inverts the selection of the file, i.e., if the file was selected then it unselects the file otherwise it selects the file. All other selections are unaffected. (This button is used for selecting several files that are not contiguous.)

RIGHT ž extends a selection, i.e., selects the file and all files between this file and the nearest selected file. Exceptions: Has no affect if the file is already selected. Has no affect if the chosen file is **BEWTEEN** two already selected files.

Note: The **LEFT** and **MIDDLE** buttons can actually be clicked when the cursor is anywhere in the line. The **RIGHT** button must be clicked when the cursor is at the left edge of the line.

Selecting a file inverts the line:

Edit		Delete		Undelete	
Expunge		Load		Compile	
Copy		Rename		See	
Update		Hardcopy			
{phylum}<halasz>lisp>*.DCOM;* browser					
Name	Length	Author	Created		
ARCHIVETOOL.DCOM;4	7597	Halasz.PA	6-Feb-85	1:25:13	PST
COPIER.DCOM;1	67283	Halasz.PA	22-Mar-84	15:26:12	PST
FPI.DCOM;4	16065	Halasz.PA	17-Oct-83	20:06:06	PDT
INTERUNIX.DCOM;9	8415	halasz.PA	17-Nov-83	23:06:38	PST
TEDITKEYFIXES.DCOM;2	1874	halasz.PA	14-Feb-85	21:08:23	PST
TOUCH.DCOM;3	1523	Halasz.PA	15-Dec-84	13:51:14	PST
Info Options					
Length	ByteSize	Pages	Type		
Created	Written	Read	Author		

The Command Menu

Most of the commands in the command menu carry out an operation on the selected files.

The commands work as follows:

Edit ž Starts up an editor (TEdit or DEdit) on each of the selected files. If the command is selected using the LEFT mouse button, DEdit is called on Lisp symbolic files and TEdit on all other files. If the command is selected using the MIDDLE mouse button, you will be asked to choose the editor to use from a menu.

Delete ž Marks the selected files for deletion but does not actually delete them. An expunge command is required to do the actual deletion. Files marked for deletion have a line drawn thru them.

```
{phylum}<halasz>lisp>*.DCOM;* browser
```

Name	Length	Author	Created
ARCHIVETOOL.DCOM;4	7597	Halasz.PA	6-Feb-85 1:25:13
GOPYER.DCOM;1	87283	Halasz.PA	22 Mar 84 15:26:12
FPI.DCOM;4	16065	Halasz.PA	17-Oct-83 20:06:06
INTERUNIX.DCOM;0	8415	halasz.PA	17 Nov 83 23:06:38
TEDITKEYFIXES.DCOM;2			
	1874	halasz.PA	14-Feb-85 21:08:23
TOUCH.DCOM;3	1523	Halasz.PA	15-Dec-84 13:51:14

Info Options

Undelete ž Removes the deletion marks (if any) from the selected files so that they will not be deleted in the next expunge.

Expunge ž Actually deletes all of the files that have been marked for deletion and then updates the files listing. Not this command ignores the selected files. It operates on the *marked* files instead.

Load ž Applies LOAD to the selected files.

Compile ž Compiles the selected files.

Copy ž Copies the selected files using a COPYFILE for each file. You will be prompted for the destination of the COPYFILES in the FileBrowser's prompt window. If there is a single selected file, you will be asked for a destination *file name*. If there is more than one file selected, you will be asked for a destination *path name* (i.e., a directory). In the latter case, each of the selected files will be copied to the specified directory retaining their old file name.

Rename ž Like Copy, except does a RENAMFILE for each selected file.

See ž For each selected file, opens a separate window (prompting for a region) and prints the contents of the file in the window. The windows are scrollable but not editable.

Update ž Updates the files listing. Updates are necessary if a change is made to the listed directory. Except following the Expunge command, the FileBrowser does not automatically update the files listing when changes are made to the files.

The Update command must be used to insure that the file listing reflects the true state of the files listed.

If this command is selected using the LEFT mouse button, a file listing is made using the current FileNamePattern. If the MIDDLE mouse button is used, you are given the a choice of using the current FileNamePattern or specifying a new FileNamePattern for the file listing. If the new FileNamePattern option is chosen, you will be prompted for the new pattern in the prompt window.

Hardcopy ž Applies LISTFILES (See LispCourse #16, p. 13) to the selected files.

The Info Options Menu

The Info Options menu can be used to set what attributes of a file get displayed in the file listing. Each item in the menu represents one of the attribute of a file (e.g., Length, CreationDate, Author, etc.).

Clicking the mouse in the item inverts the selectedness of the item. Selected items are shaded.

When the file listing is updated (e.g., using the Update command) the line for each file will include the selected attributes.

Example: Only Length attribute selected.

Update		Hardcopy		Archive	
{phylum}<halasz>lisp>*.DCOM;* browser					
Name	Length				
ARCHIVETOOL.DCOM;4	7597				
COPIER.DCOM;1	67283				
FPI.DCOM;4	16065				
INTERUNIX.DCOM;9	8415				
TEDITKEYFIXES.DCOM;2	1874				
TOUCH.DCOM;3	1523				
Info Options					
Length	ByteSize	Pages	Type		
Created	Written	Read	Author		

Using the FileBrowser

You can simultaneously open as many FileBrowsers as you like.

Beware of the order in which files are listed in the FileBrowser. Each device appears in a different order. In particular, you should be careful to observe the order in which the several versions of a file are listed. On some devices, versions will be in ascending order, on other devices in descending order, on still other devices in lexical or even random order.

Dealing with Devices

In general, you deal with files and directories and NOT directly with the devices these files and directories are stored. However, there are several properties of devices that are important to the user:

1. Amount of available space on the device

Each device can store only a limited amount of data. Often you want to know how much of this space is left. The following functions provide that information for various devices:

Local Disk ž the function (DISKFREEPAGES *DeviceName*) returns the number of pages (512 bytes) still available on the named local disk partition.

Example:

```
77_(DISKFREEPAGES 'DSK5)
8341
78_
```

File Servers ž There is no Interlisp interface to finding out the amount of space left on a directory on a file server. For the IFSs, you can CHAT to the IFS and then CONNECT to the directory and issue the DSKSTAT command.

Example:

```
Phylum IFS 1.38.1L Executive ...
@login (user) halasz (password)
@connect (to directory) notecards
@dskstat
35373 pages used out of 40000 in directory <notecards>.
495072 pages used, 28872 left in the system.
@quit
```

Floppy ž To find out the amount of space left on a floppy disk, use the (FLOPPY.FREE.PAGES) function call.

Example:

```
79_(FLOPPY.FREE.PAGES)
345
80_
```

Core Devices ž The concept of space is not relevant here since these are "simulated" devices. In general, there is only about 5000 pages available for COREDEVICE files in an Interlisp virtualmem.

2. Random access versus No random access

Some device allow programs to access the contents of a file in a random order. Other devices allow access to a file in a strict sequential order from start to finish.

Some programs require random access devices. Lafite, in particular, require mail files to be stored on random access devices.

All devices we have discussed allow random access EXCEPT the NS file servers. Therefore many programs including Lafite cannot be used with NS file servers.

3. Location of your virtual memory file

The function call (DISKPARTITION) returns the number (on Dolphin/Dorado) or name (on DLion) of the local disk partition from which the currently running Lisp has been started, i.e., the partition with the currently active virtual memory file.

On Dolphin/Dorados, this will also be the partition number of the {DSK} partition.

Using Floppies on the Dlion

On the whole, floppies on the DLion can be treated like any other device. You can COPYFILE to a from a floppy, RENAMEFILE files on a floppy, etc.

There are some special procedures for dealing with floppies.

The Physical Floppy

Insertion

Write Enable Tab

Floppy Modes

There are three modes for writing on floppies. The modes are defined as follows:

PILOT ž This is the normal mode. In PILOT mode, each floppy can contain many files. No file can be bigger than the amount of information that can fit on one floppy (about 2200 pages).

HUGEPILOT ž In HUGEPILOT mode, there is exactly one file per floppy AND each file can spread across several floppies. HUGEPILOT is generally used for files that are bigger than a single floppy.

SYSOUT ž SYSOUT mode is like HUGEPILOT mode. It is used for Lisp sysouts, which are always large files that take up more than one floppy.

Note: That the floppy mode is a property of each floppy in the sense that any given floppy can be written on in only one of these modes. You cannot write on a floppy in HUGEPILOT mode once and then PILOT mode the next time.

(FLOPPY.MODE *Mode*) is the function to change the current floppy mode. *Mode* must be one of the three modes described above. The value returned is the previous floppy mode. (FLOPPY.MODE) just returns the current floppy mode.

When reading from a floppy (e.g., COPYFILEing from a floppy to another device), the system will detect what mode is appropriate and read the file from the floppy in the correct mode.

When writing to a floppy (e.g., COPYFILEing from a file server to a floppy), you must set the floppy mode as appropriate. If you are in HUGEPILOT or SYSOUT mode, the system will prompt you for another floppy whenever it needs one.

Example: Copying a sysout from phylum to floppy.

```
8_ (FLOPPY.MODE 'PILOT)
```

```
PILOT
```

```
9_ (COPYFILE '{PHYLUM}<halasz>Init '{FLOPPY}INIT)
```

```
{FLOPPY}INIT;1
```

```
10_ (FLOPPY.MODE 'SYSOUT)
```

PILOT

```
11_ (COPYFILE '{phylum}<halasz>new.sysout
      '{floppy}lisp.sysout)
```

Please insert floppy for Lisp Sysout #1

<about 15 min passes>

Please insert floppy for Lisp Sysout #2

<about 15 min passes>

Please insert floppy for Lisp Sysout #3

```
{floppy}lisp.sysout;1
```

```
12_ (FLOPPY.MODE 'PILOT)
```

SYSOUT

Formatting, Compacting, Scavenging

(FLOPPY.FORMAT *Name*) ž Before being used, every new floppy must be formatted. Formatting gives the floppy a name and sets up the floppy so Interlisp can write on it. The *Name* of a floppy is any arbitrary atom.

Formatting a floppy erases the information already on the floppy which can be good or bad, depending on whether you intended to erase the information or not.

(FLOPPY.COMPACT) ž files on a floppy need to take up contiguous space. Sometimes you may have enough free space on your floppy for a file, but the file still won't fit because the free space is not contiguous but spread out between existing files. (FLOPPY.COMPACT) will compact the free space on a floppy so that all free space is contiguous.

For example, if you create three files and then delete the middle one, you will have some discontinuous free space (i.e., the space previously taken up by the deleted file between the two existing files and the free space after the third file). The space between the two files cannot be used by large files. (FLOPPY.COMPACT) will move the last file so that all the free space is in one block at the end of the floppy, thus making it possible for all of the free space to be used by large files.

(FLOPPY.SCAVENGE) ž tries to fix up a floppy that has gotten messed up for some reason or another.

FLOPPY.SCAVENGE will also retrieve files that have been accidentally deleted, providing the space used by those files has not already been overwritten by another file. In particular, if you accidentally delete a file and run FLOPPY.SCAVENGE before writing another files on the disk, you will always get back your deleted files.

Storing and Retrieving Floppy Images

Sometimes its convenient to treat each floppy as one giant file, no matter how many files are actually stored on the floppy. For example, you might want to copy a floppy containing several files to a file server and then make a copy of the floppy from the file server. One procedure would be to do the copies file by file using COPYFILE. A quicker procedure, however, would be to treat the floppies as one giant file and copy that one file back and forth to the file server.

The functions FLOPPY.TO.FILE and FLOPPY.FROM.FILE allow you to treat a whole floppy as a single giant file.

(FLOPPY.TO.FILE *FileName*) ž will copy a whole floppy (treated as a single file) to the file named by *FileName*.

(FLOPPY.FROM.FILE *FileName*) ž will copy the file named by *FileName* to a whole floppy (treated as a single file).

Example:

```
10_ (FLOPPY.TO.FILE '{PHYLUM}<halasz>WHOLEFLOPPY)
    {PHYLUM}<halasz>WHOLEFLOPPY;1
    <<Insert a blank floppy>>
11_ (FLOPPY.FROM.FILE '{PHYLUM}<halasz>WHOLEFLOPPY)
    {PHYLUM}<halasz>WHOLEFLOPPY;1
    <<Insert a blank floppy>>
12_ (FLOPPY.FROM.FILE '{PHYLUM}<halasz>WHOLEFLOPPY)
    {PHYLUM}<halasz>WHOLEFLOPPY;1
```

Documentation

Can be found on *{eris}<lisp>harmony>doc>floppy.tty*.

Opening and Closing Files

Before a program to read from or write to a file, the file must be opened by Interlisp. The file remains open as long as the program is reading from or writing on the file. When the program is finished, it closes the file.

If one program is writing on an open file, no other program can open that file. The second program will get a "File won't open" error when it tries to access the file.

Two programs can read from the same file at the same time.

Ordinarily, opening and closing of files is done automatically. Sometimes, however, a program bombs without properly closing its files. In this case, the files remain open and can prevent other programs from accessing the files. When this happens, the user has to close the open files by hand.

The following function are used for closing files:

(OPENP) ž returns a list of all the open files.

(CLOSEALL) ž closes all open files

(CLOSEF *FullFileName*) ž closes the file specified by *FullFileName*.

Example:

```
10_ (TEDIT '{phylum}<halasz>test.ted)
```

```
File won't open
```

```
{phylum}<halasz>test.ted;1
```

```
11_ (OPENP)
```

```
(({phylum}<halasz>test.ted;1)
```

```
12_ (CLOSEALL)
```

```
(({phylum}<halasz>test.ted;1)
```

```
13_ (TEDIT '{phylum}<halasz>test.ted)
```

```
{process}#6.23456
```

```
14_
```

References

There is no FileBrowser documentation

Floppy documentation is on *{eris}<lisp>harmony>doc>floppy.tty*.

Opening and closing files is covered in Section 6.1 of the IRM

LispCourse #18: Files and Directories ž Part 4

Methods for Dealing with Files Outside of Interlisp

Interlisp provides a nearly uniform interface for dealing with files on a wide variety of devices. Because it attempts to be uniform, it does not take full advantage of the full capabilities of each device.

Many devices allow you to log into the device directly and interact with a specialized Exec for that device. These device Execs give you access to the full power of each device. However, each such Exec is different, depending on the software running on that device.

There is NO access to the following devices outside of Interlisp:

- DLion Local Disk

- Floppies

- Core Devices

There is are non-Interlisp Execs on the following devices:

- IFS file servers (the IFS Exec)

- NS file servers

- Vax-Unix file servers

- Dolphin/Dorado Local Disk (the Alto Exec)

For devices that run as file servers, you can login from within Interlisp by using the CHAT program. For the Dolphin/Dorado Local Disk, you must log out of Lisp and use the Alto Exec.

CHATing to an IFS

CHAT is a package in Interlisp that allows you to open a window that is a virtual terminal to any other device on the network (that supports virtual terminals). This window acts exactly like a standard character-type computer terminal of the type you never see around PARC. From this virtual terminal you can log onto the device and enter commands into its Exec directly without going through Interlisp.

Files stored on an IFS can be manipulated by using CHAT to log into the IFS and then using the file manipulation commands of the IFS executive.

Opening a CHAT window

To log into an IFS, use the CHAT package. To start CHAT, type (**CHAT *Host***) into the Lisp Exec window. *Host* is the name of the IFS you want to login to. If *Host* is not specified, then you will be prompted for a name in the Exec window.

Alternatively, you can choose the *CHAT* entry from the BackgroundMenu. You will then be given a menu with a list of IFS (and other non-IFS) servers. Choose the appropriate name or *Other*, which will prompt you for a Host server (i.e., an IFS server in this case) name in the Prompt window.

CHAT will reuse the most recent CHAT window, if it is available. Otherwise, it will prompt you for a region and open a new Chat window in that region.

Once the window opens, you are "chatting" to the IFS executive in the Chat window. Everything you type in is just forwarded to the IFS and is NOT interpreted by Lisp.

CHAT will automatically log you on to the IFS. If you want to prevent CHAT from logging in (e.g., if you want to log in under someone else's name), start CHAT using (**CHAT *Host* 'NONE**) in the Lisp Exec window.

The IFS executive

The IFS executive has an @ prompt. All commands must be terminated by *<RETURN>*.

Some commands have sub-commands. When the IFS is in sub-command mode (i.e., is expecting a sub-command), then the prompt will be a @@. In sub-command mode, a *<RETURN>* will exit the sub-command mode and execute the current command.

When typing into the IFS exec:

BS ž erases the preceding character

CTRL-W ž deletes a word

DEL ž deletes an entire command or sub-command.

CTRL-C may be used to abort any command.

Typeout will stop whenever the window fills up with text and IFS will wait for you to type any character before continuing. If you type ahead, this feature is disabled.

When typing a command to the IFS exec, you need only type enough of each command to make it uniquely identifiable. For example, the *Directory* command can be specified by typing *Di*, but not by typing just *D* because *D* is not sufficient to distinguish between the *Directory* and *Delete* commands.

The available commands are:

Login (user) *user-name* (password) *password* ž Logs you into IFS. This is necessary before issuing most other commands. Ordinarily, Chat will do this for you automatically.

Logout or Quit ž Logs you out and closes the CHAT window.

Connect (to directory) *directory-name* (password) *password* ž Sets your default directory to be *directory-name*, and gives you owner-like access to it. The *password* may be omitted if *directory-name* is your own directory or one to which you have connect privileges.

Directory (default) *directory-name* ž Sets your connected directory to be *directory-name*, but without changing your access rights (and therefore without requiring a password). Connected directories behave as in Interlisp (see LispCourse #16, p. 1).

List (files) *file-designators* ž Lists the names of all files matching *file-designators*, which is a list of up to 10 file names (separated by spaces), any of which may contain '*'s to denote multiple files. The files matching each *file-designator* are listed in alphabetical order.

If you terminate the last *file-designator* with a comma followed by *<RETURN>* (rather than just *<RETURN>*), IFS enters a sub-command mode (with the @@ prompt) in which you may specify additional information to be printed about each file:

Type	file type and byte size
Size	size in pages
Length	length in bytes
Creation	date of file creation

Write date of last write
Read date of last read
Backup date of last backup
Times times as well as dates
Author creator of file
Protection file protection
Verbose same as Type Size Write Read Author
Everything

Sub-command mode is terminated when you type just RETURN in response to the '@@' prompt..

Delete *file-designator* ž Deletes all files matching *file-designators*, which is a list of up to 10 file names (separated by spaces), any of which may contain '*'s to denote multiple files. The version number defaults to the lowest existing version; to delete all versions, you must end each *file-designator* with '!*'.

IFS prints out each file name, followed by '[Confirm]'. You should respond with 'Y' or <RETURN> to delete the file, or with 'N' or to leave it alone.

If you terminate the last *file-designator* with a comma followed by <RETURN>, IFS enters a sub-command mode (@@ prompt) in which you may request the following additional actions:

Confirm (all deletes automatically) ž IFS will not ask you to confirm deleting each file but will just go ahead and do it.

Keep (# of versions) *number* ž IFS will retain the *number* most recent versions of each file and delete all remaining versions. That is, to delete all but the most recent version of each file, specify 'Keep 1'.

Rename *existing-filename* (to be) *new-filename* ž Changes the name of *existing-filename* to be *new-filename*. If you terminate *new-filename*

with *ESC* rather than *RETURN*, the body of *old-filename* (with directory and version stripped off) will be appended to *new-filename*.

Print (files) *file-designator* or **Press (files) *file-designator*** ž Requests that all Press files matching *file-designator* be sent to your default printing server ('Print' and 'Press' are synonyms).

IFS prints out the name of each file followed by '[Confirm]'; you should respond with 'Y' or <*RETURN*> to print the file, or with 'N' or <*DEL*> to skip over it.

If you terminate the last *file-designator* with a comma followed by <*RETURN*>, IFS enters a sub-command mode (@@ prompt) in which you may specify the following parameters:

Copies *number* ž Specifies the number of copies of each Press document to print.

Server *server-name* ž Specifies the name of the printing server to which the Press files are to be transmitted. This may be either a registered name or an internetwork address of the form '*net#host#*' (don't leave off the trailing '#').

In the absence of any sub-commands, IFS will cause one copy of each Press file to be printed on your default printing server.

Printing request may be examined and canceled with the commands **Show Printing-requests** and **Cancel (printing requests)**, respectively.

Note that *only* Press-format files can be printed; IFS checks that every file is a Press file and will refuse to print any file that is not.

DskStat ž Prints the number of used pages and the maximum allowed in the connected directory, followed by the number of free pages in the system. One IFS page is 1024 words or 2048 characters, which is equivalent to four Alto pages.

Systat ž Shows who is presently using IFS, what service they are accessing (FTP, Telnet, CopyDisk, or Mail), and the name or inter-network address of the machine they are coming from.

Commands dealing with protections

The Chat Executive contains several commands by means of which you may manipulate protections of files and directories.

Change Protection (of files) *file-designators* ž See LispCourse #15, p.14)

Change Directory-Parameters (of directory) *directory-name* ž Changes the information associated with the directory as a whole in the manner specified by the *sub-commands*. Sub-command mode is entered automatically. The directory must be either your own or one to which you are connected.

You may change the default file protection by means of the Read, Write, and Append sub-commands in the same manner as in the Change Protection command. Additionally, you may change the create and connect access using the sub-commands:

Create (access permitted to) *groups*

Connect (access permitted to) *groups*

The 'No' prefix may be applied to these as well as to the others.

You may change your default printing server by means of the sub-command:

Printing-Server *host-name*

Show Directory-Parameters (of directory) *directory-name* ž Displays all information about *directory-name*, and additionally prints some other parameters, such as the disk limit and the owner of a files-only directory, that may be changed only by an IFS administrator.

The Interlisp CHAT program

While CHATing to an IFS (or any other kind of host) you can communicate with the Interlisp CHAT program by clicking the MIDDLE mouse button in the CHAT window. This will bring up a menu with the following options:

Close ž Close this connection. Once the connection is closed, control is handed over to the main tty window. Closes the window.

Suspend ž Same as Close, but always leaves the window open.

New ž Closes the current connection and prompts for a new host to which to open a connection in the same window.

Freeze ž Hold typeout from this Chat window. Bugging the window in any way releases the hold. This is most useful if you want to switch to another, overlapping window and there is typeout in this window that would compete for screen space.

Dribble ž Open a typescript file for this Chat connection (closing any previous dribble file for the window). The user is prompted for a file name; a name of NIL just closes the old dribble file.

Input ž Prompts for a file to take input from. When the end of the file is reached, input reverts to T.

Clear ž Clears the window and resets the simulated terminal to its default state. This is useful if undesired terminal commands have been received from the remote host that place the simulated terminal into a funny state.

The mouse button LEFT, when inside a CHAT window, holds output as long as the button is down.

The following variable control CHATs behavior:

CLOSECHATWINDOWFLG ž If non-NIL, every Chat window is closed on exit. If NIL, the initial setting, then the primary Chat window is not closed. Default value is ????.

CHAT.FONT ž If non-NIL, the font that Chat windows are created with. If CHAT.FONT is NIL, Chat windows are created with (DEFAULTFONT 'DISPLAY) [To be covered in a later class!]. Default value is ????.

DEFAULTCHATHOST ž The host to which CHAT connects when it is called with no HOST argument. Default value is ????.

CHAT.ALLHOSTS ž A list of host names, as uppercase litatoms, that the user desires to Chat to. Chatting to a host not on the list adds it to the list. These names are placed in the menu that the background Chat command prompts with. Default value is ????.

CHAT.DISPLAYTYPE ž The type of display (a number) that Chat should tell the remote host the user is on. If Datamedia emulation is desired, this variable should be set to the number corresponding to the terminal type Datamedia for the remote host. If the remote host does not respond to the terminal type protocol in Pup Telnet, this variable is irrelevant. Default value is 10.

CHAT is documented in Section 20.5 of the IRM.

CHATing to an NS File Server

You can use CHAT to access an NS file server as well. The CHAT package works the same way as in CHATing to an IFS. However, the Exec running on the NS server is very different from the IFS Exec.

See ??? for documentation on the NS Exec.

CHATing to an Vax-Unix File Server

You can use CHAT to access a Vax-Unix file server. The CHAT package works the same way as in CHATing to an IFS.

When CHATing to a Vax-Unix file server, you are interacting with a full-scale computer system equal in complexity to Interlisp. The Unix shell (the Unix term for an Exec) supports much more extensive repertoire of file manipulation capabilities than does Interlisp. Unfortunately, the philosophies are somewhat different, so that it is sometimes hard to manipulate Interlisp files using Unix programs .

See any of a number of Unix books on the market or the Unix programmer's manual for more information on using Unix.

The Alto Exec on Dolphin/Dorado Local Disk

When you boot a Dolphin or Dorado set up for Lisp, you enter into an Exec called the Alto Exec. From this Exec, you type "Lisp" to get into Lisp. When you LOGOUT from Lisp, you are returned to the Alto Exec.

The Alto Exec has a few file manipulation capabilities that you can use to manipulate files on your local disk. Commands include DELETE, RENAME, FILESTAT. There are also separate Alto programs that run under the Alto Exec called Neptune (like the FileBrowser) and FTP (analogous COPYFILE) that can be used to manipulate files.

See the *Alto Users Guide* for documentation on the Alto Exec, Neptune and FTP.

LispCourse #19: Living in the Network World

Basic Concepts

Advantages of Networks

Interlisp-D runs perfectly well in a stand-alone, single machine world.

But, the full power of Interlisp-D comes when your D-machine is attached to a network and can communicate to the other machines on the network.

Networked machines have two basic advantages:

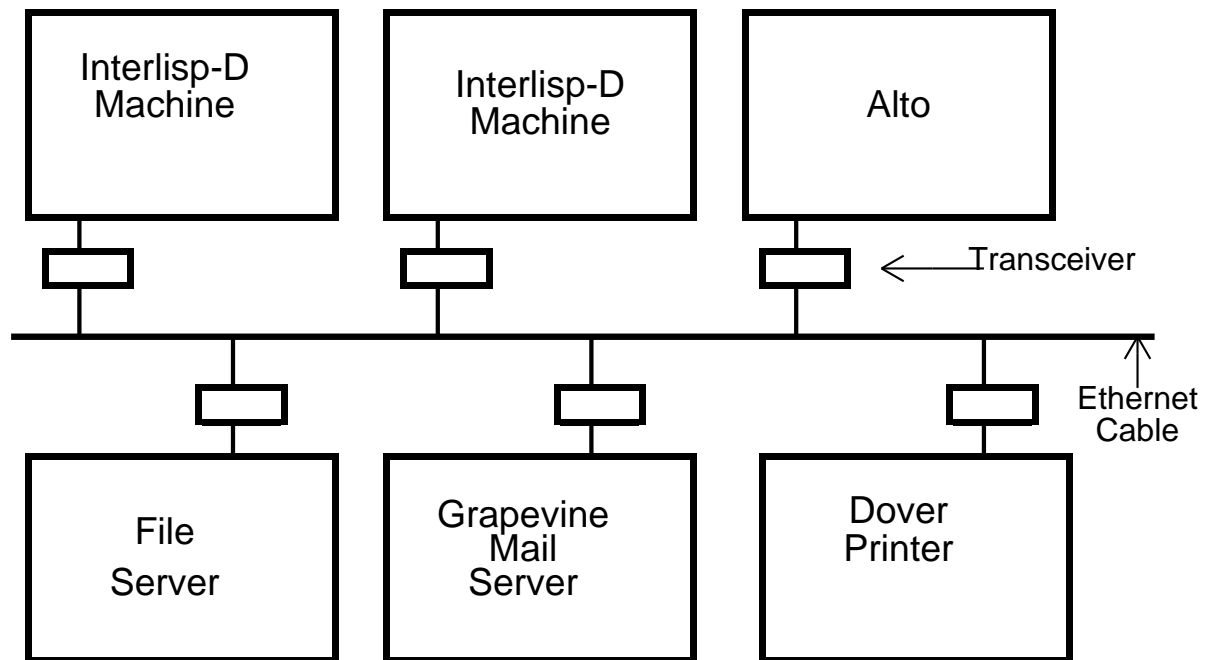
- 1) *Communication* ž you can communicate with other users on other machines on the network. Examples: electronic mail and sharing information by storing a common file on a file server.
- 2) *Sharing of expensive resources* ž many machines can share expensive resources such as printers, disk drives, etc. Without a network, every machine would need its own printer, its own large disk, its own tape drive, etc.

Basic Network Concepts ž Servers and Clients on a Local Network

Xerox's brand of network is called an Ethernet. At the level of this discussion, the Ethernet is pretty much like most of the Local-area networks one can buy to connect together groups of computers.

The Ethernet is simply a coaxial (i.e., cable television) cable. Two or machines can "tap" onto this cable with a special piece of hardware called a *transceiver*.

The various computers on the Ethernet can communicate to each other by sending messages over this cable.



In principle, any two machines on this network can communicate with each other. For example, the two Interlisp-D machines could communicate with each other directly.

In practice, machines on an Ethernet are usually separated into two classes: **servers** and **clients**.

Servers – machines that exist to perform one or more specialized services such as storing files, printing documents, processing mail, etc. These machines sit on the network and wait for a client machine to request some service. Once the request arrives, they perform the service.

Clients – the ordinary user machines. Programs (e.g., Interlisp-D) that the user runs on these machines occasionally need some specialized service (e.g., a file printed, a file stored, mail sent or retrieved) that they request from a server somewhere on the network.

Most communication on a network takes place between clients and servers.

Communication between servers and servers also take places as different servers need to coordinate their services. For example, when I send mail to you, my mail server communicates with your mail server in the process of transferring the mail.

Some typical servers:

Boot Servers ž provide programs necessary to boot machines over the network. For example, provide Othello during 3-boot of the DLion or provide the NetExec while net-booting a Dorado/Dolphin.

Name Servers ž provide translations from names to locations on the network. For example, tells Interlisp-D that PHYLUM is located on network number 6, and is machine number 255 on that network.

Authentication Servers ž provide name and password checking. When a user wants to log into any machine on the network, that machine okays the user's name and password with an authentication server.

File Servers ž provide space to store files.

Mail Servers ž provide electronic mail delivery, distribution lists, etc.

Print Servers ž computer attached to a printer; queues and prints documents.

Specialized Servers ž e.g., dictionary server, dial-in servers, tape servers, etc.

NOTE: A server is a program, i.e., a piece of software. On any given machine, there may be several servers (i.e., programs) running at once. The servers are logically separate, though physically dependent. For example, it is standard to run the Boot server and the Name server on the same machine on a network.

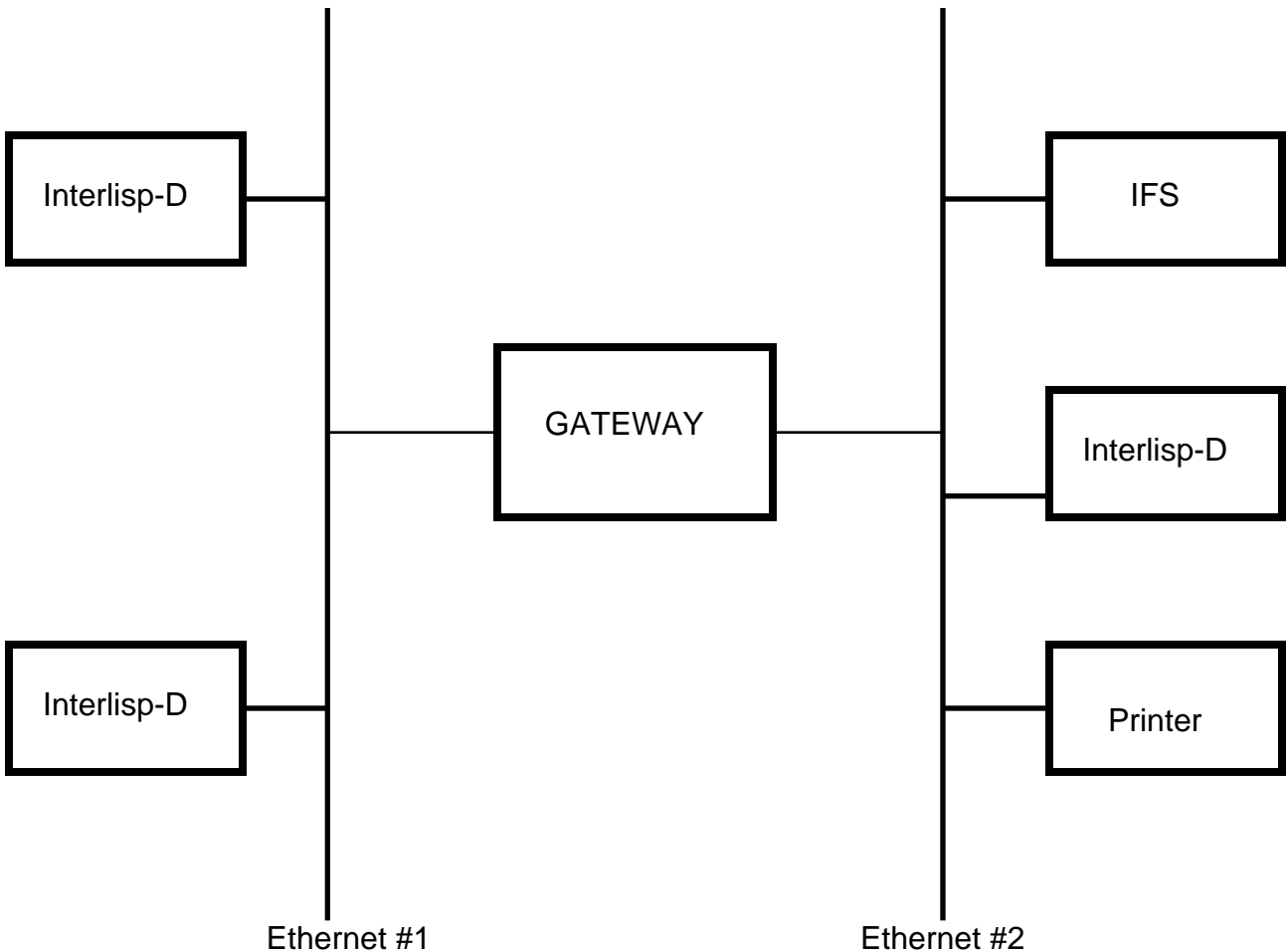
Moreover, on PUP networks (see below), the same Grapevine "server" (read, machine) provides both mail and authentication service.

The Internet: Connecting Together Many Local Networks

A single local network can handle only a limited number of machines that are located within a relatively small distance of each other.

Typically, local Ethernets are connected together to form a larger "network of networks" called an *internet*, as in "*The Xerox Internet*".

Local networks are connected together using *gateways*.



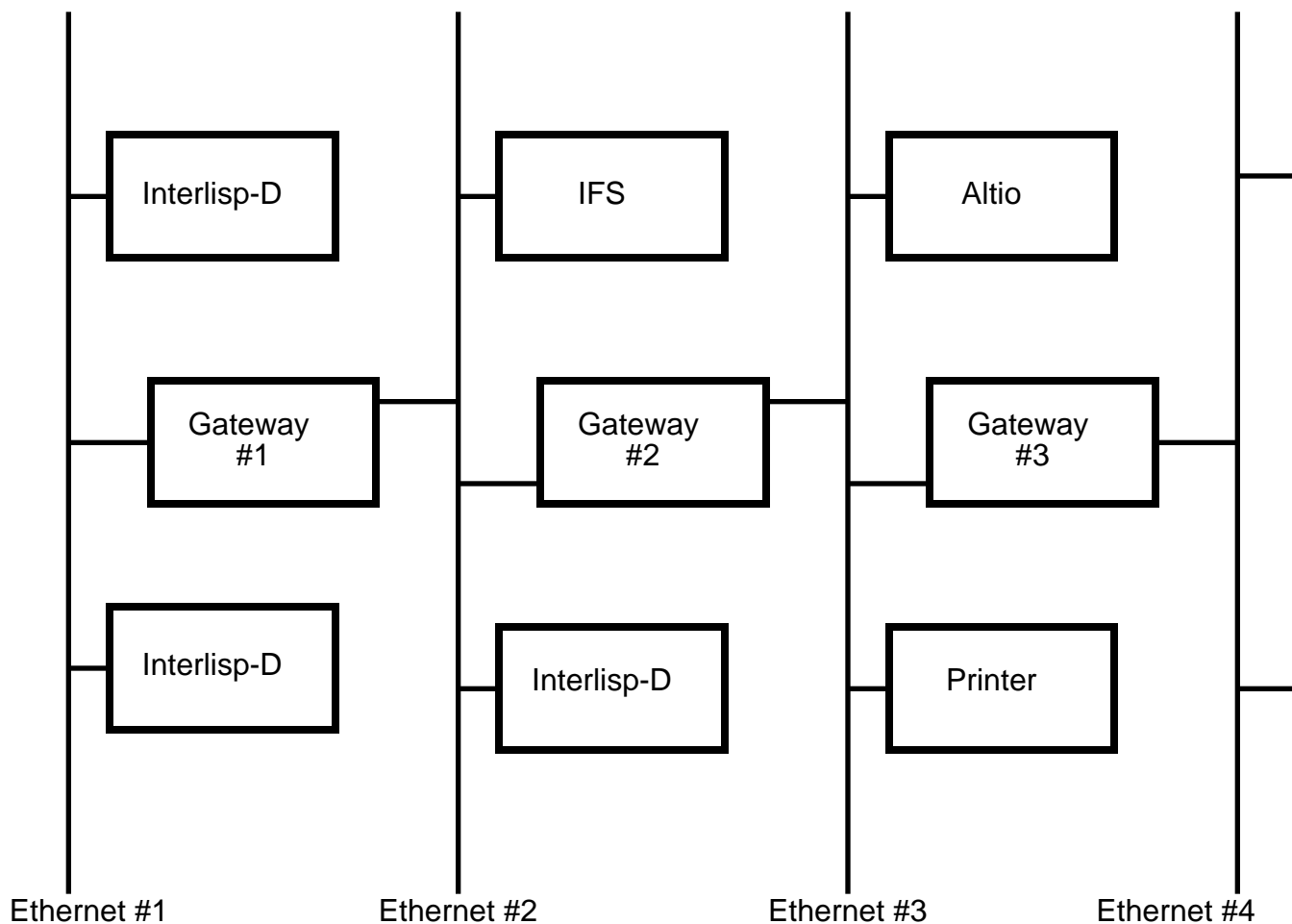
A **gateway** is a computer that is connected to two or more networks. If a machine on one network wants to send a message to a machine on another network, it sends the message to a gateway on its network. The gateway takes the message off of the network of the originating machine and puts it on the network of the destination machine.

In short a gateway transfers message from one local network to another local network. If your machine is on one network and your file server is on another

network, when the gateway goes down you are effectively cut off from your file server.

The Xerox Internet consists of many, many (100??) local Ethernets connected together by gateways. There are maps of this network on ????

In the Xerox Internet not all of the Ethernets can talk directly through a single gateway to all of the other Ethernet. A message may have to pass through several networks and several gateways to get from one machine to another. For example, to get from an Interlisp-D machine on Ethernet#1 to a Printer on Ethernet#3, the message would have to pass from Ethernet#1 through Gateway#1, Ethernet#2, and Gateway#2 to Ethernet#3.



Gateways also do *routing*, i.e. they help figure out what sequence of Ethernets and Gateways will get a message from its source to its destination.

In particular, the source machine sends the message to a gateway on its network. This first gateway transfers the message to a second network and sends the message on to some other gateway on the second network that is also connected to a network that is closer to the destination. This gateway passes the message on to an even closer gateway. This process repeats until the message reaches its destination.

The point is that no gateway needs to "know" the entire route for the message. Each gateway only needs to know what networks are likely to be "closer" to the destination machine than the gateway that gave it the message.

You don't need to know much about gateways. Internetwork transfer and routing is all done for you (the Interlisp-D user) automatically.

But: gateways do go down sometimes and sometimes they get overworked and slow down. In this case, message traffic on the Internet can slow down or stop altogether for some machines.

Example:

The ISL Dorados are on Ethernet #6.

Phylum is also on Ethernet #6.

The ISL DLions are on Ethernet #204.

NewWing is a gateway between Ethernets #6 and #204.

If NewWing goes down or slows down, then the Dorados will be able to get at Phylum with no problem, but the DLion will have problems because their message are not being transfered from net #204 to net #6.

Note: In general the machines that run gateways are the same machines that run the Name server and the Boot server on the Ethernet, although all three are logically separate services that could be run on three different machines.

Physical Ethernets ž 3Mb Experimental versus 10Mb Product

There are two different kinds of physical Ethernets: the 3Mb Experimental Ethernet and the 10Mb Product Ethernet. The two are distinguished by the transceiver hardware and the type of coaxial cable used. They also transmit at different speeds - 3 million bits per second versus 10 million bits per second.

Most machines have the hardware to support either the 3Mb Ethernet or the 10Mb Ethernet, but not both.

In general DLions and all product equipment uses the 10Mb Product Ethernet. The Altos (including all IFSs and many mail servers), Dolphins, and Dorados use the 3Mb Experimental Ethernet (i.e., the original PARC invention).

There are NO logical or functional differences between the two Ethernets. In the Xerox Internet, 3Mb and 10Mb Ethernets are mixed together. There are gateways containing both 3Mb and 10Mb Ethernet hardware that transfer messages between 3Mb networks and 10Mb networks.

The bottom line: if you have both a Dorado and a Dlion, they can't be on the same network. You'll need two separate Ethernets running through your office, one 10 and one 3. The two machines can speak to each other, but only through a gateway that connects 3 and 10 Mb networks. (Special note: you can actually but Dorados with 10Mb hardware, but we don't have any in ISL.)

Communicating on an Ethernet ž Packets and Protocols (PUP versus NS)

Packets

The Ethernet is a packet network: each message between two machines is broken down into short **packets** of information. These packets are then sent out over the Ethernet one after the other until the entire message has been transmitted.

The transmission of a packet between two machines is actually a "conversation" between two the two machines. The source machine sends out a packet. The source machine then waits before sending out the next packet until it recieves some sort of acknowledgement. If the destination machine receives the packet correctly, then it sends an "okay" acknowledgment packet to the source machine. If it receives the packet with some sort of error, it sends back an error acknowledgment packet. If

the source receives an error acknowledgment , it resends the packet, otherwise it goes on to the next packet.

Several "conversations" can be going on at once on an Ethernet. While one computer is preparing the next packet to be sent, another computer can be actually sending its packet on the cable. Thus, the various machines on an Ethernet share (technically, *multiplex*) the available time on the Ethernet.

Protocols (PUP versus NS (versus TCP/IP))

For two machines to carry on a conversation over a network, they must see eye-to-eye on a few matters. In particular, they must agree on the format of each packet and they must agree on the exact meaning of each of the message transmitted.

There are standards, called **protocols**, that specify these two things.

Level One Protocols

The standards for the format of each packet are called **level-one protocols**.

There are *three* different level-one protocols in use within Xerox:

the PUP protocol ž the original PARC protocol

the NS protocol ž the product protocol

the TCP/IP protocol ž the Arpanet and university-favorite protocol

Each of these protocols specifies a slightly different format for each packet of information sent across the network. The major difference is in how you specify the network and machine numbers.

Most systems can speak (and understand) only one of the protocols. For example, Star and the NS file servers speak only NS. Cedar, the IFSs and the Grapevine speak only PUP protocols. Only the Vax speaks TCP/IP.

This means that a Star cannot speak to an IFS or to a Grapevine mail server because the two cannot agree on the format of the Ethernet packets.

Interlisp-D, however, can speak all three level-one protocols (TCP/IP will be available in Intermezzo). Therefore, Interlisp-D can exchange packets with Star and the NS file servers AND with the IFSs and Grapevine mail servers AND with the Vaxes running TCP/IP.

[Note: Some gateways handle only NS packets. I have been stuck in the situation where there are two machines that both speak PUP and were physically connected together by a sequence of gatewayst. But the two machines could not communicate because one of the gateways connecting the two machines could not handle PUP packets. Ugh!!!]

Higher Level Protocols

Agreeing on a packet format is not enough. To communicate, two machines must agree on the set of possible messages and the exact meaning of these messages.

For example, if one machine wants to ship a file to another machine, the two machines must agree on the exact set of messages to exchange in order to get this done: how to specify the file name on the destination machine, how to encode the data in the file, how to signal errors in the received packets, how to notify the destination machine that the file transmission is complete, and so on.

The standards for the exchange of these "task-oriented" messages are known as **higher-level protocols**.

There are a multitude of higher level protocols:

file transfer protocols

mail protocols

telnet (CHAT) protocols

leaf (page level file access) protocols

and many, many more

Each of these protocols specifies the set of messages that need to be interchanged between two machines to carry out some specific task like transferring a file, delivering mail, opening a CHAT connection, etc.

Example:

An file transfer protocol might have the following messages:

Here's the file name to call the file: *Name*

Here's the next 100 bytes of the file: *Data*

Here's bytes N thru M of the file: *Data*

File transmission is done.

Bytes N thru M received okay (in error).

File name not allowed.

...

The bottom line: if two machines speak the same task-oriented protocol, they can effectively communicate in carrying out a task, e.g., in transferring a file. If they don't speak the same task-oriented protocol, there is no way they could communicate well enough to carry out the task.

The PUP World versus the NS World

In principle, the same higher-level protocol could be carried out using either NS or PUP level-one packets.

In practice, the PUP-based higher-level protocols are entirely different from the NS-based higher level protocols. Basically, when PUP was redesigned into NS, all of the PUP-based higher-level protocols were totally redesigned as well.

For example, the file transfer protocol for PUP-based file servers (i.e., FTP protocol) is entirely

different from the file transfer protocol for NS file servers (the NS filing protocol).

Star can't speak to an IFS because it can't produce PUP packets AND it doesn't support the FTP protocol.

Interlisp-D exists in both worlds (in fact in three worlds because there are also TCP/IP higher level protocols). From Interlisp-D you can access both IFSs (using the FTP protocol) and the NS file servers (using the NS filing protocol).

The PUP World

Addresses

In the PUP world, every machine is uniquely identified (i.e., addressed) by an 8-bit (0 to 255) network number and an 8-bit machine number. The complete address of the machine is the network number and the machine number, each followed by a "#".

Examples:

Phylum's address is *6#225#*

My Dorado is *6#60#*

The pool DLion in ISL is *204#36#*

QUAKE is *6#357#*

EXPRESSO is *64#154#*

Certain programs require you to specify a destination machine by its PUP address; TeleRaid, for example.

Interlisp-D also allows you to specify a PUP address in place of a device name for any device reference. For example, instead of (*CHAT 'PHYLUM*), you can use (*CHAT '6#225#*). Instead of *DIR {PHYLUM}<HALASZ>*, you can use *DIR {6#225#}<halasz>*. These addresses can come in handy when the name server is down.

Name Service

Most machines in the PUP world are given a literal for a name; this is true of all servers and most client machines. Thus the name of my machine is Halasz, the name of machine 6#225 is Phylum, etc. Note: some DLions in ISL do not have name because they ran out of space in the name tables.

The name servers running on the PUP-gateways keep a table that translates these names into PUP addresses. Whenever you refer to a machine by name [as in (CHAT 'PHLYUM)], Interlisp-D consults the name server for the address corresponding to that name.

In order to get to a particular type of server in the PUP-world, you have to know the name or PUP address of the particular machine or the machine running the server.

The PUP name servers cannot answer interesting questions like *"What machines are on net 36"* or *"What IFSs do you know about?"*.

In Interlisp:

(ETHERHOSTNAME *PupAddress*) ž returns the name of the machine whose address is *PupAddress*. (ETHERHOSTNAME '6#225#) returns *Phylum*.

(PORTSTRING (ETHERHOSTNUMBER *Name*)) ž returns the PUP address of the machine specified by *Name*. (PORTSTRING (ETHERHOSTNUMBER 'Halasz)) returns *6#60#*.

Authentication Service

In the PUP world, users are registered using the Grapevine mail system.

All users are registered on a Grapevine mail server in their location (called a *registry*). A user's Grapevine name is usually his last name followed by the registry where he works. For example, my Grapevine name is *Halasz.pa* since *.pa* is the Palo Alto registry. Other registries are:

- .wbst ž Webster, N.Y.
- .pasa ž Pasadena
- .es ž El Segundo
- .sthq ž Stamford Headquarters
- .henr ž Henrietta, N.Y.

.dlos ž Dallas

???

When a user logs onto any machine on the network, that machine asks the nearest Grapevine server to authenticate the user and her password.

Because all Grapevine mail servers are in constant communication, you can log into a machine anywhere on the Xerox Internet. That machine will query the Grapevine server nearest to it, which in turn will query your Grapevine server here at PARC. If you typed in the right password, the PARC Grapevine will say okay to the remote Grapevine server which will say okay to the machine you are logging in to.

The Grapevine is a very limited database of all of the users on the Xerox PUP-based Internet. The Grapevine system maintains a list of all the user's on the Xerox Internet. However, it keeps only the user's name and no other interesting information such as office number or phone number or position in the company.

Moreover, you can't even access the list of users in any very interesting way. For example, you can't ask "what users are there in Palo Alto whose last name is Jackson?" or even "Who works at PARC?".

(This contrasts with the NS world, where more interesting queries are possible).

Mail Service

Mail in the PUP world is also handled by the Grapevine system. The recipient of a mail message is specified by his or her Grapevine user name (e.g., Halasz.pa).

When you send mail in Interlisp-D, Lafite sends the mail to the nearest Grapevine server which then distributes the mail to the Grapevine servers of each of the recipients of the mail. The Grapevine system does the work of figuring out the relevant Grapevine server for each of the recipients.

When you retrieve mail, Lafite queries the Grapevine servers for your registry to see if there is any mail waiting for you. If so, it copies the mail from the Grapevine server to your Lafite mailbox.

In addition to delivering and storing mail, the Grapevine provides for the maintenance of a set of distribution lists. Each distribution list consists of a name ending in an "^" (e.g., LispUsers^), a registry (as in LispUsers^.pa), and a list of members of that list. Sending mail addressed to a distribution list will cause the mail to be sent to all of the members of the distribution list.

You can log into the Grapevine and alter various aspects of distribution lists. For example, you can add or remove yourself from a distribution list, see who is on a distribution list, etc.

To do this from Interlisp, LOAD the library package *{eris}<lisp>harmony>library>maintain.dcom*. When the LOAD is finished, the function call (*MAINTAIN*) will log you into the Grapevine and start up a Grapevine "exec" in your TTY window.

From the Grapevine "exec" you can execute a number of commands to query or change distribution lists.

Example:

When you are finished with your interaction with Grapevine, type "Quit" to the exec. This will return you to the Lisp Exec in the TTY window.

For full documentation of the Grapevine "exec", see the document *{indigo}<laurel>maintain.press*.

File Service, Print Service

In the PUP-world, file service is provided by the oft-discussed IFSs.

In the PUP-world, print service is provided by the Dover printers such as Quake and Espresso and the full-press printers such as Jedi and RockNRoll.

Before a file can be printed, it must be translated into a format that the printer can understand. Printers in the PUP-world print PRESS format files, in contrast the NS-worlrd printers which print InterPress format files. While these file formats have nothing to do with PUP and NS in principle, in practice Press versus Interpress follows the PUP versus NS distinction.

Still to Come on Networks

The NS World

Cross-overs between PUP and NS worlds

LispCourse #20 : Living in the Network World ž Part 2

Completions and Corrections

FTP Server package

There is a Lisp Library package that allows a Lisp machine to act as a server for PUP FTP.

Load {eris}<lisp>harmony>library>ftpserver.dcom and run eval the function (FTPSEVER). This will start up a FTP server process that runs in the background.

Once the server is running you can copy files back and forth to your machine's local disk from another machine running Interlisp (or running Alto FTP).

For example, if I start up FTPSEVER on my machine, I can go over to ISLPool DLion and do a (COPYFILE '{DSK}<LISPFILES>INIT '{HALASZ}INIT) to copy an INIT file from the DLion local disk to the local disk on my machine (i.e., the Dorado named Halasz). I could also have referred to my machine by number as in (COPYFILE '{DSK}<LISPFILES>INIT '{6#60#}INIT).

Copying works both ways: (COPYFILE '{HALASZ}INIT '{DSK}<LISPFILES>INIT) would work just as well.

You can also do a DIR and run the FILEBROWSER to a machine running the Interlisp FTP sever. For example, if my Dorado is running FTPSEVER then I can do a (FB {HALASZ}) or (FB {6#60#}) from the ISL Dlion to get a listing of the files on my Dorado.

Documentation is on {eris}<lisp>harmony>library>ftpserver.tedit (& .press).

Note on the use of the word *host*.

In the Interlisp documentation, the word "*host*" is often used to refer to a machine. A *host name* is what we refer to here as a *machine name*. For example, Phylum is referred to as a host and "Phylum" as a host name. The distinction between machines and hosts is non-existent.

The NS World

Machine Addresses

Machine addresses seem to be less important (from the user's point of view) in the NS world than in the PUP world.

Each machine on the NS network has an address: a 32-bit (i.e., between 0 and a very big number) network number and a 48-bit machine number. As in the PUP world, both the network number and the machine number are followed by a "#". When printed out, the machine number prints as three 16-bit quantities separated by periods.

To find the address of your machine, evaluate the variable `\MY.NSADDRESS`. To find the address of some other machine from its name (see below for NS name conventions) use the function `LOOKUP.NS.SERVER` as in `(LOOKUP.NS.SERVER 'Phylex:)`, which returns `204#0.125000.20217#`.

Examples:

Phylex:'s address is

StarFile:'s address is `131#0.125000.24314#`

PaperMate:'s address is `142#0.125000.12122#`

ISLPoolDLion's address is `142#0.125000.32462#`

Halasz's Dorado address is `6#0.52612.100312#`

Note: The NS network number is, by convention, the same as the PUP network number for the same physical network. For example: the ISLPoolDLion is known as `204#36#` in the PUP world and `204#0.125000.32462#` in the NS world.

Moreover, Phylex: is on the same physical network (i.e., cable) as the ISLPoolDLion. Phylex: is known as `204#0.125000.20217#` in the NS world. Phylex: has no PUP address because it does not speak PUP.

NS Names: Machines and people are part of a common name space

Basic Concepts: Objects, Domains, and Organizations

In the PUP world, the naming of machines is separate from the naming of people (or groups of people). Machines are named using a litatom

processed by a Name server. People are named using a Grapevine name and password processed by a Grapevine authentication server.

For example:

PHYLUM and *ERIS* refer to machines

Halasz.pa and *Feuerman.pasa* refer to people

LispUsers^.pa and *NoteCardsInfo^.pasa* refer to groups of people.

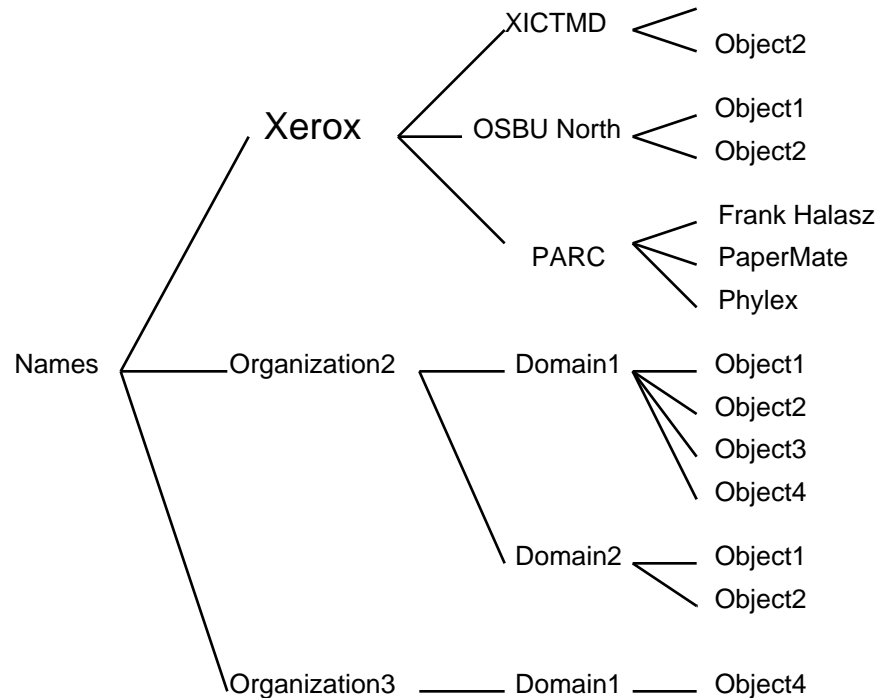
In the NS world all objects, including machines, people, and groups, are named in the same way, i.e., according to the NS naming standard.

The NS naming standard is hierarchical and works as follows:

Every object has an *object name*. An object is a machine, a person, a group, or any other "thing" in the NS world.

Every object also belongs to a particular *domain*, which has a *domain name*. There are always many objects within a given domain.

Every domain belongs to an *organization*, which has an *organization name*. There are generally many domains within an organization.



An *organization* is intended to refer to a single company or major division within a company. For example: all of Xerox is considered a single organization.

A *domain* is intended to be a small division of an organization that shares common resources such as file servers and printers. A domain might be anywhere from large facilities like PARC (with 160 people) to small projects like WBST129UL (with 25 people).

Unlike Grapevine registries (e.g., *.pa*), a domain is not intended to specifically refer to geographic location. Often, however, domains are used to divide an organization into organizational/geographic groups.

Examples from within Xerox:

PARC is a domain

OSBU North is one domain.

OSBU South is another domain.

OSBU Bayshore is a third domain.

The Webster, N.Y. facility has 20 or so different domains corresponding to different organizational groups located in

Webster. Examples: *WBST128*, *WBST129*, *WBST129UL*, and *WRC*.

NS Names and Aliases; Properties of Objects

An object's NS name consists of an object name, a domain name, and an organization name separated by ":"s.

Examples:

Frank Halasz:PARC:Xerox

Phylex:PARC:Xerox

Pluto:OSBU North:Xerox

Bill Liles:XSIS:Xerox

Within a domain, an object may have several alternative object names known as *aliases*. An object's alias can always be used in place of its real object name to refer to the object.

Examples:

Within PARC:Xerox

Halasz: is an alias for *Frank Halasz:*

aifile: is an alias for *Phylex:*

print: is an alias of *LispPrint:*

Thus:

Frank Halasz:PARC:Xerox and

Halasz:PARC:Xerox refer to the same object. So

do *Phylex:PARC:Xerox* and *aifile:PARC:Xerox*

Every named object in the NS world has a bunch of properties that describe the object. For example, every object name has an address, either a NS network address or a NS mail server address. For the most part these properties are not accessible to the ordinary user.

However, every object has a property that tells what type of object it is. For example: people have a property called USER. FileServers have a property called FILE.SERVICE.

The possible properties that define types of objects are:

USER ž a person

PRINT.SERVICE ž a print server

FILE.SERVICE ž a file server

MEMBERS ž a distribution list

These properties come in handy when you want to know all the file servers in PARC or all the people in the OSBU North domain. (See below for how to do these queries).

Default Domain and Organization

In Lisp, you can leave the domain and organization parts off of any NS name. If you do so, the values of the atoms **CH.DEFAULT.DOMAIN** and **CH.DEFAULT.ORGANIZATION** will be used for the domain and organization parts of a name wherever necessary.

Usually, CH.DEFAULT.DOMAIN and CH.DEFAULT.ORGANIZATION are set in your system INIT file. You can change them in your personal INIT file or by SETQing them in the Lisp Exec if you refer to domains and/or organizations different than the ones set in the system INIT file.

At Parc, CH.DEFAULT.DOMAIN is set to *PARC* and CH.DEFAULT.ORGANIZATION is set to *Xerox*.

Examples: Given the standard settings for PARC,

Phylex: refers to *Phylex:PARC:Xerox*

StarFile: refers to *StarFile:PARC:Xerox*

Tundra:OSBU North: refers to *Tundra:OSBU North:Xerox*

Clearinghouses: Name and Authentication Servers for NS World

Introduction

In the NS world, a server called a *Clearinghouse* carries out the functions of both the PUP Name server (machine name to machine address translation) and the Grapevine authentication service (checking peoples names/passwords and distribution list maintenance).

Every domain has one or more Clearinghouses.

Every Clearinghouse knows the names and addresses of the Clearinghouses for every domain and organization on the NS internet.

Every so often (each night or every other night), all the Clearinghouses on the network exchange information about who they are and what domains they cover, etc. Thus, every Clearinghouse has reasonably up-to-date information about all other Clearinghouses.

Clearinghouse Services

Name Service

Every time you type in an NS name, your D-machine interrogates its Clearinghouse server for the translation for the name. For example, if you type in *DIR {Phylex:Parc:Xerox}<halasz>**, your machine will interrogate its Clearinghouse for the address of Phylex:Parc:Xerox.

If the named object is not in the same domain as your Clearinghouse, then your Clearinghouse tells your machine the appropriate Clearinghouse to contact for the domain of the object. Your machine then queries that Clearinghouse to get the address of the named object.

Authentication Service

Similarly, if you attempt to login to a NS file server (which is done every time you access it, even for a DIR), the file server will ask its Clearinghouse about your NS name and password. If you are registered on that Clearinghouse, you will be okayed. Otherwise, that Clearinghouse will give the file server the location of the Clearinghouse for your domain and the file server will check with your Clearinghouse.

For example, if I log into *Tundra:Osbu North:Xerox*, Tundra gets the location of the PARC Clearinghouse (since my name is *Halasz:PARC:Xerox*) from the Osbu North Clearinghouse. Tundra then checks the name and password with the PARC Clearinghouse. If I am registered on the PARC Clearinghouse, then I am allowed to login on Tundra.

Note: The NS world and the PUP/Grapevine world are totally separate. You must be registered on a Clearinghouse somewhere to access NS servers of any type. A Grapevine user name and password is not enough!!

Distribution Lists

Clearinghouses also maintain distribution lists for the NS mail system. Unfortunately, there is no interface like MAINTAIN whereby normal users can make changes to these distribution lists. All changes to the lists on a given Clearinghouse must be done by the administrator of that Clearinghouse. (At PARC, Carol Lehner is the Clearinghouse administrator).

Your Clearinghouse and the set of known Clearinghouses

Your Clearinghouse

The previous descriptions refer to "your Clearinghouse". In the NS world, each D-machine has to discover for itself the location of its Clearinghouse (i.e., the nearest Clearinghouse that is up). This works as follows:

The first time you refer to an NS object (e.g., an NS file server or an NS printer) your D-machine goes out and tries to find the nearest Clearinghouse server to it. It does this by looking around in turn on each of the various networks it knows about until it finds a Clearinghouse.

While its looking, you will see messages of the form *Looking for Clearinghouse servers on net 132* appearing in your Prompt window. Each such message represents one net that your machine is looking for Clearinghouses on. When a Clearinghouse has been found, it prints out something like *Noting Clearinghouse 131#0.125000.77654* in the Prompt window.

The parameter **CH.NET.HINT** determines the network on which your machine begins its search for a Clearinghouse. It should be set in your system or personal INIT files. If CH.NET.HINT is set to correct network number, the search for a Clearinghouse will be efficient (providing your Clearinghouse is up). If CH.NET.HINT is incorrectly set, your machine may look on many, many nets before it finds a Clearinghouse. At PARC, CH.NET.HINT should be set to 89.

The function call (**START.CLEARINGHOUSE T**) will carry out a search for the nearest Clearinghouse and set the appropriate variables to make it "your" Clearinghouse. this function is done automatically for you whenever you make your first NS reference. However, if your Clearinghouse goes down you may have to manually reinitiate the search using **START.CLEARINGHOUSE**.

The set of known Clearinghouses

Every time you refer to an NS domain other than your own, your machine asks your Clearinghouse for the location of the Clearinghouse for that domain. It then tries to make contact with that Clearinghouse.

While contact is being made a message like *Finding Clearinghouse server for WBST129* is displayed in your Prompt window. When it finds the Clearinghouse the

message *Noting Clearinghouse 204#0.125000.23452* will be printed.

Your machine then remembers the location of that domain/Clearinghouse pair so that the next time you access the domain, it will know what Clearinghouse to contact.

Example: If I type *DIR {Tundra:OSBU% North}*, my machine firsts makes contact with the OSBU North Clearinghouse printing the appropriate messages in the Prompt window. It then queries that Clearinghouse for the address of Tundra.

The function call (**SHOW.CLEARINGHOUSE**) will print a graph of all of the Clearinghouses that your machine knows about. It prompts for a region for the graph window.

Accessing the Clearinghouse Information from Interlisp

Interlisp allows provides several functions that allow you to query the information on the Clearinghouse network. These function allow you, for example, to find the names of all the NS print servers in PARC.

The functions are the following:

(**CH.LIST.ORGANIZATIONS** *ObjectPattern*) ž returns a list of all of the organizations on the NS Internet whose organization name matches the *OrganizationPattern*. *OrganizationPattern* can include the * wildcard character, standing for 0 or more of any characters. If *OrganizationPattern* is NIL or just *, then the list returned will contain all of the organizations known.

Examples:

```
10_(CH.LIST.ORGANIZATIONS)
("..." "a long organization." "aflc" "AIL" "airgate" "Allen-Bradley Co." "Boeing" "Comm Test" "Conserve" "C`Servers" "CTMD" "Demo" "DPI" "Earth" "Fuji Xerox" "G/C" "GDP" "Leesburg" "Lou OPD" "MANKKAA"
```

```
"MATRA" "NASA/JSC" "Nash" "Ois-l" "org" "R&D" "Rx"
"RX IT" "RX Sweden" "RX-N" "RX/Denmark" "Rxa"
"RXCH" "RXDK" "RXEG" "RXF" "RXFINLAND" "Rxg"
"rxhh" "rxhol" "rxihq" "Rxihq Tsd" "rxn" "RXS" "RXSF"
"RXSweden" "rxtsd" "RXUK" "Sanyo Kiko" "SBD"
"SIEMENS" "SiemensAG" "SMD" "TA" "veroh" "Versatec"
"Xci" "XCSS" "Xerox" "Xerox OSD" "Xerox Visitors"
"Xopd" "XOS" "xosm") 11_(CH.LIST.ORGANIZATIONS
'Xerox*]
("Xerox" "Xerox OSD" "Xerox Visitors")
```

(CH.LIST.DOMAINS *DomainPattern*) ž returns a list of all of the domains whose domain name matches the *DomainPattern*.

DomainPattern consists of the domain and organization parts of an NS name. The domain part (but not the organization part) can include the * wildcard character, standing for 0 or more of any characters. If the organization part is left off, the default organization (i.e., the value of CH.DEFAULT.ORGANIZATION) is used. If *DomainPattern* is NIL or just *, then the list returned will be a list of all domains in the default organization.

Examples:

```
20_(CH.LIST.DOMAINS "*:")
("A&E1" "A&E2" "A&E3" "AAAI" "Albany"
"Albuquerque" "AlphaMesa" "Alphaservices-Es"
"Alphaservices-Pa" "AlphaServices-PTL"
"Alphaservices-Rx" "alphaservoices-rx" "ATL-
ECE" "Atlanta" "Austin" "BIRMINGHAM" "Bones"
"Brookriver" "CIN OPS" "CrashTest" "ctmd"
"dagobah" "DallasInfomart" "DEMO ROOM"
"detroit" "DlosCE" "DlosEtron" "DLOSL200"
"DlosL300" "DlosLC" "DlosLV" "DlosLV-Comm"
"DlosMBT" "DlosNSC" "DlosSkeff" "Ece-Hq" "ECE-
WASH" "ecewest" "EDGE-Net" "EDNPS Test" "E1
Segundo" "el`e,c`at`u`o`u" "Es A&E1" "ES A&E1 CHS"
"ES A&E2" "ES A&E3" "ES EDNPS Test" "ES GSD"
"ES GSD/WCO" "eS M1" "ES M4" "ES PSI Test" "ES
XC OST" "ES XC Pri" "ES XC XDMS" "ES XC16"
"Evaluation" "FAX-TEST" "GGW-Test" "Grnch"
"Harb" "HENR801A" "HENR801B" "HENR801C"
"HENR801E" "Henr801F" "HENR868" "Houston" "HUB
TEST" "HWDENGdlos" "ISD-NAM" "lac" "LAS VEGAS"
"loop ECE" "LSBG-ECE" "M1 ES" "Minneapolis
Demo" "MWEAOC" "mwwaoc" "NCRHQ" "NEFSO2" "NER-
```

```
OSM" "NetA" "NetB" "ns/csc" "NSC" "NSC-5.0"
"OGC" "OPD-ENG" "OPD-HQ" "OPD-IS" "OPD-LE"
"OPD-MFG" "OPD-MPO" "OPD-MS" "orange" "OS
Service" "osba" "Osbu Bayshore" "OSBU North"
"OSBU South" "OSD Associates" "OSM-OKC" "OSM-
TRG" "OSMDB" "Parc" "Parc Place" "Phoenix"
"PQAnet1" "PQAnet2" "PQAnet3" "PR" "ProdTest"
"ProductTest" "Psd-Executive" "PSD5700"
"Rainbow" "Raisin" "roch" "ROCH041" "Roch805"
"Roch805t" "ROCH853" "Roch888" "ROCH892"
"RochECE" "SalmonDomain" "San Antonio" "Santa
Fe" "South" "STHQ" "Test Base" "Test Net"
"TestServices" "TOMF" "TransientRoutes" "TSC"
"TYSON" "Vista" "Walnut Creek" "waoc" "WBST MFG
HUB" "WBST102A" "wbst102t" "wbst102tchs"
"WBST105" "Wbst105B" "WBST114" "Wbst128"
"WBST129" "WBST129UL" "WBST200" "WBST200LL"
"WBST200UL" "WBST205LL" "wbst205t" "WBST205UL"
"WBST207" "Wbst207ul" "WBST208" "WBST212"
"WBST218LL" "WBST218UL" "WBST223" "WBST223CHS2"
"WBST300" "WBST304" "WBST311" "WBST311C"
"wbst311J" "WBST311X" "WBST845" "WRC" "xais-
demo" "xc es" "XNS-DLBK" "XNS-MAR" "XOS WR"
"XOS-MAR" "xos-mwr" "XOS-OPS" "XOS-SCBC" "XOS-
SR" "XRCC" "Xsis" "XSYS North" "Xsis-Ai" "XSYS-
HQ")
```

```
21_(CH.LIST.DOMAINS "P*C:Xerox"]
("Parc")
```

```
22_(CH.LIST.DOMAINS "WBST*:")
("WBST MFG HUB" "WBST102A" "wbst102t"
"wbst102tchs" "WBST105" "Wbst105B" "WBST114"
"Wbst128" "WBST129" "WBST129UL" "WBST200"
"WBST200LL" "WBST200UL" "WBST205LL" "wbst205t"
"WBST205UL" "WBST207" "Wbst207ul" "WBST208"
"WBST212" "WBST218LL" "WBST218UL" "WBST223"
"WBST223CHS2" "WBST300" "WBST304" "WBST311"
"WBST311C" "wbst311J" "WBST311X" "WBST845")
```

(CH.LIST.OBJECTS *ObjectPattern Property*) ž returns a list of all of the objects whose object name matches the *ObjectPattern* AND who have property *Property*. *ObjectPattern* consists of an NS name. The object part (but not the domain and organization parts) can include the * wildcard character, standing for 0 or more of any characters. If the domain and/or organization parts are left off, the default domain and/or organization is used. If

ObjectPattern is NIL or just *, then the list returned will be a list of all objects in the default domain and organization.

The *Property* argument indicates the type of object to look for and can be one of: USER, PRINT.SERVICE, FILE.SERVICE, MAIL.SERVICE, MEMBERS, or ALL as described above. If the *Property* argument is NIL, then ALL will be used meaning all types of objects.

Examples:

```
28_ (CH.LIST.OBJECTS (QUOTE *:Parc:Xerox)
      (QUOTE FILE.SERVICE]
      ("phylex" "Starfile")
```

```
29_ (CH.LIST.OBJECTS (QUOTE *:Parc:Xerox)
      (QUOTE PRINT.SERVICE]
      ("Kukai" "PaperMate" "print")
```

```
30_ (CH.LIST.OBJECTS ' *:Parc:Xerox 'USER)
      ("Akis Doganis" "Ann Derrick" "Arnold J Blum"
       "Art Farley" "Barbara Hemstad" "Bart Kinne"
       "Beaumont Sheil" "Benny Pugh" "Bev Manes" "Bill
       Hunter" "Bill Jackson" "Bill van Melle" "Bill
       Winfield" "Bob Allen" "Bob Ritchie" "Brad
       Burns" "CA Lehner" "Cari Sullivan" "Carlin
       DeCato" "Carol Lehner" "Cathy Turner" "Charles
       Orgish" "Cheryl James" "Courtney Tagupa" "Cyndi
       Vanderhorst" "D. Austin Henderson" "Dale Mann"
       "Dan Jordan" "Dan Russell" "Dave Pirogowicz"
       "David Myron Levy" "David Porter" "David
       Vinayak Wallace" "David Weckler" "Diane
       Hutchins" "Don Charnley" "Dorene Allen" "Doug
       Walters" "Ed Fiala" "Eric Rawson" "Eric Schoen"
       "Eric Steffensen" "Facilities Monitoring"
       "Fernando Ponce" "Frances Grimble" "Frank
       Halasz" "Frank Shih" "Frank Vest" "Fumiko
       Mannes" "Gary Chang" "Gary Emanuel" "Gary
       Rhoades" "Gary Toyama" "Gene Hall" "Giuliana
       Lavendel" "Gloria Warner" "Greg Nuyens" "Gregor
       Kiczales" "Guest" "Hal Murray" "Harriet Weeks"
       "Henry S. Thompson" "Herb Jellinek" "Hugh Vander
       Plas" "I-Wei Wu" "Irene Lile" "Jacqeline M.
       Guibert" "Jay Trow" "Jean Gascon" "Jeanette
       Figueroa" "Jeannie Lewandowski" "Jim Cooper"
       "Jim D'Alfonso" "John Brown" "John D. Sybalsky"
       "John L. White" "John Larson" "John S. Brown"
       "John Shaw" "John White" "Jon Bokelman" "Joseph
       Kaminski" "Julian Orr" "Karen Martelli" "Kathy
       Jarvis" "Kelly Roach" "Kenneth Beckman" "Kerry
```


Brown" "Larry Masinter" "Librarian" "Lillian
 Barth" "Lorraine Watanabe" "LouAnne Johnson"
 "Lynne Seymour" "Maia Pindar" "Mariela Esser"
 "Mark Chow" "Mary Hausladen" "Meg Withgott"
 "Melissa Monty" "Michael Dawson" "Michael
 Fisher" "Michael Herring" "Michael Plass"
 "Michael Sannella" "Michael Young" "Michalene
 Casey" "Michel Desmarais" "Mike Dixon" "Mimi
 Gardner" "Mitchell Lichtenberg" "Nancy Freige"
 "Neil Gunther" "Nicholas Briggs" "PARCPublic"
 "Patricia Sheehan" "Paul Ricci" "Paul Turner"
 "PC Demo" "Per-Kristian Halvorsen" "Peter
 Struss" "Richard Burton" "Richard E. Sweet"
 "Richard Martin" "Robert Allen" "Robert
 Bachrach" "Robert Spinrad" "Robert Tremain"
 "Ronald Kaplan" "Ronald Schmidt" "Salina
 Snipes" "Sharon Johnson" "Sharon Penner" "Star1"
 "Star2" "Star3" "Star4" "Stephen Jackson"
 "Stephen Quarterman" "Steve Martino" "Steve
 Wallgren" "Steven Purcell" "Susan Newman" "Susi
 Lilly" "Susie Mulhern" "Sweetsun Chen" "Tak
 Oki" "Tami DeMerritt" "Tayloe Stansbury" "Terry
 Haney" "Thomas Hartmann" "Tiao-Yuah Huang" "Tim
 Brunner" "Tim Diebert" "Toby Morrill" "Tom
 Moran" "Victor Bojorquez" "Victoria Carlson"
 "Wes Dorman" "Yoko Nonaka" "Zoran Popovic")

(CH.LIST.ALIASES.OF *ObjectPattern*) ž returns a list of all of the aliases of *Object*, where *Object* is an NS name.

The object part (but not the domain and organization parts) can include the * wildcard character. However, the function will return the aliases of only the first object found that matches the pattern. Thus, it makes little sense to use wildcards in this function.

If the domain and/or organization parts are left off, the default domain and/or organization is used. If *Object* is NIL, * is assumed and the aliases of the first object in the default domain will be returned.

Examples:

36_(CH.LIST.ALIASES.OF (QUOTE Frank% Halasz))

(Halasz:)

37_(CH.LIST.ALIASES.OF (QUOTE StarFile:))

(Help Server: Help Service:)

38_(CH.LIST.ALIASES.OF (QUOTE Phylex:))

(aifile:)

```
39_(CH.LIST.ALIASES.OF (QUOTE aifile:))
(aifile:)
40_(CH.LIST.ALIASES.OF (QUOTE *:PARC:))
(ht: HThompson:)
```

(CH.RETRIEVE.MEMBERS *Object* 'MEMBERS) ž retrieves the list of members of a distribution list. *Object* is the NS name of the distribution list. A distribution list is an NS object with a MEMBERS property, i.e., an object returned by **(CH.LIST.OBJECTS "*:PARC:XEROX" 'MEMBERS)**.

Example:

```
48_(CH.LIST.OBJECTS '* 'MEMBERS]
("AllParc" "AllXerox" "Alpha BWS" "BWS Users"
 "HelpGroup" "ICL Star Users" "LispAccess"
 "LISPCORE" "NewXAISEmployees" "PTS")
49_(CH.RETRIEVE.MEMBERS (QUOTE PTS:)
 'MEMBERS]
(Carol Lehner: Toby Morrill:)
```

Mail Service

The NS world provides a mail delivery service similar to, but separate from, the Grapevine mail system. To use this mail system, you will need get registered in a Clearinghouse and have a mail folder set up for you on some mail server. At PARC, see Carol Lehner to have this done.

If you load {eris}<lisp>harmony>library>nsmail.dcom then evaluate the function call **(LAFITEMODE 'NS)**, you can use Lafite to gain access to your NS mail. The Lafite *Get Mail* command will retrieve mail from your NS mail folder and the *Send Mail* command will send the mail out using the NS mail system.

You can send and receive mail in only one world at a time. To read your Grapevine mail again, evaluate **(LAFITEMODE 'GV)**. To return to NS mail use **(LAFITEMODE 'NS)** again.

You can intermix Grapevine and NS mail messages in a Lafite mail folder. That is, you needn't use different mail folders as you switch between the two mail

systems. However, the *Answer* and *Forward* will not work correctly for NS mail when the LAFITEMODE is GV, and vice versa.

When sending NS mail, you need to include the full NS name of the recipient (i.e., name/alias, domain, and organization). If the recipient belongs to the same domain and/or organization, you can omit these parts of the NS name. For example, to send mail to me the recipient list should be *Halasz:PARC:Xerox* or *Frank Halasz:PARC:Xerox*.

Your NS mail server and the Clearinghouse take care of delivering the mail to the appropriate mail server for each recipient.

I think there are distribution lists allowed in the NS mail system. These are the NS objects with the MEMBERS property discussed above. Mailing to these distribution lists is like mailing to each member of the list. For example, mailing to *PTS:PARC:Xerox* is like mailing to *Lehner:PARC:Xerox* and *Morril:PARC:Xerox*, since these are the only two MEMBERS of *PTS:PARC:Xerox*.

Unlike in Grapevine, these distribution lists can be changed only by the Clearinghouse administrator. Although they can be examined using the CH.LIST.OBJECTS and CH.RETRIEVE.MEMBERS functions described above.

File Service, Print Service, Press versus Interpress printers

File service in the NS world is provided by NS file servers. These are similar to IFSs, but do not have some of the IFS features. The salient points of NS file server have been discussed in the LispCourse sections on filing.

Print service in the NS world is provided by NS printers (the Xerox 8044 printer). The 8044 NS printers are abysmally slow but in general have much higher print quality than the Dover and full-press printers in the PUP world. All NS printers print files only in the Interpress document format. Press files cannot be printed in the NS world.

Most files in Interlisp can be printed on either the Press (Pup) or Interpress (NS) printers. But note that the fonts available on the two printers are different.

Interlisp does most of its screen displays in PUP-world fonts like Gacha, TimesRoman, and Helvetica. When printed on a Press printer, these fonts appear as they do on the screen (except at higher resolution). When printed on an Interpress printer, these fonts are translated into their NS analogs: Terminal, Modern, and Classic. For most applications, this is okay. But files which are, for example, carefully adjusted to fit on a Press page will not fit in the same way (or not at all) on an Interpress page.

You can use the NS fonts on the screen (e.g., in a TEdit window) by setting the correct font variables. When printed on a Press printer, these fonts will be printed in their Pup-world analogs. When printed on an Interpress printer, these fonts will appear as they do on the screen (modulo the resolution).

Interfaces Between the PUP and NS Worlds ž Interlisp-D and Mail Gateways

The PUP and NS world are generally separate worlds. But since Interlisp-D speaks to both worlds, you can often mix and match operations in the PUP world with operations in the NS world. For example, you can COPYFILE a file between an IFS and an NS file server as in (*COPYFILE* '{*phylex*:}<*halasz*>init' '{*phylum*}<*halasz*>init).

In all of the interactions where you are dealing with a mix of NS and PUP, your Interlisp machine is the intermediary between the two worlds. For example, COPYFILE copies the file from the NS server to the Lisp virtual memory using the NS filing protocol and then copies it from the Lisp virtual memory to the IFS using the FTP protocol.

There is one interface between the PUP and NS worlds that exists apart from your machine: the mail gateway between the Xerox PUP Internet and the Xerox NS Internet. The mail gateway is a machine running somewhere in OSD that takes Grapevine mail addressed to NS mail recipients, translates it into NS mail and gives it the NS mail system to deliver. Similarly, it takes NS mail intended for Grapevine recipients, translates it to Grapevine mail, and then gives it to the Grapevine to deliver.

To send mail through the gateway from Grapevine to an NS mail recipient, you should address the mail to "*NSName*".ns, where *NSName* is the NS Name of the recipient. For example, to send me NS mail from Lafite running in GV mode,

you should address the mail to *"Halasz:PARC:Xerox".ns* (don't forget the " before and after the NS Name).

To send mail through the gateway, from the NS world to the Grapevine world, you have to carry out a similar, but presently (9-Apr-85 00:16:00) unknown trick in specifying the recipients address.

Documentation on Networks

There is relatively little documentation on the Interlisp/NS world.

The Interlisp/PUP world is scattered throughout the IRM and package documentation, since almost all parts of Interlisp take advantage of the network at times.

The Interlisp interface to both NS and PUP networks is covered in Chapter 21 of the IRM. All of the Clearinghouse functions described above are covered here. Most of the chapter, however, is aimed at programmer's interface to the PUP and NS networks.

The Lafite interface to NS mail is documented in
{eris}<lisp>harmony>library>nsmail.tedit (&.press).

There are several PARC Blue&White reports covering various aspects of both the PUP and NS network designs. You can probably get a list of these from the TIC.

LispCourse #21: Fonts

Fonts ž Characterized by Family, Size, Face, Rotation, and Device

A **font** is a description of how alphanumeric characters should look when displayed on the screen or on printed hardcopy.

Every font has five basic characteristics, *family*, *size*, *face*, *rotation*, and *device*:

Family ž the "style" of the font, e.g., TimesRoman, Gacha, Helvetica, Modern, etc.

In the Interlisp-D (and most Xerox products), style characteristics like sans-serif and proportional spacing are not independent characteristics of a font. Each font family simply has a particular set of style characteristics. For example, Helvetica is sanserif, proportionally spaced; TimesRoman is serif, proportionally spaced; Gacha is sanserif, fixed space; etc.

In Interlisp-D, family is specified by a single atom, e.g., GACHA, TIMESROMAN, or TERMINAL.

Size ž the size of the characters measured in points (72 points to an inch).

Size actually measures the distance from the top of the tallest character to the bottom of the lowest character (e.g. the distance between the top of the L and the bottom of the g).

In Interlisp-D, size is specified by an integer.

Face ž the weight, slant, and spacing of the characters. Bold, Italic, and BoldItalic are the typical examples of font faces. In general, a font face has three parameters:

Weight ž measure the thickness of the characters. Possible values in Interlisp-D are BOLD, MEDIUM or LIGHT.

Slope ž indicates the slant of the characters. Possible values in Interlisp-D are ITALIC and REGULAR.

Expansion ž measures the spacing between characters. Possible values in Interlisp-D are REGULAR, COMPRESSED, and EXPANDED.

In Interlisp-D, a font's face is generally specified by a list of three atoms where the three atoms correspond to the weight, slope, and expansion parameters, respectively.

Three character atoms using the first letter of each field in the list are also allowed. For example, MIC can be used in place of (MEDIUM ITALIC COMPRESSED).

A few special atoms also exist for the common faces. These atoms are:

STANDARD = (MEDIUM REGULAR REGULAR) = MRR

ITALIC = (MEDIUM ITALIC REGULAR) = MIR

BOLD = (BOLD REGULAR REGULAR) = BRR

BOLDITALIC = (BOLD ITALIC REGULAR) = BIR

Rotation ž a number between 0 and 360 that indicates the degree to which the characters are rotated (clockwise?) from upright.

Font rotations are generally 0 degrees, except in special applications.

Many devices (see below) allow only 0 and 90 degree rotations, where 0 is the portrait mode orientation (as on this page) and 90 is the landscape mode orientation. Other devices support a full range of font rotations from 0 to 360.

Device ž specifies the device on which the font is to be displayed or printed.

In Interlisp-D, the standard devices are DISPLAY, PRESS, and INTERPRESS, referring to the D-machine display, Press printers, and Interpress printers respectively. Other devices can be added to the system by loading various packages.

In Interlisp-D, a font is specified by a list of five items, its *family*, its *size*, its *face*, its *rotation*, and its *device*.

There are no defaults for the family and size parameters. If absent, *face* defaults to STANDARD; *rotation* defaults to 0; and *device* defaults to DISPLAY.

Examples:

(TIMESROMAN 12 BOLD 0 PRESS)

(MODERN 10 (MEDIUM MEDIUM REGULAR))
 (HELVETICAD 24 STANDARD 90 INTERPRESS)
 (TIMESROMAND 36 MRR NIL DISPLAY)

Not all combinations of family, size, face, rotation and device exist as actual fonts. For example, GACHA does not exist for size bigger than 12 points.

Moreover, the Modern, Terminal, and Classic dynasty of font families (the Interpress fonts from the NS world) does not generally exist for Press devices. Similarly, the Helvetica, Gacha, and TimesRoman dynasty of font families (the Press fonts from the PUP world) does not generally exist for Interpress devices.

Each font as characterized by family, size, face, rotation and device contains all the information necessary for displaying and/or printing some set of alphanumeric characters \dot{y} usually the alphabet, the digits, plus most of the common special characters like period, colon, semi-colon, etc.

Note: Not all fonts, however, contain all characters. For example, the LOGO font has information describing only the characters X, E, R, & O.

Fonts differ widely on how many of the special characters they handle. Some fonts have all of them, other have only a few of the most common. There are font dictionaries on ???.

Describing Fonts to Interlisp-D \dot{z} font lists & font descriptors

When specifying a font to Interlisp-D, you can use one of two descriptions: a *font list* or a *font descriptor*.

A *font list* is the list of 5 elements described above, i.e., (Family Size Face Rotation Device).

A *font descriptor* is an Interlisp object that specifies a font. To create a font descriptor, use the function call (**FONTCREATE *Family Size Face Rotation Device***), where Family, Size, Face, Rotation, and Device are as described above. This will return a font descriptor object which you can use to specify a font.

All Interlisp functions eventually use font descriptors. If you specify a font list to a function, it will create the font descriptor for you.

The difference is one of time: FONTCREATE immediately goes out to the file server containing the font information, finds the correct font information, and loads it into virtual memory. This takes time, sometimes a lot of time.

If you want control over when to spend this time, then you call FONTCREATE yourself and use font descriptors.

If you don't want control over the time, then use font lists. The first time any function needs to use each font, there will be long delay while FONTCREATE goes and fetches the font information.

I always use FONTCREATE in my INIT file for all my default fonts. Therefore, I spend the font finding time altogether at the beginning (i.e., whenever I load my system).

Fontclasses ž the "same" font for different devices

Because not all devices support the same fonts, Interlisp-D has an entity called a *fontclass*. A fontclass contains a list of font-device pairs describing what should be considered to be the "same" font on the different devices.

The list is of the format (*DisplayFont PressFont InterpressFont OtherFontPair1 OtherFontPair2 ...*). *DisplayFont* is a font specification for a font with device DISPLAY. *PressFont* is a font specification for a font with device PRESS. *InterpressFont* is a font specification for a font with device INTERPRESS. Each *OtherFontPairi* is a list of two items: a device name and a font specification for that device.

Examples of fontclass lists:

```
((TIMESROMAN 12)
 (TIMESROMAN 10 STANDARD NIL PRESS)
 (CLASSIC 10 STANDARD NIL INTERPRESS)
 (IRIS (ROMAN 10 STANDARD 0 IRIS)))
```

To create a fontclass, use the function call (**FONTCLASS *Name FontClassList***). *Name* is an arbitrary (and optional) atom that names the fontclass. *FontClassList* is a fontclass list as described above. The function will return an Interlisp object called a fontclass.

Almost any function in Interlisp-D that accepts a font list or a font descriptor as an argument will accept a fontclass object instead. It will then extract from the fontclass the appropriate font description for the device it is working with.

Fontdirectories ž Telling Interlisp where to find font information

Font Information Files

Before Interlisp can use a font, it has to load in all of the information about that font (e.g., the bit maps of the characters, the width and height of each character, etc.). This loading is done by the FONTCREATE function, either when called by the user or automatically as described above.

Once the font information for a given font has been loaded in, it stays in virtual memory and is used each time the font is used.

The exact information necessary for Interlisp to use a font differs between devices. In particular, for the DISPLAY device Interlisp needs an exact rendition (i.e., a bit map) of the character it displays on the screen. For PRESS and INTERPRESS devices, Interlisp needs only the exact size (width, height) of each character. The exact rendition of the character is stored ONLY on the printer itself.

This leads to a problem sometimes. It can happen that PRESS or INTERPRESS font information is available to Interlisp, but the printer on which you print your file does not have the required font. In this case, the printer will usually try to substitute a similar font that it does have information for. If the printer can't do the substitution, then it prints an error message on the header page. Note: this happens after the file has left Interlisp and been sent to the printer. Interlisp cannot detect error like this.

For example, if you send a file with TimesRoman 16 to Quake, it prints it as TimesRoman 17 with an appropriate warning on the header page.

Font information for the DISPLAY device is stored in files located on file servers or on the local disk. One font is stored per file. The name of the file is used to

indicate the font contained in the file. The file name extension is one of .DISPLAYFONT, .STRIKE, or .AC.

For example, TimesRoman10.strike contains information about the TimesRoman 10 font for displays.

Font information for the **PRESS** device is all stored on one giant file call FONTS.WIDTHS stored on a file server or on your local disk.

Font information for the **INTERPRESS** device is stored in files located on file servers or on the local disk. The name of the file is used to indicate the font contained in the file. The extension is always .WD.

For example, CLASSIC10-C100.WD is a file containing information about the Classic 10 font for Interpress printers.

Font Directories

Various fonts are located on various directories on various file servers. So Interlisp is designed to search through all known font directories until it finds the font it is looking for. (This is what usually takes the time during a FONTCREATE). If it can't find the font, an error occurs.

There are several global variables that specify what directories/files Interlisp should know about when looking for font information. These variables are:

DISPLAYFONTDIRECTORIES ž a list of the directories Interlisp should search in (in order) when looking for the files containing font information for the DISPLAY device.

```
Default in ISL is ( {ERIS } <LISP> FONTS >
  { PHYLUM } <STARFONTS> SCREEN > L FONTS >
  { PHYLUM } <ALTOFONTS >
  { PHYLUM } <ALTOFONTS > ORIGINAL >
  { INDIGO } <ALTOFONTS >
  { INDIGO } <ALTOFONTS > ORIGINAL > { DSK } )
```

DISPLAYFONTEXTENSIONS ž a list of file name extension Interlisp should use to look for font information for the DISPLAY device.

Default is (DISPLAYFONT STRIKE AC). When looking for information on TimesRoman 10 Interlisp will look for TimesRoman10.displayfont, TimesRoman10.strike, or TimesRoman10.ac.

PRESSFONTWIDTHSFILES ž list of FONTS.WIDTHS files Interlisp should search in (in order) when looking for the font information for the PRESS device.

Default in ISL is ({ERIS} <LISP> FONTS > FONTS . WIDTHS
 { PHYLUM } <ALTOFONTS> FONTS . WIDTHS
 { INDIGO } <FONTS> FONTS . WIDTHS
 { DSK } FONTS . WIDTHS)

INTERPRESSFONTDIRECTORIES ž a list of the directories Interlisp should search in (in order) when looking for files containing the font information for the INTERPRESS device.

Default in ISL is ({ERIS} <LISP> FONTS >
 { PHYLUM } <STARFONTS> FORMATTER > WIDTHS >)

These variables are usually set in the site INIT file. You may want to set them to include additional directories. For example, you may create your own display fonts (see below) and store them in your personal font directory.

You may also want to move some frequently used font files to your local disk so that you don't have to wait on the file server all the time. In this case, you should make sure that these gloabl variables are set to include the partition/directory on your local disk that contains the fonts. For example, I keep fonts on partition 4 of my Dorado disk. My INIT file has the following clause (ADDVARS (DISPLAYFONTDIRECTORIES {DSK4}) (INTERPRESSFONTDIRECTORIES {DSK4}) (PRESSFONTWIDTHSFILES {DSK4})).

Available Fonts

You can find out what fonts are already loaded into your virtual memory and/or what fonts are available to be loaded into your virtual memory given the current settings of the font directores/files global variables. To do so use the function call (**FONTSAVAILABLE *Family Size Face Rotation Device FilesTooFlg***). *Family*, *Size*, *Face*, *Rotation*, and *Device* are as specified above, except that any or all can be the wildcard atom * indicating all values of the field (e.g., all sizes or all rotations).

FONTSAVAILABLE returns a list of font lists for all of the fonts already in virtual memory that match the *Family*, *Size*, *Face*, *Rotation*, and *Device* specified.

If *FilesTooFlg* is non-NIL, then the list returned will include all fonts that COULD be loaded into virtual memory given the current global variable settings.

Examples:

```
50_(FONTSAVAILABLE 'GACHA '* NIL 0 'DISPLAY NIL)
((GACHA 8 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (GACHA 12 (MEDIUM REGULAR REGULAR) 0
 DISPLAY)
 (GACHA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY))
```

Default Fonts

When a particular Interlisp application prints something on the screen or on a printer, it has to make a choice of what font to use. If there is no way for the user to specify which font to use (e.g., as in DEdit) or if the user hasn't specified a specific font (e.g., as in TEdit), Interlisp will use some sort of default font.

Often you want to change this default font. For example, you might want to change the font used in DEdit windows. Or, you might want to change the font that TEdit starts-up using.

Unfortunately, the status of default fonts in the system is very, very sad. There are several different mechanisms in use for setting default fonts. Different packages use different combinations of these mechanisms.

Worse yet, some packages use the same mechanism and same global variables to set default fonts. When this happens changing a default font for one package (e.g., DEdit)

automatically changes the font for some other unrelated package (e.g., TEdit), and vice versa.

Even in this chaos, there is some order. There are three basic ways of specifying default fonts.

FONTPROFILE – beginnings of a central mechanism for fonts

There is a global variable called FONTPROFILE. Its value is a list. Each item on the list is of the form (*Name Number FontClassList*) or of the form (*Name OldName*). Each item defines a font named *Name*. In the first format, *Name* is defined by the font class specified by *FontClassList*. The *Number* represents the font number for purposes of printing out Lisp code in the LISTFILES package. In the second format, *Name* is simply defined to be the same as the font previously called *OldName*.

The standard FONTPROFILE list looks like:

```
( (DEFAULTFONT 1 (GACHA 10) (GACHA 8) (TERMINAL 8) )
  (BOLDFONT 2 (HELVETICA 10 BRR) (HELVETICA 8 BRR) (MODERN 8
BRR) )
  (LITTLEFONT 3 (HELVETICA 8) (HELVETICA 6 MIR) (MODERN 8
MIR) )
  (BIGFONT 4 (HELVETICA 12 BRR) (HELVETICA 10 BRR) (MODERN 10
BRR) )
  (USERFONT BOLDFONT)
  (COMMENTFONT LITTLEFONT)
  (LAMBDAFONTBIGFONT)
  (SYSTEMFONT)
  (CLISPFONT BOLDFONT)
  (CHANGEFONT)
  (PRETTYCOMFONT BOLDFONT)
  (FONT1 DEFAULTFONT)
  (FONT2 BOLDFONT)
  (FONT3 LITTLEFONT)
  (FONT4 BIGFONT)
```

```
(FONT5 5 (HELVETICA 10 BIR) (HELVETICA 8 BIR) (MODERN 8
BIR) )
(FONT6 6 (HELVETICA 10 BRR) (HELVETICA 8 BRR) (MODERN 8
BRR) )
(FONT7 7 (GACHA 12) (GACHA 12) (TERMINAL 12))
```

Various packages use the named fonts to print things out. For example, when printing comments in Lisp code, Interlisp uses the font called COMMENTFONT to print out comments. It uses the font called LAMBDAFONT to print out the name of functions. And so on.

In fact, most of the entries on FONTPROFILE are concerned with printing out Lisp code.

HOWEVER, the DEFAULTFONT entry is used for a number of other packages, in fact way, way to many packages. Changing the DEFAULTFONT entry on FONTPROFILE will affect the following default fonts:

DEdit ÿ the font used to print things out in the DEdit window

TEdit ÿ the font that TEdit starts up in and uses if you don't specify another font

Exec ÿ the font the Lisp Exec and the Break Exec use to print in their Exec windows. Note: changing DEFAULTFONT does not immediately change the font being used in the top level Exec window. To do this, you should type (**DSPFONT DEFAULTFONT TOPW**) into the Exec window after changing the FONTPROFILE. The Exec window will then change fonts immediately to the DEFAULTFONT.

??? ÿ DEFAULTFONT is probably used by many other package I don't know about.

Note the problem. Since all of these packages use DEFAULTFONT, you cannot change them independently; changing one changes them all.

Changing the FONTPROFILE

To change the FONTPROFILE, you have to do two things:

- 1) Edit the FONTPROFILE list using DEdit; i.e., execute **(DV FONTPROFILE)**. Change any entry (most probably the DEFAULTFONT entry) to the desired font.
- 2) Execute the function call **(FONTPROFILE FONTPROFILE)**. This will install your changes in the system.

Most often you want to change your DEFAULTFONT for all time. In this case, you have to put an entry into your INIT file. The procedure for doing this is the following:

- 1) Edit the FONTPROFILE list using DEdit; i.e., execute **(DV FONTPROFILE)**. Change any entry (most probably the DEFAULTFONT entry) to the desired font.
- 2) Edit your INIT COMS list using DEdit; i.e., call **(DC INIT)**. Place the following two clauses into your INITCOMS:

(VARS FONTPROFILE)

(P (FONTPROFILE FONTPROFILE))

This will save the edited FONTPROFILE list and cause it to be installed whenever your INIT file is loaded.

- 3) Save your new INIT file by doing a (MAKEFILE 'INIT) after connecting to your home or lisp directory.

The Future

In the future, I hope that all fonts will go through this FONTPROFILE and that there will be separate entries for each package (e.g., a TEDITDEFAULTFONT entry, a DEDITFONT entry and so on.). But, ...

Per package global variables ÿ the decentralized mechanism for fonts

Many packages maintain their own set of global variables that they use to determine their default fonts. To change the default font for these packages, you have to set the appropriate global variables for the package to the desired font description (i.e., font list, font descriptor, or font class).

Example: Lafite has 7 global variables that determine the font to use for various windows and tasks in Lafite. The variables are:

LAFITEEDITORFONT, LAFITEMENUFONT, LAFITETITLEFONT,
LAFITEDISPLAYFONT, LAFITEHARDCOPYFONT,
LAFITEBROWSERFONT, and LAFITEENDOFMESSAGEFONT

If you want to change, for example, the font in which LAFITE displays its message on the screen you have to set the variable LAFITEDISPLAYFONT to a new font descriptor.

(SETQ LAFITEDISPLAYFONT (FONTCREATE 'Helvetica 10 'BOLD))

You may want to put these changes in your INIT file to make them permanent. For example, in my INIT file I have several clauses of the form:

(VARS (LAFITEDISPLAYFONT (FONTCREATE 'Helvetica 10 'BOLD)))

To determine the global variables to set to change the default font for a package, you have to look in the parameters/variables section of the documentation for that package. There is no global directory for font related global variables!!!!.

Window title font

Setting the default font for window title bars has its own mechanism.

To change the title bar font, use the function call:

(DSPFONT *FontDescription* WindowTitleDisplayStream), where *FontDescription* is a description of the desired font for the window title bars.

Example, in my INIT file I have the following clause:

*(P (DSPFONT (FONTCREATE 'HELVETICA 14 'BOLD)
WindowTitleDisplayStream))*

Designing your own screen fonts

You can design your own screen fonts (put not printer fonts) using the LispUser package EDITFONT. See the documentation on EDITFONT for further details.

References

What is documented about fonts is described in Section 19.8 of the IRM.

FONTPROFILE is documented in Section 6.8.5 of the IRM.

Fontclasses are documented in the Harmony Release Notes.

Also, see the parameters/variables sections for all the Lisp packages like CHAT and LAFITE.

LispCourse #22: Overview of Lisp Packages; Using the Documentation

Lisp Packages

Recall: Lisp packages are sets of related functions contained on a single file (or small set of files) that are designed to carry out some particular task in the Interlisp environment, e.g., text editing, file transfer, etc.

There are by convention two types of public packages in the Interlisp world, *LispUsers* packages and *Lisp Library* packages. The former are random packages submitted by random hackers. The latter are packages maintained by the AISBU Lisp development group.

A third kind of "package" are the large application systems build on top of Interlisp such as LFG and NoteCards. We will not talk about these types of large applications here.

To get the functionality of a package (if it is not already contained in the default sysout), you need to find and then load the file(s) containing the package.

LISP LIBRARY Packages of Interest to the Non-Programmer

At PARC, the following packages are stored on {eris}<lisp>*release*>library> [where *release* is Harmony or Intermezzo, or ...]. They are packages designed and implemented and maintained by the AISBU Lisp group. Thus their reliability and usability is extremely high.

The Big Ones

FILEBROWSER ž Edit, Delete, Load, Compile, Copy, Rename, See, Hardcopy, & Info on files on any file device through a common graphic interface. Call: (FILEBROWSER FILEPATTERN). *Documentation = ???*

SKETCH ž Sketch is a drawing program that enables you to place text and graphics to achieve desired images. The figures can be copy-selected into TEdit documents to allow a mixture of text and graphics in the same document.
Documentation = SKETCH.TEDIT

TEDIT ž The Interlisp-D text editor. *Documentation = IRM*

GRAPHER ž contains a collection of functions and an editor for laying out, displaying, and editing graphs (i.e., networks of nodes and links).

Documentation = GRAPHER.TEDIT

Mail Handling

LAFITE ž Interlisp mail program, a la Laurel/Hardy. *Documentation = LAFITE.TED*

MAINTAIN ž Lisp implementation of Grapevine MAINTAIN program for adding or deleting names from mail distribution lists. *Documentation = ???*

NSMAIL ž Add on to Lafite for handling NS Mail. *Documentation = NSMAIL.TEDIT*

MAILSCAVENGE ž The Lisp Library package MAILSCAVENGE is used to rebuild the internal pointers in a mail file that has been damaged. Lafite generally reports “Can’t parse file” and terminates its Browse command when it detects damage in a file. The simplest remedy is to call MAILSCAVENGE, then browse the file again. *Documentation = MAILSCAVENGE.TEDIT*

Networks & Files

COPYFILES ž Functions for copying sets of files from one directory to another. *Documentation = COPYFILES.TEDIT*

FTPSERVER ž Lisp implementation of PUP FTP (File Transfer Protocol) server. Lets others FTP from you while you’re running lisp. *Documentation = FTPSERVER.TEDIT*

RS232, RS232CHAT, RS232EXEC, RS232FTP, RS232LOGIN ž Basic software drivers for serving the RS232 serial interface including CHAT and FTP service. *Documentation = RS232.TTY*

TCP, TCPCHAT, TCPFTP ž Basic software drivers for communicating with TCP/IP (Arpanet protocols) hosts (e.g., Sun workstations, Vaxes, etc.) including CHAT and FTP service. *Documentation = TCP.TEDIT*

TELERAID Interlisp-D has facilities for looking at sysout files and machines across the network, used by hackers to debug when your DLion goes into a 9XXX error in the maintenance panel. *Documentation = TELERAID.TEDIT*

Special Printers

FX80STREAM ž a library of routines used for driving an Epson FX-80 dot-matrix printer. With FX80STREAM you can use the full set of Interlisp-D device-independent graphical operations to compose pages on your FX-80, including printing TEdit documents. *FX80STREAM.TEDIT*

FXPRINTER, FXPARALLELPRINTER ž allows a user to print Lisp files to an Epson FX-80 printer. *Documentation = FXPRINTER.TED & FXPARALLELPRINTER.TED*

PRINTER ž package to do hardcopy of bitmaps and listfiles with multiple fonts for C.ITOH (Cheap!!!) printer connected to Dolphin parallel port. *Documentation = PRINTER.TTY*

Fonts & Bitmaps

READAIS ž Read, write, transform AIS (color/grey scale) files (common files containing pretty images). *Documentation = READAIS.TXT*

BIG ž Function NEWFONT sets up default fonts to be "size", where size is one of the atoms BIG, MEDIUM, STANDARD, SMALL. Changes the prettyprint fonts, the default font for the break window, typin, etc. Does this by resetting FONTPROFILE and then setting the fonts for all of the known windows. *Documentation = BIG.TTY*

BITMAPFNS ž Miscellaneous functions for manipulating BITMAPs. Reading and writing bitmaps, reading certain press files, creating window with image of bitmap. *Documentation = BITMAPFNS.TTY*

EDITBITMAP ž provides an interface (EDIT.BITMAP) for manipulating bitmaps. It puts up a menu of bitmap manipulation commands, one of which is HAND.EDIT which accesses EDITBM, the Interlisp-D bitmap editor. Other commands include shifting (in four directions), rotation (left and right 90 degree), inverting (horizontally, vertically, about diagonals), interchanging black and white, adding a border. *Documentation = EDITBITMAP.TTY*

Misc Tools

PAGEHOLD ž Redefines the default PAGEFULLFN, and provides hooks for making individual non-TTYDISPLAYSTREAM windows scrollable. Scrolling is "held" for up to PAGE.WAIT.SECONDS seconds, during which time an attached

"button" on the window softly flashes, and then the hold is "released". Holding down either SHIFT key will continue the "hold" (i.e., prevent "release"); letting up on either SHIFT key will "release" the "hold". *Documentation = PAGEHOLD.TEDIT*

SAMEDIR ž Advises MAKEFILE so that user can't inadvertently write out a file onto a directory other than the one it came from. Checks FILEDATES property against connected directory. *Documentation = SAMEDIR.TTY*

SINGLEFILEINDEX ž Package for giving user an alphabetical function index on the front of any lisp file listed thru lisp. Index number for a function indicates function's linear occurrence within file. Within the lisp source, each function is preceded by it's index number right justified on the page. *Documentation = SINGLEFILEINDEX.TEDIT*

Games & Demos

HANOI ž Displays and solves famous Towers of Hanoi problem. Can run as a background process. HANOIWINDOW can be reshaped. Call: (HANOI NRINGS WINDOW FONT ONCE). *Documentation = HANOI.TTY*

KINETIC ž Graphics demo. Fast random BITBLTs on a window. Call: (KINETICDEMO). *Documentation = ???*

UTILPROC ž Simple utility processes, including hall-of-mirrors demo. *Documentation = ???*

WINK ž Movie of Marilyn Monroe winking. Call: (SHOWMOVIE). Needs BITMAPFNS. *Documentation = ???*

LISPUSERS Packages of Interest to the Non-Programmer

The following packages are stored on {eris}<lispusers> at PARC. They are packages designed and implemented by random Lisp hackers and are not supported by the AISBU Lisp group. Their reliability and usability varies greatly.

Tools ž Editors, Printing functions, Graphics, etc.

AREDIT ž Tool for submitting, viewing, & editing Lisp ARs (action requests/Interlisp bug reports) from within Interlisp. Type (AR.FORM) to create

containing DATE, AUTHOR, FILE, CHANGES, & COMMENTS info is added.
Documentation = EDITHIST.DOC

EXEC ž This small package allows the user to create extra EXEC windows in which to do EVALQing. A new EXEC window may be created in either of two ways. The user may either do (EXEC) or button the EXEC menu item added to the background menu. *Documentation = EXEC.TTY*

FINGER ž Finger is a facility for determining and displaying information about other users running Interlisp-D. It displays the user's name, the Etherhostname (or the octal net address when no nameserver is available) and the user's idle time (time since last keystroke or mouseaction). Only other users who have the finger server loaded will be displayed. *Documentation = FINGER.TEDIT*

HEADLINE ž Three functions for manipulating windows containing headlines: HEADLINE, BILLBOARD, and CLOSEHEADLINES. Useful for titling a screen image or leaving message on screen while away. *Documentation = HEADLINE.TTY*

HISTMENU ž Provides simple way to access Interlisp history list using menu. REDO, UNDO, FIX, and ?? selected items. Call: (HistoryMenu histMenuLength histMenuPosition). *Documentation = HISTMENU.TED*

LANDPRESS ž Allows landscape printing of ASCII text files on press printers (e.g., Dovers). Function LANDPRESS produces same product as MAKEPRESS, only printed sideways on the page allowing for wider output. *Documentation = LANDPRESS.TTY*

MAILOPS ž Functions for poking into and "scavenging" Laurel/Lafite/... mail files. *Documentation = MAILOPS.TEDIT*

NOTEPAD ž Allows user to do artwork at bitmap level in NOTEPAD windows. Trajectories: sketch, line, circle, ellipse, open curve, closed curve. Objects/editing: text, are of screen, shade rectangle, fill, edit area. Style: brush, use mask, mask, use grid, grid, use symmetry, point of symmetry, text font, shade. *Documentation = NOTEPAD.TTY*

PRESTOIP ž PRESS TO InterPress. Converts PRESS files to INTERPRESS files. Call: (PRESS.TO.IP PRESSFILE IPFILE). *Documentation = PRESTOIP.TTY*

PROMPTREMINDERS ž User can be periodically reminded of important things by messages which are aggressively winked and flashed in the PROMPTWINDOW. *Documentation = PROMPTREMINDERS.TTY*

REMIND ž Facility for scheduling LISP events to take place at a specified later time. Reminders are stored on REMINDERS.LISP on user's directory and an entry on AFTERSYSOUTFORMS causes them to be loaded via LOADREMINDERS which may also be put in user's init file. *Documentation = REMIND.TTY*

SPACEWINDOW ž Puts a small "Space Allocation" window on screen. Shows a bar-chart of the amounts of the four types of memory space that have been allocated (fixed data, variable data, atoms, pnames). Display is updated every 60 seconds. *Documentation = SPACEWINDOW.TXT*

Hardcopying Screen Images

COPYIMAGE ž a small utility which facilitates making hardcopy or PRESS output of either ANY window on a screen, or else the entire screen. To use load COPYIMAGE.DCOM which updates the background menu, and then button the CopyImage item of the background menu. Hardcopy output goes to the FULLPRESS printer of your DEFAULTPRINTINGHOST. *Documentation = COPYIMAGE.TTY*

FULLSCREEN ž This package allows an entire screen's image to be printed on an Interpress printer. *Documentation = FULLSCREEN.TXT*

Changes/Additions to the Interlisp-D Interface

ANIMATE ž This small package contains functions for moving a non-rectangular bitmap smoothly around the screen, ways of using these to get big cursors, and bitmaps for a large arrow and a hand to be used as large cursors. *Documentation = ANIMATE.TTY*

AUXMENU ž This package creates a middle button background menu containing a number of convenient functions. These functions include a menu-driven CNDIR, LOGOUT, and other functions that require little user-interaction. The purpose of the package is to minimize typing. *Documentation = AUXMENU.TEDIT*

DEDITK ž Adds single button method for combining the most frequently combined pairs of BI/BO and BEFORE/AFTER/DELETE/REPLACE in DEDIT - Load and call (DEDITK). *Documentation = DEDITK.TED*

MACWINDOW ž Advises SHRINKW and EXPANDW to produce a zooming effect by showing the outline of their window arguments as they shrink or grow. *Documentation = MACWINDOW.TXT*

MOVE-WINDOWS ž MOVE-WINDOWS is a tool to help you re-arrange your screen quickly. Once loaded, buttoning in the background will shift you into window-moving mode. Buttoning again in the background will get you out of that mode. In window-moving mode, buttoning in a window with either the LEFT or MIDDLE button will either reshape or move the window: if you button down near a corner, the corner is moved; near a side, that side is moved; in the center, the whole window is moved. Buttoning in a window with the RIGHT button will call the usually window menu (DOWINDOWCOM). This is the best way to close windows, for example. *Documentation = MOVE-WINDOWS.TEDIT*

SNAPSCROLL ž Loading SNAPSCROLL advises (SNAPW) so that snapshot windows are henceforth shapeable and scrollable. This enables the user to, for example, snap a long list off the screen, and then reshape it to a reasonable size and scan it at will. *Documentation = SNAPSCROLL.TED*

TEDITKEY ž TEditKey is a package which provides a keyboard interface to TEdit. On a Dandelion, the interface takes advantage of the non-Alto keys. On Dorados and Dolphins, a window mimicking the Dlion function keys provides the same abilities. *Documentation = TEDITKEY.TEDIT*

TINYTIDY ž TINYTIDY takes the icons on your screen and lines them up along the edge. *Documentation = TINYTIDY.TEDIT*

Clocks

CROCK ž Function for creating and manipulating an analog face clock. Menu allows user to change style of clock. Call: (CROCK REGION). *Documentation = CROCK.TEDIT*

LCROCK ž Puts a digital/analog clock in the unused area of the LOGOW. (START.LCROCK <myLogo> <Position>) starts it up. <myLogo>, if non-NIL, will replace the "Interlisp-D" logo; and <Position>, if non-NIL, will be the lower-left corner (NIL means to use the position of the existing logo window). The

globalvar CROCKUPDATERATE.MS is the number of milliseconds between automatic updates. *Documentation = LCROCK.DOC*

Games & Demos

BLTDEMO ž Implements Smalltalk graphics demo in Interlisp. Spinning star, bouncing ring & box. Call: (BOUNCE X Y) where X and Y are velocities defaulting to 3. Box shows whatever is near cursor. Interesting recursive effects can be seen if you move the cursor near the box. *Documentation = BLTDEMO.TXT*

FACEINVADER ž A game. The object of the game is to shoot the bouncing 'face' before it overruns your base. Call: (FI INSTRUCTIONS?).
Documentation = ???

JARGON ž N random broken definitions from the infamous hacker's dictionary, snarfed from MIT-AI. (The globalvar JARGON.FILE.LOCATION points to the database file.) Starts up on load, or type (JARGON.READ) *Documentation = ???*

KAL ž Kaleidoscope demo. Call: (KAL). Control with middle button menu.
Documentation = KAL.TED

LIFE ž This Life program is a translation of the SmallTalk version in the book Goldberg, Robson: The Language and its Implementation. *Documentation = LIFE.TXT*

LINEDEMO ž This package contains a couple of random demonstration programs having to do with drawing random lines or polygons. *Documentation = LINEDEMO.TED*

NQUEENS ž Solves N Queens problem. How to place N queens on a chess board so that they don't attack each other. Graphics demo. Call: (NQUEENS N).
Documentation = ???

PACMAN ž Game. Runs in b/w or color. CALL: (PACMAN). *Documentation = ???*

PEANO ž Peano curves graphics demo. CALL: (PEANODEMO LEVEL SCALE). *Documentation = ???*

PLAY ž Offers Interlisp-D users a disciplined way to play simple musical melodies on Xerox 1108 machines. (PLAY.DEMO) demos the PLAY package. Main functions: PLAY.NOTES, PLAY.MELODY, PLAY.KEYBOARD.
Documentation = PLAY.TTY

QIX ž QIX is a small graphic demo modelled after the videogame of the same name. *Documentation = QIX.TEDIT*

SOLITAIRE ž The card game Solitaire (graphics demo). Call: (SOLO).
Documentation = ???

TRAJECTORY-FOLLOWER ž Provides a function which causes a snake to crawl along a trajectory. Trajectory is specified by a set of KNOTS and a CLOSED flag. *Documentation = TRAJECTORY-FOLLOWER.TTY*

DMT Equivalents (i.e., run after machine has been idle for a bit)

BLACKOUT ž (Blackout text interval) makes a back window the size of the screen and bounces a square around on it, like DMT, etc. Good for servers. Text and interval default to "Type Key" and NIL (= forever) if not given.
Documentation = BLACKOUT.TEDIT

BOUNCE ž Is another dmt variant. Blacks out the screen & draws patterns on it until you hit a mouse button or type any character. Can be started 3 ways.
Documentation = ???

FRACTAL ž An eyewash DMT program that draws fractals. *Documentation = FRACTAL.TEDIT*

Alternative Page Hold Schemes

NOWAITPRINT ž a function which will temporarily diddle a window's pagehold characteristics so that a print-without-holding may be performed. *Documentation = NOWAITPRINT.TTY*

YAPFF ž is Yet Another Page Full Function. I actually don't like this one much better than any of the others around, but its another point in the space of possible actions on end-of-page. *Documentation = YAPFF.TEDIT*

Using the Interlisp-D Documentation

The State of Interlisp-D w.r.t. Documentation

Sad, sad, sad, sad.

Interlisp-D documentation is:

1. *Incomplete* ž much of Interlisp is simply undocumented

2. *Out of date* ÿ last major manual revision was October, 1983
3. *Dispersed* ÿ documentation is spread across various binders, files, file servers, etc.
4. *For programmer's only* ÿ There is almost no user level documentation for the Interlisp-D environment; its all oriented toward the programmer and the system implementor. Interlisp-D the language is not properly distinguished from Interlisp-D the computer environment.

Bottom line is that documentation is THE major flaw of Interlisp-D, especially for the non-programming user of the Interlisp-D environment.

How to Use the Available Documentation

Isolate the user information from among the programmer/implementor information.

How? Beats me!!!!!!

Documentation Sources

General

Interlisp Reference Manual

(October 1983 version, plus updates available on
{eris}<lispmanual>)

Release Notes (Chorus, Fugue 1 to 6, Carol, Harmony, Intermezzo)

Documentation files for LispUsers and LispLibrary Packages

Installation and use on a Dlion

1108 Users Guide

Mesa Users Guide (Chapter 2, Getting Started & Chapter 35, Othello)

{eris}<lisp>harmony>doc>Hello.tedit (PARC only)

{eris}<lisp>release>doc>GettingStarted.tedit (PARC only)

{eris}<lisp>harmony>doc>LocalFile.TEDIT

Installation and use on a Dolphin/Dorado

{eris}<lisp>release>doc>GettingStarted.tedit (PARC only)

{indigo}<altodocs> (PARC only)

Introductions (?)

Friendly DLion primer from LRDC

Sysdoc stuff?

Overview of the relevant sections of the IRM (October, 1983 version)

Sections 6.1, 18.16, & 18.17 ÿ files from lisp's point of view

Chapter 8 ÿ the P.A. including the history list

Chapter 9 ÿ Error handling and Breaks

Chapter 11 ÿ the File package including INIT file maintenance

Section 14.1 ÿ Sysouts

Section 14.2 ÿ GREET and INIT files

Section 14.3 ÿ Directories

Section 14.7 ÿ GAINSPACE when arrays full

Section 18.14 ÿ the keyboard

Section 18.18 ÿ Hardcopies

Section 18.20 ÿ Processes and the PSW

Section 19.20 ÿ Windows

Chapter 20 ÿ DEdit, TEdit, CHAT, Break windows, EDITBM, TTYIN

Basically, ignore the rest if you don't know how to program well.

Release Notes

Since the manual is constantly being made obsolete by new releases of the system, you should learn to use the release notes.

For each new release, study the release notes carefully trying to remember the things that have changed; just have to wade through looking for things that make sense from user's point of view. The index at front is a rough guide.

It is often best to consult the release notes before going to the IRM, since much of the information contained in the IRM will be out-of-date compared to the Release Notes.

Online help - the APROPOS function

In general, there is no online help facility in Interlisp-D.

There is one handy function, however, called APROPOS. APROPOS takes a single argument which is an arbitrary literal atom. (APROPOS LitAtom) will search through your virtual memory, looking for all atoms whose name contains the atom LitAtom. (APROPOS 'FLG) prints the names, values, and function definitions for all atoms in the current virtual memory that have FLG in their name. (APROPOS 'FONT) prints the names, values, and function arguments for all atoms in the current virtual memory that have FONT in their name.

Example, find the name of the variable that tells the name of the release (e.g., Harmony or Intermezzo) being used:

```
96_ (APROPOS (QUOTE RELEASE]
RELEASE.MONITORLOCK
- Function arglist: (LOCK EVENIFNOTMINE)
RELEASE.PUP - Function arglist: (EPKT)
RELEASERESOURCE
- Property list: (MACRO (ARGS (& &) (SUBPAIR
&
ARGS --)))
RELEASEBREAKWINDOW
- Function arglist: (BRKDS PREVIOUSDS)
RELEASE.XIP - Function arglist: (EPKT)
NIL
97_ (APROPOS 'NAME]
FILENAME - Function arglist: (NAME)
FILENAMEFIELD - Function arglist: (FILE FIELDNAME)
PACKFILENAME - Function arglist: U
NAMEFIELD - Function arglist: (FILE SUFFIXFLG DIRFLG)
FULLNAME - Function arglist: (X RECOG)
DIRECTORYNAME - Function arglist: (DIRNAME STRPTR CREATE?)
PACKFILENAME.STRING
- Function arglist: U
HOSTNAME - Function arglist: U
- Variable value: NIL
DIRECTORYNAMEP - Function arglist: (DIRNAME HOSTNAME)
HOSTNAMEP - Function arglist: (NAME)
TYPENAME - Function arglist: (DATUM)
```



```

STKNTHNAME      - Function arglist: (N POS)
STKNAME         - Function arglist: (POS)
SETSTKNAME      - Function arglist: (POS NAME)
STKARGNAME      - Function arglist: (N POS)
SETSTKARGNAME   - Function arglist: (N POS NAME)
COMPILEDTYPEP  - Function arglist: (X)
RENAMEFILE      - Function arglist: (OLDFILE NEWFILE)
UNPACKFILENAME - Function arglist: (FILE ONEFIELDFLG DIRFLG
STRING)
UNPACKFILENAME.STRING
                - Function arglist: (FILE ONEFIELDFLG
DIRFLG)
USERNAME        - Function arglist: (FLG STRPTR
PRESERVECASE)
                - Variable value: HALASZ
SETUSERNAME     - Function arglist: (NAME)
ALTOFILENAME    - Function arglist: (X)
TYPENAMEP      - Function arglist: (DATUM TYPE)
                - Property list: (DMACRO (X
(COMPILEDTYPEP X
)))
CHANGENAME1     - Function arglist: (DEF X Y)
CHANGENAME1A    - Function arglist: (DEF OLD NEW MAP)
PROPNAMEP      - Function arglist: (ATM)
ROOTFILENAME    - Function arglist: (NAME COMPFLG)
PROCESS.NAME    - Function arglist: (PROC NAME)
ETHERHOSTNAME   - Function arglist: (PORT USE.OCTAL.DEFAULT)
CANONICAL.HOSTNAME
                - Function arglist: (HOSTNAME)
GREETFILENAME   - Function arglist: (USER)
CHANGENAME      - Function arglist: (FN FROM TO)
FSTKNAME        - Function arglist: (POS)
FIRSTNAME       - Variable value: FRANK
FONTNAME        - Function arglist: (NAME)
                - Variable value: PARC
DIRFILENAME     - Function arglist: (FILEGROUP)
DIRPRINTNAME    - Function arglist: (FILEGROUP FLG)
UPPERCASEFILENAMES
                - Variable value: T
RESTORENAMES    - Function arglist: (FN)
MSHASHFILENAME  - Variable value: NIL
DEFAULTRENAMEMETHOD
                - Variable value: NIL
RECORDFIELDNAMES
                - Function arglist: (RECORDNAME FLG)
RENAME          - Function arglist: (OLD NEW TYPES FILES
METHOD)
RESOURCENAME    - Property list: (CLISPPWORD (FORWORD .
resourceName))
COMP.NAMEDLET   - Function arglist: (ARGS)
NAMEDLET        - Property list: (DMACRO COMP.NAMEDLET)
MSWORDNAME      - Function arglist: (X)
MOUSESTATE-NAME
                - Function arglist: (KEYNAME MOUSEONLYFLG)
NONSYSPPROPNAMEP
                - Function arglist: (ATM)
INSPECTABLEFIELDNAMES
                - Function arglist: (DECL TOPONLYFLG)
NAMEOFEDITW     - Function arglist: (NAME TYPE)
PARSE.NSNAME     - Function arglist: (NAME #PARTS)
DEFAULTDOMAIN)
NSNAMETYPE#     - Variable value: 89
                - Property list: (GLOBALVAR T)

```

```

JOBNAME          - Variable value: {LPT}LISPPRINT:PARC.;1
NSNAME          - Property list: (COURIERDEF (
COURIER.READ.NSNAME COURIER.WRITE.NSNAME
COURIER.NSNAME.LENGTH))
NSNAME.TO.STRING
                - Function arglist: (NSNAME FULLNAMEFLG)
COURIER.READ.NSNAME
                - Function arglist: (STREAM PROGRAM TYPE)
COURIER.WRITE.NSNAME
                - Function arglist: (STREAM NAME PROGRAM
TYPE)
COURIER.NSNAME.LENGTH
                - Function arglist: (NSNAME PROGRAM TYPE)
NSNAME2         - Property list: (COURIERDEF (
COURIER.READ.NSNAME COURIER.WRITE.NSNAME))
NS.SERVER.NAMES.TO.ADDRESSES
                - Variable value: NIL
EQUAL.CH.NAMES  - Function arglist: (NAME1 NAME2)
CH.NAME.TO.STRING
                - Function arglist: (NSNAME FULLNAMEFLG)
CANONICAL.CH.NAME
                - Function arglist: (NAME)
CH.CANONICAL.NAME
                - Function arglist: (NAME)
FONTNAME.IP     - Function arglist: (FONTDESC)
FLOPPY.NAME     - Function arglist: (NAME)
FLOPPY.SET.NAME
                - Function arglist: (NAME)
FLOPPY.GET.NAME
                - Function arglist: NIL
MAKESYSNAME    - Variable value: INTERMEZZO
MBUTTON.CHANGENAME
                - Function arglist: (TEXTOBJ OBJ NEWNAME)
GV.PORTFROMNAME
                - Function arglist: (SERVERNAME)
FULLUSERNAME    - Function arglist: (UNPACKEDFLG)
GV.NEWNAME      - Function arglist: (NAME GV.NEWNAME
IDENTIFYUSER PASSWORD)
GVNAMETYPE     - Variable value: 1
REGROOTNLSNAME - Variable value: "GrapevineRServer"
LA.SHORTFILENAME
                - Function arglist: (FILE EXT
KEEPVERSIONFLG)
DEFAULTMAILFOLDERNAME
                - Variable value: {DSK2}ACTIVE.MAIL
LA.LONGFILENAME
                - Function arglist: (FILENAME EXT)

PROFILEFILENAME
                - Function arglist: NIL
TOCFILENAME     - Function arglist: (MAILFILE)
PROMPTFORFILENAME
                - Function arglist: (WINDOW DEFAULT PROMPT)
LAFITEPROFILE.NAME
                - Variable value: LAFITE
LAFITETEMPFILEHOSTNAME
                - Variable value: CORE
LAFITE.READ.NAME.FIELD
                - Function arglist: (STREAM ARGS)
FB.FETCHFILENAME
                - Function arglist: (ENTRY)
FB.RENAMECOMMAND
                - Function arglist: (PREFIX FILEENTRY
WINDOW)

```

```

FB.STARTOFNAME - Function arglist: (FILENAME SPEC)
VTYPENAME - Function arglist: (DATUM)
DISPLAY/NAME - Function arglist: (ND)
                - Property list: (CODE {CCODEP}#65,43610)
FONTNAMELIST - Function arglist: (FONTDESC)
                - Property list: (CODE {CCODEP}#65,43524)
AR.USERNAME - Function arglist: NIL
AR.GET.FILENAME - Function arglist: (NUM PUTFLG)
AR.SUBMIT.NUM.FILE.NAME
                - Variable value:
{PHYLUM}<LISPARS>LISPARS.NUM
AR.FILENAME - Function arglist: (ARN)
AR.SUBMIT.FILE.NAME
                - Variable value:
{PHYLUM}<LISPARS>LISPARS.SUBMIT
AR.INFO.FILE.NAME
                - Variable value:
{PHYLUM}<LISPARS>LISPARS.TDS
AR.INDEX.DEFAULT.FILE.NAME
                - Variable value: {PHYLUM}<LISPARS>AR.INDEX
HASHFILENAME - Function arglist: (HASHFILE)
SKETCH.ELEMENT.TYPE.NAMES
                - Variable value: (MAP SKIMAGEOBJ GROUP)
TEXTBOX
BOX --)
SKETCH.ELEMENT.NAMEP
                - Function arglist: (X)
SK.UNDO.NAME - Function arglist: (HISTEVENT)
SK.GET.HARDCOPY.FILENAME
                - Function arglist: (SKW)

```

APROPOS can be very handy to go wandering around the system looking for a partially remembered variable or function name or for discovering what variables effect, e.g., FONTS.

Note the success of APROPOS at these tasks depends on the variables and functions be named in a sensible manner. A font-related variable named SIXTH-BASE would never be found by the (APROPOS 'FONT) function call. Such is not always the case in Interlisp-D.

Homework

Start reviewing the Lisp programming covered in the first six or so sessions.

LispCourse #23: What is Programming; Basic Lisp Revisited

What is Programming?

Procedures and Data: The Semantics of Programming

A program is basically a description of a sequence of *actions* (i.e., a *procedure*) to be carried out on a set of *objects*. In the computer world, the objects are different types of "information" and are thus called *data*.

For example: (PLUS 3 4) is a trivial Lisp program. The procedure PLUS takes the action of adding two numbers. The objects or data are the numbers, in this case 3 and 4.

Writing a program involves specifying two things:

- 1) a description of the procedure to be followed
- 2) a description of the data to be used.

The abstract description of the procedural component of a program is called its **control structure**. The abstract descriptions of the data components of a program are called its **data structures**.

Programming is all about control structures and data structures. Learning to program means learning to build combinations of control and data structures that accomplish target tasks effectively and efficiently.

For example:

Assume that the goal is to program up a simple database mapping people in ISL to serial numbers of their machines.

The first task is to determine a data structure for the database, e.g., a list of lists where each sublist begins with a name the second and subsequent items are serial numbers for that person's machine(s). Note that I would have to further specify the data structure for the name and serial numbers (e.g., they could be atoms or they could be lists,

depending on what operations I would want to perform on them).

The second task is to write the Lisp procedures to create, access, and modify the chosen data structure. This may feed back to modify the data structure – for example, you might discover that the procedures would be much simpler if the second element of each person’s list were the number of machines that the person had.

Lisp is a language for specifying the control and data structures of a program. Fortran, Basic, C, Pascal, Mesa, etc. are alternative languages for doing the same thing.

The same basic concepts of control structure and data structure appear in all of these languages. They differ only in how they express various control and data structures. Control/data structures that are easy to express in Lisp may be painfully hard to express in Fortran, and vice versa.

Lisp is unusual among these languages in that it blurs the distinction between procedures and data, both syntactically and semantically. A list [e.g., (PLUS 2 3)] that is a piece of data to one Lisp program, may be a description of a procedure that is executed by another Lisp program. Languages like Fortran and Pascal make a much cleaner distinction between procedure and data.

Blurring the distinction between procedure and data can be good or bad, depending on the task you are trying to program.

In this course, we’ll start out making clear distinctions between procedure and data in our Lisp programming style. Later, we may look at some of the advantages of blurring over this distinction a bit.

Procedure versus Process: Running a Program

A program is a *procedure*, i.e., a description of a sequence of actions to be carried out.

For the program to be useful, this description must be transformed into the actual sequence of actions being described. This transformation is commonly called "running the program".

The sequence of actual actions that take place when a program runs is called a *process*.

The entity that transforms the procedure description (i.e., program) into a running process is an *interpreter*.

At the bottom level, the ultimate interpreter of any program is the hardware of the computer on which the program is running. But in most languages (as in Lisp), there is another program (an interpreter) that is responsible for transforming the program from the higher level-language into the running process. This interpreter is the entity responsible for "understanding" the language and carrying out the requested actions.

Note that the interpreter is itself a program being interpreted (directly or indirectly) by the machine hardware.

Understanding Lisp requires understanding the Lisp interpreter, the program that transforms Lisp code into a running process. The nice thing about Lisp is that the interpreter is itself a Lisp program, making it easy to understand (and modify) if you know Lisp.

Seems a bit circular ÿ but its really a process of decomposition where you break Lisp down into simpler and simpler actions until you get to those actions the machine hardware can execute directly.

So, theoretically, there are three components to understand about programming: control structures, data structures, and interpreters. Unfortunately, this analysis skips all of the pragmatic aspects of programming in the real world.

Correctness, Efficiency, Maintainability, & Adaptability Quickly and with Minimal Effort: The Pragmatics of Programming

A good program is:

Correct ÿ it correctly accomplishes the task it is intended to

Efficient ÿ it make efficient use of the available resources; in particular, it works as quickly as possible using as few of the computer's resources (memory, disk space) as possible.

Maintainable ÿ it should be easy to debug and make minor changes to; in particular, it should be easy enough to understand that someone other than the original programmer can do the maintenance

Adaptable ÿ it should be easy to make large changes to the original program

Good programs must also be written **quickly and with minimal effort**: i.e., using as little programming time and as little programming effort as possible. A program that takes 1000 person-years to write will never get written.

Programming is a constant trade-off between these five goals since it is nearly impossible to satisfy all five simultaneously.

Interlisp takes a definite stand on which of these are important. In particular, correctness and efficiency are discarded in favor rapid-prototyping, i.e., programming quickly and with minimal effort.

Efficiency and correctness are achievable in Interlisp, but only extremely careful programming and close attention to issues of efficiency and exact correctness.

Maintainability and adaptability in Interlisp (and in most modern programming languages) are a matter of good programming style. If you follow the rules of good programming, then your programs will be maintainable and adaptable.

In this course we will focus in particular on these "rules of good programming", covering in detail the notions of abstraction and modularity. In the later part of the course, we will briefly consider the tools for writing efficient and correct programs in Interlisp.

Reviewing the Basics of Lisp Programming

The READ-EVAL-PRINT Loop

Lisp is in a continual loop in the Exec window

Read user input

Evaluate user input

Print result of evaluation

All work in Lisp is done in the Evaluation phase of this loop!

Basic Syntactic Building Blocks: Atoms and Lists

One defining feature of Lisp is that its syntax is trivial.

There are two basic building blocks of Lisp: *Atoms* and *Lists*.

Together Atoms and Lists are known as *S-expressions* (almost) everything in Lisp is an S-expression.

An *atom* is a symbol represented by one or more alphanumeric characters. Atoms are the "words" of Lisp; they provide a way in which to reference the actions and objects in Lisp world.

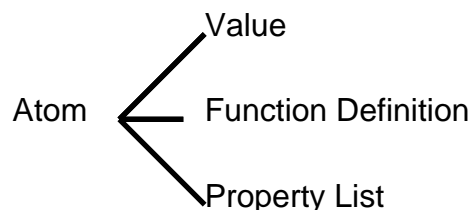
Examples: Sam, FOO, 123, A2233, VeryLongAtomName, Dashed-Atom.

A *list* is a "(", followed by zero or more atoms or lists, followed by a ")". Lists are the "sentences" of Lisp; they provide the structure to glue atoms together to make statements or to represent things.

Examples: (A B C); (A (FOO BAR) C); (SETQ A 5); (CAR (CAR A)).

Facts about atoms:

Every atom can reference three things: a *value*, a *function definition*, and a *property list*.



An atom's *value* is some other Lisp object; e.g., another atom, a list, a window, a process, etc.

Function definitions are discussed below.

An atom's *property list* is simply a list with a special format: $(Prop1 Value1 Prop2 Value2 \dots PropN ValueN)$, where each *PropI* is an atom and each *ValueI* is some other Lisp object; e.g., another atom, a list, a window, a process, etc.

When evaluated by the Lisp evaluator, an atom evaluates to its value.

Numeric atoms are atoms whose name consists of digits only. The value of a numeric atom is the atom itself.

Non-numeric atoms are called *litatoms*.

T and *NIL* are special atoms. The value of *T* is *T* and the value of *NIL* is *NIL*. *NIL* is also a list, in particular, the empty list $()$. *NIL* is the only thing that is both a list and an atom.

Facts about lists:

The first item in a list is called its *CAR*.

The rest of the list, after the *CAR* is removed, is called its *CDR*.

The operation of the Lisp evaluator w.r.t. lists is discussed in the next section.

That's all there is of Lisp syntax: all Lisp programs are built up from combinations of atoms and lists.

Forms and the Lisp Interpreter

The basic building block for procedures in Lisp is the *form* or *function call*. A form is simply a list structure, the *CAR* of which is the name of a function (i.e., an atom that references a function definition) and the *CDR* of which is a list of the arguments passed to the function.

Example: $(IDIFFERENCE 5 2)$, *IDIFFERENCE* is the function name and $(5 2)$ is a list of the arguments to be passed to the *IDIFFERENCE* function.

The Lisp evaluator evaluates forms in the following manner:

The CAR of the list is assumed to be the name of a function.

First, each element in the CDR of the list is evaluated, resulting in a list of evaluated arguments.

Then, the function named by the CAR is *applied* to the resulting evaluated argument list.

What it means to *apply* a function to an argument list is discussed in the following section.

Special Forms & QUOTE

There are some functions which are special in the sense that their arguments are NOT evaluated before they are executed. They are executed using their unevaluated arguments. [Called NLambda functions]

QUOTE is such a special form. QUOTE simply returns its unevaluated argument. QUOTE can be abbreviated by a '. (QUOTE A) is equivalent to 'A.

QUOTE is used to prevent evaluation where it is not required.

Example:

```
31_ (SETQ A 4)
```

```
4
```

```
32_ (SETQ B 5)
```

```
5
```

```
33_ (COPYFILE A B)
```

```
4: File not found
```

```
5: File not found
```

```
NIL
```

```
34_ (COPYFILE 'A 'B)
```

```
{PHYLUM}<HALASZ>B;5
```

EVAL

EVAL is a function which evaluates its arguments. Note that means that the arguments of EVAL are evaluated TWICE!

```
33_ (SETQ Harp 'Viola)
```

```
Viola
```

```
34_ (SETQ Viola 'Tuba)
```

```
Tuba
```

```
35_ Harp
```

```
Viola
```

```
36_ Viola
```

```
Tuba
```

```
37 _ (EVAL Harp) [means evaluate the value of Harp]
```

```
Tuba
```

Lisp Control Structures

Basically, a Lisp procedure is a list of forms to be evaluated one after the other in sequence.

Example, the following might be considered a Lisp procedure:

```
(SETQ A 55)
```

```
(SETQ B 66)
```

```
(IPLUS A B)
```

Note a sequence of actions is a very limited control structure. Using this control structure, it is impossible to specify an action to be done only under certain circumstances or to be repeated a variable number of times.

Interlisp has many special forms that provide more interesting control structures. We covered two: COND and Iterative Loops.

COND

COND is a special form that implements a conditional control structure.

COND has the following form:

```
(COND
```

```
  (Test1 Consequents1)
```

(*Test2 Consequents2*)
 (*Test2 Consequents2*)
 ...
 (*TestN ConsequentsN*)

Each *Testi* is an S-expression (usually a predicate) that evaluates to NIL or non-NIL. Each *Consequentsi* consists of 0 or more S-expressions.

COND works as follows:

Test1 is evaluated.

If it returns a non-NIL value, each S-expression in Consequents1 is evaluated in turn, and then the COND is exited. The value of the COND is the value of the last S-expression in Consequents1.

If it returns a NIL value, go on to Test2

Test2 is evaluated.

If it returns a non-NIL value, each S-expression in Consequents2 is evaluated in turn, and then the COND is exited. The value of the COND is the value of the last S-expression in Consequents2.

If it returns a NIL value, go on to Test3.

...

TestN is evaluated.

If it returns a non-NIL value, each S-expression in ConsequentsN is evaluated in turn, and then the COND is exited. The value of the COND is the value of the last S-expression in ConsequentsN.

If it returns a NIL value, the COND is exited with a value of NIL.

Example:

Return X if X is an atom, NIL otherwise.

```
(COND
  ((LITATOM X) X)
  ((NUMBERP X) X))))
```

Iterative control structures: The FOR Loop and its cousins

Iteration is a control structure that makes it possible to repeat the same operation on each element in a sequence (list) of things.

"To iterate" means "to repeat".

The FOR Loop

The major iterative construct in Interlisp is the FOR loop.

[Footnote: The FOR loop construct is not a standard part of most Lisps. The iterative control structure is available in all Lisps, it just has a different syntax than the Interlisp FOR loop.]

The FOR loop has the following form:

(FOR *variable* IN *list* DO *operation*)

Note: FOR is a special form; the elements of the CDR are not evaluated automatically.

IN and DO are keywords.

Variable is unevaluated. It is the name of an atom to be used as the local variable in the iteration.

List is evaluated. It is a S-expression whose value is a list.

Operation consists of 0 or more S-expressions to be evaluated. Ordinarily, these S-expression make some use of the value of *the* atom in the *variable* role.

FOR works as follows:

The *variable* is bound (i.e., temporarily SETQed) to the CAR of the *list*. The S-expressions in *operation* are then evaluated.

Then the *variable* is bound to the second item in the *list* and the the S-expressions in *operation* are again evaluated.

Then the *variable* is bound to the third item in the *list* and the S-expressions in *operation* are evaluated.

...

The *variable* is bound to the last item in the *list* and the S-expressions in *operation* are evaluated.

The FOR loop returns NIL.

Example:

```
(FOR Window in (OPENWINDOWS) DO (CLOSEW
Window))
```

Alternative FOR loops

1. DO versus COLLECT

The DO keyword can be replaced by the COLLECT keyword in the FOR loop. In this case, on every iteration the value of the last S-expression in operation will be saved. The FOR loop will then return a list of these values (in order) instead of returning NIL.

Example:

```
4_ (FOR Item IN '(A B)(C D)(E F) COLLECT (CAR
Item)(CDR Item))
((B)(D)(F))
```

2. "IN list" versus "FROM n TO m BY k"

FOR also allows for iteration over a sequence of numbers. To do this, replace the "IN list" construction with the construction "FROM n TO m BY k", where n, m, and k evaluate to numbers.

Note: The "BY k" is optional and defaults to "BY 1" if it is not specified.

Example:

```
5_ (FOR N FROM 1 TO 10 COLLECT (PLUS N
6))
(7 8 9 10 11 12 13 14 15 16)
```

WHILE and UNTIL loops: Alternatives to FOR

WHILE and UNTIL loops allow repetitive operations without an explicit sequence.

WHILE has the form:

(WHILE *predicate* {DO, COLLECT} *operations*)

Note: WHILE is a special form.

Predicate is evaluated. It is an S-expression that evaluates to NIL or non-NIL.

Operations is 0 or more S-expression to be evaluated.

The WHILE loop repeatedly evaluates *operations* as long as *predicate* evaluates to non-NIL.

DO versus COLLECT works exactly as in FOR. If DO is used, then the WHILE loop always returns NIL. If COLLECT is used, the WHILE loop returns a list containing, in order, the value of the last S-expression in *operations* from each iteration.

Example:

```
11_ (WHILE (MouseButtonDown) COLLECT
      (WHICHW))
```

UNTIL is similar to while, but it iterates until its *predicate* becomes non-NIL, i.e., as long as its *predicate* is NIL.

"UNTIL *predicate*" is equivalent to "WHILE (NULL *predicate*)".

Defining and Applying Functions

All of the work in Lisp is done when the Lisp evaluator evaluates a form or function call.

Recall that evaluating a form involves applying a *function* to a list of evaluated arguments.

A function is simply a sequence of forms that have been packaged into a unit and named.

Once packaged into a function, the sequence of forms can be manipulated as a single entity. The process of treating a sequence of forms as a single entity that carries out a single (but compound) action is known as *procedural abstraction*. We'll have more to say about procedural abstraction later in the course.

In Lisp packaging and naming a sequence of forms is known as *defining a function*. Programming in Lisp consists of defining functions that call other functions you have already defined (or will define before you run the program).

Defining functions

DEFINEQ is a special form that allows you to define functions in Lisp. It has the form:

(DEFINEQ *Defn1 Defn2 ...*)

Each *Defni* is the name and definition of a function and has the form: **(*FunctionName FunctionDefn*)**.

FunctionName is any atom.

FunctionDefn has the form:

**(LAMBDA *ParameterList*
FunctionBody)**

LAMBDA is a keyword indicating that the list is function a definition. Just put it there. It is an historical remnant from Church's Lambda calculus, on which Lisp was originally built.

ParameterList is a list of the parameters (arguments) for the function.

FunctionBody is 1 or more Lisp forms.

Example:


```
(DEFINEQ
  (SumOfSquares      [FunctionName]
    (LAMBDA [Keyword]
      (X Y) [ParameterList]
        (PLUS (TIMES X X)(TIMES Y
              Y) [FunctionBody]
```

Applying functions

Recall that the final step in evaluating a form involves *applying* the function named by the CAR to the list of evaluated arguments.

APPLY works as follows:

1. The value of each parameter in the function definition is (temporarily) set to the corresponding element of the evaluated argument list.
2. All of the forms of the function body are evaluated in turn using the normal rules of Lisp evaluation.
3. The value returned by the function is the value of the last (only) form in the function body.
4. All parameters (i.e., L) are set to back to their original value.

The process of setting the values of the parameters to the values of the arguments is known as *binding* the local variables. Note that binding affects the parameters only locally within the function. The value of the same atom outside of the function is unaffected.

Lisp Interpreter: A combination of EVAL and APPLY

EVAL and APPLY combine to produce the Lisp interpreter, as the following example illustrates:

Evaluating the form (SumOfSquares (TIMES 2 3) 2)

1. Each item in the CDR of the form is **evaluated** in turn:

1.1) (*TIMES 2 1*) evaluates to 6 (by recursive call to the Lisp evaluation of a form).

1.2) 2 evaluates to itself.

2. The SumOfSquares function is **applied** to the list (6 2).

2.1) X is bound to 6 and Y is bound to 2.

2.2) (PLUS (TIMES X X)(TIMES Y Y)) is evaluated by recursive call to the Lisp evaluation of forms, resulting in the value 40.

2.3) X and Y are rebound to their previous values (if any).

2.4) SumOfSquares is exited returning, 40.

3. The value of the form (40) is printed.

Lists: the primary Lisp data structure

The primary data structure in Lisp is the list. When you want to represent an object or piece of data in Lisp, the first thing you think of is a list!!!

There are other data structures in Lisp, but lists are the only ones we covered in the earlier part of the course.

Lists can be used to represent almost any data:

Examples:

A person's name might be a list of three atoms: (*First Middle Last*) as in (*Frank Geza Halasz*) or (*John Seely Brown*).

ISL-People might be represented by a list of names (which are themselves lists): *((John Seely Brown) (Dana ?? Bloomberg)(Thomas P. Moran)(Frank Geza Halasz))*

PARC-Personnel might be a further aggregation of lists representing each lab. Each lab list would consist of two items, a lab name and a list of lab people: *(PARC (ISL ((John Seely Brown) (Dana ?? Bloomberg)(Thomas P. Moran)(Frank Geza Halasz)))(CSL (Robert ?? Ritchie)(Robert ?? Haggman) ...)(SCL (Adele ?? Goldberg) ...))*

List Manipulation Functions

Interlisp has lots and lots of functions for creating, maintaining and decomposing these list data structures. Some of those we covered earlier (See LispCourses #2 & #3) were:

List Decomposition Functions

CAR ž returns the first element of the list

CDR ž returns the list minus the first element

CxxR (where x=A or D) ž compositions of CAR and CDR.

NTH ž returns the list *tail* starting at the Nth element

LAST ž returns the list tail containing only the last element

List Composition Functions

APPEND ž returns the concatenation of two or more lists

LIST ž creates a list consisting of its arguments

CONS ž (CONS Arg1 Arg2) returns a list with Arg1 as its CAR and Arg2 as its CDR.

List Manipulation Functions

REVERSE ž returns list with reverse order of top-level elements

LENGTH ž returns number of top-level elements in list

List Formats

Each of the lists given in as examples above has a format that determines the meaning of the elements of the list. Programs that might use these lists would have to have built into them the necessary procedures for making use of these formats.

For example, the name list has three elements representing the first, middle and last names. If a program needs the first name of a person, the programmer would have to know enough about the format to take the CAR of the person's name list.

The examples given above were ad-hoc list formats. Much of Lisp programming involves creating (and documenting) such ad-hoc list formats. But, there are some special list formats that are commonly used as data structures in Lisp:

An **ASSOC list** has the following form: $((key1\ data1)(key2\ data2)(key3\ data3)(key4\ data4) \dots (keyn\ datan))$, where each *key_i* is an atom and each *data_i* consists of 0 or more S-expressions.

The function **ASSOC** is used to retrieve items from an ASSOC list. ASSOC takes two arguments, a key and an ASSOC list. It returns the first *key-data* pair in the list whose key is equal to the key argument.

A **Prop list** has the form: $(prop1\ value1\ prop2\ value2\ prop3\ value3 \dots propn\ valuen)$, where each *prop_i* is an atom and each *value_i* is exactly one S-expression.

The function **LISTGET** retrieves a prop values from a PROP list. LISTGET takes 2 arguments, a prop list and a prop. It returns the value corresponding to prop on the prop list.

LISTPUT adds a prop-value pair to an already existing prop list. Its three arguments are a prop list, a prop, and a value. If the prop is already on the list, it simply updates its value. It returns the value.

Exercise: Learning Data Abstraction

Write a program that manipulates a database of people in ISL, their office numbers and their phone numbers.

First, determine a data structure for your database.

Then write functions for:

creating the database

adding people to the database

getting a list of all the first names in ISL

getting a list of all the last names in ISL

getting a list of all the phone numbers in ISL.

(Hint: you will find FOR-COLLECT loops very handy here.)

References

LispCourse Notes #2 thru #6.

LispCourse #24: Data Abstraction

Programs as Representations of the Real World

Extensionally, computer programs carry out a sequence of information manipulating actions on a set of computational objects called data.

Intensionally, computer programs the actions and objects are usually designed to be some sort of model of some "real world" actions and objects. In other words, computer programs are *representations* of real world actions and objects.

Data is used to represent real world objects. Data structures represent the structure of real world objects.

In talking about computer programs, we often blur the distinction between data structures and the real world objects which they represent.

Data, Compound Data & Data Structures

Atoms are the *primitive data* structures in Lisp.

Some objects can be represented by atoms alone; e.g., numbers by NUMBERPs and English words by LITATOMS.

However, most objects are more complicated and can only be represented by a combination of many simpler pieces of data. In Lisp, this combination is usually achieved using lists that "glue" together atoms and other lists. A list is *compound data*.

A *data structure* is a "scheme" for using compound data to represent a complex object.

Example:

A persons name is an object with some structure, i.e., it has a first name, a middle initial, and a last name. In Lisp, we might represent this structure using a list data structure with three elements. where the first element was an atom representing the first name, tyhe second element an atom representing the middle initial, and the third element an atom representing the last name.

Compound data is important because it allows us to deal with the many pieces of simpler data as a single entity, just as we deal with the real world complicated object as a single entity.

Data structures are important because they allow us to take a very simple and straightforward compounding scheme (i.e., lists) and represent very complex objects. The data structure provides the "rules" by which to interpret a list structure into representation of a complex object.

Data Abstraction: Dealing with Compound Data

Dealing with compound data and data structures can lead to difficult programming if the proper rules are not followed.

The Need for Data Abstraction

Example 1

Consider the following programming problem:

write a set of functions that fill out tax forms. There will be one function for each tax form. On each tax form, you need to fill in the person's name at least once and sometimes two or three times . The name is passed to you as a list of three items of the form (*First Middle Last*).

How should you handle the placing of names on the tax forms?

One solution is the following: every time you need to print the persons name, print the CAR of the name list, then the CADR of the name list and then the CADDR of the name list.

What happens if for some reason you start getting names in the form (*Last First Middle*).

You would have to go through each function looking for each place that you print out the person's name and change the procedure to be CADR of name list followed by CADDR of name

list followed by CAR of name list. UGH!! This could be a god-awful job.

BUT - what if you had done the following to begin with: Write three functions called **FirstName**, **MiddleName** and **LastName** that take a name and return the indicated part of the name. **FirstName** would simply take the CAR of the NameList, the **MiddleName** function the CADR and the **LastName** the CADDR.

Then every time you want to print the person's name, you would print (*FirstName NameList*) followed by (*MiddleName NameList*) followed by (*LastName NameList*). Everything would work as in the initial case above.

However, when the name format change can along, you would have to change only the three functions **FirstName**, **MiddleName** and **LastName** to use CADR, CADDR, and CAR respectively.

None of the tax form function would have to change at all!!!!
Thus the change in name formats would be trivial.

Example 2

Consider the following programming problem:

You are writing the program for filling out Schedule G (Income Averaging). You have a record of the person's last 4 years' tax liabilities in the form (*NameList Year-1 Year-2 Year-3 Year-4*). In your calculations, you need the Year-i tax liability.

Solution: you write four functions called **Year-1** thru **Year-4** that take the liability list and return the Year-I tax liability. You implement these functions by taking the (CAR (NTH LiabilityList N)) for the correct N in each case.

BUT - what if the liability list format changed a bit to be (*LastName Firstname Year-1 Year-2 Year-3 Year-4*). Then you would have to go back and change all four functions.

However, if you had written a function called **ListOfLiabilities** that took the liability record and returned a list of the four liabilities. Then you could have written the **Year-I** functions to use the value returned by **ListOfLiabilities**.

If you did this, then when the liability list format changed, you would have had to change only one function, **ListOfLiabilities**, instead of four.

Not a big deal, but if there were 100 variables instead of 4 it might be a big deal.

Data Abstraction

What do these examples show? The need for data abstraction to improve program maintainability and adaptability.

Data abstraction is a programming design methodology in which:

- 1) You carefully separate functions that **USE** data from functions that **ACCESS** and manipulate data structures. Changes to the data structures will then affect only the data structure functions and not the data using functions.
- 2) You carefully separate functions dealing with the various levels of a compound data structure. Changes to one level will then effect only the functions dealing with that level of the compound data and not all other levels.

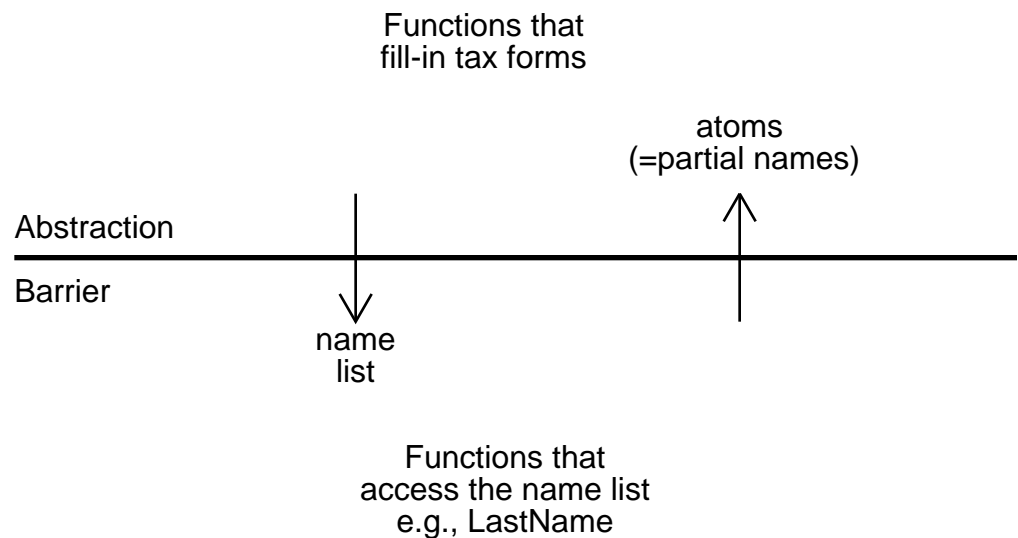
Abstraction Barriers

Data abstraction is accomplished by creating *abstraction barriers* between function that use data and functions that access data AND between the functions that access various levels of data from a compound structure.

An abstraction barrier is a line that separates the functions of a program. Functions on one side of the line can make no (unnecessary) assumptions about the data structures used by functions on the other side of the line, except as described in the well-defined interface between the two sets of functions.

Example

In the tax forms/name list example above, an abstraction barrier was required between the form filling programs and the name list data structure.



The form filling programs should not make any assumptions about the format of the name list (or even that it is a list instead of an atom or a string or whatever).

Instead, if they need to access the name list structure, they should call the functions (FirstName, LastName, etc) on the other side of the abstraction barrier that deal specifically with accessing the name list structure.

There should be a well defined protocol that states that whenever a data using function (i.e., one that fills in a tax form) calls an access function (e.g., LastName), it passes down a name thing and gets back an atom that represent the first, middle or last name, as indicated.

How this is to be accomplished is entirely up to the access function.

The using function should make no assumptions whatsoever about the format of the name thing.

Similarly, the access functions should make no assumptions about how the atom they return will be used.

Constructing Access Functions: Constructors, Selectors & Mutators

The trick in writing a program with strong abstraction barriers is to write a good set of access functions for isolating the access to your data structures from the rest of your program (and for isolating the access to low levels of the data structure from functions that deal with higher levels of the data structure).

These access functions serve as the interface between the rest of your program and your data structures. If your program wants access to the data structures, it has to call one of these access functions.

The access functions take data in an agreed upon form and do whatever work is necessary to translate it into a form compatible with the chosen data structures.

They also take data from the data structures and do whatever work is necessary to translate it into an agreed upon form to pass back to the rest of the program.

Constructors, Selectors and Mutators

There are basically three kinds of access functions: **constructors**, **selectors** and **mutators**.

Constructors take individual pieces of simpler data and build a data structure from them, returning the data structure.

Example: (**MakeName** *First Middle Last*) takes three atoms and returns a name data structure.

Selectors take a compound data structure and return individual pieces of data from that data structure.

Example: (**FirstName** *NameList*) takes a name data structure as an argument and returns the first name portion of that name data structure.

Mutators take a compound data structure and alter individual pieces of data in that data structure.

Example: (**ChangeFirstName** *NewFirstName*) takes a name data structure as an argument and returns the same data structure with the first name portion altered to be *NewFirstName*.

A complete set of access functions for a name data structure might be the following:

Constructor: **MakeName** takes three atoms representing the first, middle and last names and returns a name object.

Selectors: **FirstName** takes a name object and returns the first name. **LastName** and **MiddleName** do the corresponding thing.

Mutators: **ChangeFirstName** takes a name object and returns the altered name object. **ChangeLastName** and **ChangeMiddleName** do the corresponding thing.

In a program with good data abstraction, any function that wanted to create a new name would have to call **MakeName**. Any function that wanted to access part of the full name would have to call the appropriate selector function. Any function that wanted to change part of the name would have to call the appropriate mutator function.

Other than these seven functions NO other functions in the program could make any assumptions about the format of the name object.

Example: Problem from LispCourse #23

Attached is a solution to the problem at the end of LispCourse#23 that makes strong use of data abstraction, both to separate the use functions from the access functions AND to separate the access functions that deal with different levels of the database data structure.

A diagram of the abstraction barriers in this program is the following:

Database

Database Access Functions

Database Entry

Database Entry Access Functions

Name	Office Number	Phone Number
Name Access Functions	Office Number Access Functions	Phone Number Access Functions

References

Winston & Horn, Chapter 5, pages 97 thru 100

Sussman & Abelson, Chapter 2, pages 71 thru 88 (and beyond)

Exercises

Rewrite your program from LispCourse #23 using good principles of data abstraction. Try changing the format of you database entries and seeing how many functions you have to change to adapt to this format change.

Do this *before* you study in detail the attached example program.

LispCourse #25: The Record Package; The Inspector

Records: Data Abstraction in Interlisp

There is a package in Interlisp called the Record Package that makes data abstraction syntactically easy. The Record Package provides a simple syntax that supports writing constructors, selectors, and mutators.

The Record Package is part of CLISP ž statements written in the Record Package syntax are translated by the Lisp interpreter (or compiler) into standard Lisp and then executed.

All of the syntax described here is just a "pretty" way of stating in CLISP what you could do directly, but less clearly, with CARs, CDRs, GETPROPs, ASSOCS, etc. in straight Interlisp.

The Record Package actually includes an interface to two related functionalities in Interlisp Ÿ records and datatypes. We will discuss records first and datatypes a bit later.

Records: Basic Stuff

A *record* is basically a description of a list structure. The description names the whole structure (i.e., the record) **and** names each of the parts of the structure.

In the terminology of the Record Package, each part of the overall record is called a *field* of the record.

Once you have described a record structure, you can create instances of the record and can access any field of these instances *by name* using the special Record Package syntax.

Example:

(fetch (Message Header) of NextMessage) is a statement that retrieves the Header part of a Message record that is the value of the atom NextMessage.

Important note: the word *record* is used to denote both the *description* of the list structure and the actual lists that are *instances* of the structure described. Which of these is intended should be clear from the context.

Record Declarations

Before a record can be used it must be declared.

A **record declaration statement** names the record and describes all of the fields in the record. It has the form:

(RECORD *RecordName Fields ExtraStuff1 ExtraStuff2 ...*)

RECORD is a keyword indicating that this is a record declaration.

RecordName is the name of the record. RecordNames must be unique in the whole system (among both records and datatypes).

Fields is a list of the parts (i.e., fields) of the record. Each element in the list is a non-NIL atom that serves as the name of the field.

The list can also contain NILs, which stand for unnamed (and therefore unaccessible) fields. Finally, the list can contain integers, which stand for the specified number of unnamed fields.

The *ExtraStuff1* statements are optional. If present, each *ExtraStuff1* can be any of several kinds of information.

Most importantly, the *ExtraStuff1* can be an assignment statement of the form:

FieldName _ Form

FieldName is one of the named fields from the *Fields* list.

Form is any Lisp form.

This assignment statement specifies the default value for the field named *FieldName* in the record.

When you create an instance of the record, if *FieldName* is not explicitly given a value in the CREATE statement, then the value returned by evaluating *Form* will be used to fill-in the field.

Examples:

```
(RECORD PersonsName (First Middle Last)
  First _ 'John Middle _ 'Dunce Last _ 'Doe)
```

```
(RECORD DatabaseEntry (Name OfficeNumber
  PhoneNumber))
```

Note: A record declaration statement is NOT an executable statement i.e., it doesn't evaluate to anything. It just serves to describe the record to CLISP, which will use this information to translate the executable statements that access instances of this record.

Constructing Record Instances: The CREATE statement.

Once a record has been declared, you can construct instances of the record using the CREATE statement.

The CREATE statement has the following format:

```
(CREATE RecordName Assignment1 Assignment2 ...)
```

RecordName is the name of the record you want to create an instance of.

The *Assignment1* statements are optional. If present, each *Assignment1* is a statement that specifies the value to be given to a particular field of the record when it is created. This statement should have the format:

```
FieldName _ Form
```

FieldName is the name of one of the fields of *RecordName*.

Form is any Interlisp form, the value of which will be placed in the field *FieldName* when the record is created.

When the CREATE statement is evaluated, it returns a list that is an instance of the specified record. The value of each field in the record is determined as follows:

If there was an assignment statement for that field in the CREATE statement, then the value of the form in that assignment statement is used.

Otherwise, if there was default assignment statement for that field in the record declaration for the record, then the value of the form in that assignment statement is used.

Otherwise, the field is set to NIL.

Examples:

```
30_(RECORD PersonsName (First Middle Last)
```

```
First _ 'John Last _ 'Doe)
```

```
PersonsName
```

```
31_(CREATE PersonsName)
```

```
(John NIL Doe)
```

```
32_(CREATE PersonsName First _ 'Sam)
```

```
(Sam NIL Doe)
```

```
33_(CREATE PersonsName First _ 'Sam Last _ 'Smith
```

```
Middle _ (CAR (LIST 'A. 'B.)))
```

```
(Sam A. Smith)
```

Selecting Fields in a Record: The *fetch* statement

Given a record (instance), you can select any of its fields using the *fetch* statement.

The *fetch* has the following format:

(*fetch* (*RecordName* *FieldName*) of *Form*)

fetch and ***of*** are keywords.

RecordName is the name of the record (description) being used.

FieldName is the name of the field to be selected from the record instance of type *RecordName*.

Form is an Interlisp form that evaluates to a record of type *RecordName*.

When the *fetch* statement is evaluated, it will return the value of the named field from the record that is the value of the given form.

Examples:

```
33_(SETQ Person (CREATE PersonsName First _ 'Sam Last _
'Smith
Middle _ 'A))
```

(Sam A. Smith)

```
34_Person
```

(Sam A. Smith)

```
35_ (fetch (PersonsName Last) of Person)
```

Smith

```
36_ (fetch (PersonsName First) of Person)
```

Sam

```
37_ (fetch (PersonsName First) of (CREATE PersonsName))
```

John

Note: if the *FieldName* is unambiguous, ***FieldName*** can be used in place of **(*RecordName* *FieldName*)**. *FieldName* is unambiguous if there is only one record *in the entire system* having a field with that name.

Opinion: Using *FieldName* instead of (*RecordName FieldName*) is just lousy programming style because it can lead to lots of trouble when you (or more likely someone else) adds a second record to the system that just happens to use *FieldName*.

Mutating Fields in a Record: The *replace* statement

Given a record (instance), you can mutate (i.e., change) any of its fields using the *replace* statement.

The *replace* has the following format:

(*replace* (*RecordName FieldName*) of *Form1* with *Form2*)

replace*, *of*, and *with are keywords.

RecordName is the name of the record (description) being used.

FieldName is the name of the field to be changed from the record instance of type *RecordName*.

Form1 is an Interlisp form that evaluates to a record of type *RecordName*.

Form2 is an Interlisp form whose value will be used as the new value of the field.

When the *replace* statement is evaluated, it will replace the old value of the specified field in the specified record (i.e., the record returned by evaluating *Form1*) using the specified new value (i.e., the value returned by evaluating *Form2*). The value returned by the *replace* statement is the new value.

Examples:

38_Person

(*Sam A Smith*)

39_ (*replace* (*PersonsName Last*) of *Person* with 'Jones)

Jones

40_ (*fetch* (*PersonsName Last*) of *Person*)

```
Jones
41_Person
(Sam A Jones)
42_(replace (PersonsName First) of Person with 'Jane)
Jane
43_(replace (PersonsName Middle) of Person with 'Maria)
Maria
44_Person
(Jane Maria Jones)
```

Records as Data Abstraction

Since records allow you access to data structures by *name* rather than by *structure*, they (to a great extent) isolate the program from changes in the underlying data structure.

Consider the alternative record definitions: (*RECORD Name (First Middle Last)*) and (*RECORD Name (Last First Middle)*) and (*RECORD Name (SS# BirthPlace Last Middle First)*)

CREATE, fetch, and replace statements for all of these would be identical.

For example: (*fetch (Name First) of NewName*) would always return the first name field, although it would be the CAR of the first type of record, the CADR of the second type of record, and the 5th element of the third type of record.

Therefore changing the underlying record structure between any of these three record declarations (and many more alternative declarations) would have no effect on the programs that accessed the records using CREATE, fetch, and replace.

Hierarchical Data Structures

Often data structures are hierarchical, with smaller data structures embedded in more global data structures.

An example from last time is the DatabaseEntry list structure. Each DatabaseEntry consisted of three items: a name, a phone number, and an office number. Each of these three items was in turn a list structure consisting of two or three atoms.

```
DatabaseEntry: List of three items
  Name: List of three items
    First: atom
    Middle: atom
    Last: atom
  PhoneNumber: List of two items
    Prefix: atom
    Extension: atom
  OfficeNumber: List of two items
    BldgNumber: atom
    RoomNumber: atom
```

To deal with this hierarchical data structure using the Record Package, you would define four independent records for DatabaseEntry, Name, PhoneNumber, and OfficeNumber.

For example:

```
(RECORD DatabaseEntry (Name PhoneNumber OfficeNumber))
(RECORD Name (First Middle Last))
(RECORD PhoneNumber (Prefix Extension) Prefix _ 494)
(RECORD OfficeNumber (Bldg RoomNumber) Bldg _ 35)
```

To select the Extension or RoomNumber from a given database entry you could use embedded **fetch** statements:

Example:

```
45_ Entry
((Sam A Smith)(494 4431)(35 2319))
46_ (fetch (PhoneNumber Extension) of
      (fetch (DatabaseEntry PhoneNumber) of Entry))
4431
```

```
47_ (fetch (OfficeNumber RoomNumber) of
      (fetch (DatabaseEntry OfficeNumber) of Entry))
2319
```

The Record Package provides a shorthand syntax for these embedded access (i.e., **fetch** and **replace**) statements, provided that you have named the fields of the various records correctly.

In particular, if the field name of the *embedding* record is the same as the record name of the *embedded* record, then you can combine the **fetch** statements by using a *path name* as the field specification in a single fetch statement. This *path name* should name the highest level record followed by the field that contains the second level record followed by the field that contains the third level record and so on until the target field is named.

Examples:

```
48_Entry
((Sam A Smith)(494 4431)(35 2319))
49_ (fetch (DatabaseEntry PhoneNumber Extension) of Entry)
4431
50_ (fetch (DatabaseEntry OfficeNumber RoomNumber) of
      Entry))
2319
```

Example that doesn't work with the shorthand syntax because the embedding record field name is not the same as the embedded record name:

```
60_(RECORD DatabaseEntry (Name Phone OfficeNumber))
DatabaseEntry
61_(RECORD PhoneNumber (Prefix Extension) Prefix _ 494)
PhoneNumber
62_(SETQ Entry
      (CREATE DatabaseEntry Name _ 'Foo
                Phone _ (CREATE PhoneNumber
                          Extension _ 4456)
```

OfficeNumber _ 99))

(Foo (494 4456) 99)

63_(fetch (DatabaseEntry Phone Extension) of Entry)

no such record path

at ... (DatabaseEntry Phone Extension) of Entry)

in (fetch (DatabaseEntry Phone Extension) of Entry)

UNDEFINED CAR OF FORM

fetch

64_(fetch (DatabaseEntry PhoneNumber Extension) of Entry)

no such record path

at ... (DatabaseEntry PhoneNumber Extension) of Entry)

in (fetch (DatabaseEntry PhoneNumber Extension) of Entry)

UNDEFINED CAR OF FORM

fetch

65_(fetch (PhoneNumber Extension) of

(fetch (DatabaseEntry Phone) of Entry))

4456

Records constructed from different list structures

The (RECORD ...) statement declares a record data structure that is to be constructed from an ordinary list. Each field in the record declaration is to be an element in that list.

There are many alternative list structures that can be constructed using the Record Package. These alternative list structures are declared, created, and accessed using the same statements as are RECORDs, except that the declaration statement starts with the given record type instead of RECORD.

Some of the most important alternative list structures are:

TYPERECORD ž just like a RECORD except that the first element of the list is always the name of record.

Example:

```
51_ (TYPERECORD Name (First Middle Last))
Name
52_ (CREATE Name
      First _ 'Sam Middle _ 'Ishikawa Last _
      'Watanabe)
(Name Sam Ishikawa Watanabe)
53_ (fetch (Name First) of IT)
Sam
```

ASSOCRECORD ž the list created is in ASSOC list format with the field names as keys.

Example:

```
54_ (ASSOCRECORD Name (First Middle Last))
Name
55_ (CREATE Name
      First _ 'Sam Middle _ 'Ishikawa Last _
      'Watanabe)
((First . Sam)(Middle . Ishikawa)(Last . Watanabe))
56_ (fetch (Name First) of IT)
Sam
```

PROPRECORD ž the list created is in Prop list format with the field names as props.

Example:

```
57_ (PROPRECORD Name (First Middle Last))
Name
58_ (CREATE Name
      First _ 'Sam Middle _ 'Ishikawa Last _
      'Watanabe)
(First Sam Middle Ishikawa Last Watanabe)
59_ (fetch (Name First) of IT)
Sam
```

Determining the record type of an object: The **TYPE?** statement

There is a fourth tool (in addition to constructors, selectors, and mutators) that is very handy in writing programs with data abstraction ž the **predicator**.

A predicator for a given data structure is a predicate that returns a non-NIL value if its argument is an example of the data structure and NIL otherwise.

For example, a predicator for the name list from last week might be:

```
(DEFINEQ (LC.NameP
  (LAMBDA (List)
    (* * Check if List is a name list structure)
    (AND
      (LISTP List)
      (EQUAL (LENGTH List) 3)
      (LITAOM (CAR List))
      (LITATOM (CADR List))
      (LITATOM (CADDR List))))))
```

The **TYPE?** statement is the Record Package mechanism for dealing with predicators.

To use the **TYPE?** mechanism, you must add a predicator as one of the *ExtraStuffI* arguments in the record declaration statement for each record. The argument must have the format:

(TYPE? Form)

TYPE? is a keyword.

Form can be an Interlisp expression using the variable DATUM.

DATUM is bound to the record instance to be tested and then the expression is evaluated, returning a NIL or a non-NIL value. Form can also be an atom which is the name of a function having one argument, the record instance to be tested. In this case, the function is applied to the record instance to get the NIL or non-NIL value.

Example:

```
(RECORD Name (First Middle Last)
  (TYPE?
    (AND
      (LITATOM (fetch (Name First) of
        DATUM))
      (LITATOM (fetch (Name Middle) of
        DATUM))
      (LITATOM (fetch (Name Last) of
        DATUM))))))
```

Once you have a TYPE? clause in the record declaration for a record, you can ask of any Lisp object whether it is an instance of that record type. To do this, use the **TYPE?** statement.

The **TYPE?** statement has the following format:

(TYPE? RecordName Form)

TYPE? is a keyword.

RecordName is the name of an already declared record type.

Form is any Interlisp form.

When the **TYPE?** statement is evaluated, the TYPE? entry is retrieved from the record declaration for *RecordName* and applied (as described under the TYPE?

clause above) to the value returned by evaluating *Form*. The value of the **TYPE?** statement is the value of this application.

For example, the record declarations for the database entries might be amended as follows:

```
(RECORD DatabaseEntry (Name PhoneNumber OfficeNumber)
  (TYPE?
    (AND (EQUAL (LENGTH DATUM) 3)
      (TYPE? Name (fetch (DatabaseEntry Name) of DATUM))
      (TYPE? PhoneNumber
        (fetch (DatabaseEntry PhoneNumber) of DATUM))
      (TYPE? OfficeNumber
        (fetch (DatabaseEntry OfficeNumber) of
          DATUM))))))
```

```
(RECORD Name (First Middle Last)
  (TYPE?
    (AND
      (LITATOM (fetch (Name First) of DATUM))
      (LITATOM (fetch (Name Middle) of DATUM))
      (LITATOM (fetch (Name Last) of DATUM))))))
```

```
(RECORD PhoneNumber (Prefix Extension) Prefix _ 494
  (TYPE?
    (AND
      (NUMBERP (fetch (PhoneNumber Prefix) of
        DATUM))
      (NUMBERP (fetch (PhoneNumber Extension) of
        DATUM))))))
```

```
(RECORD OfficeNumber (Bldg RoomNumber) Bldg _ 35
  (TYPE?
    (AND
      (NUMBERP (fetch (OfficeNumber Bldg) of
        DATUM))
      (NUMBERP (fetch (OfficeNumber RoomNumber)
        of DATUM))))))
```

With these declarations, you can do the following:

```
60_(TYPE? DatabaseEntry '(Foo (494 4456) Bar))
```

```
NIL
```

```
61_(TYPE? DatabaseEntry '((Frank G Halasz)(494 4320)(35 1654)))
```

```
1654
```

Final note: For `TYPERECORD` records you don't need to specify a `TYPE?` clause in the record declaration statement for the `TYPE?` statement to work. There is a default `TYPE?` clause that just checks for a name match between the *RecordName* given in the `TYPE?` statement and the `CAR` of the record instance.

DATATYPES: Defining New Objects in the Interlisp System

Records build data structures out of list structures. As such, records don't really create new types of objects in your Interlisp system; they just provide a convenient interface to lists that have a particular format.

In contrast, a *DATATYPE* creates a whole new type of object in the Interlisp system.

Basically, a `DATATYPE` behaves much like a `RECORD`. The `CREATE`, `fetch`, `replace`, and `TYPE?` statements all work exactly the same for `DATATYPE`s and for `RECORD`s.

But when you declare a `DATATYPE`, Interlisp goes through a lot of overhead to create a new type of object in its world.

Then, when you create an instance of the `DATATYPE`, you are not creating a list structure but a object of the new type with its own internal representation different from lists. Similarly, when you access the `DATATYPE`, you are accessing this new object type rather than a list.

The advantage of `DATATYPE`s over `RECORD`s is that access to any field in a `DATATYPE` is much, much faster.

The disadvantage of `DATATYPE`s is that there is a big overhead for the system in defining and managing new `DATATYPE`s.

Thus, if you are going to create lots and lots of instances of a given data structure and access these instances frequently, then you might want to make that structure a DATATYPE. If you are creating a few instance that are accessed infrequently, then you might want to make the structure a RECORD.

Since RECORDs and DATATYPEs are created and accessed in the same way, you can easily write and test a program using RECORDs and then switch to DATATYPEs when the program goes into real use.

Example:

To make the Name RECORD into a DATATYPE, all you would have to do is declare a DATATYPE instead of a RECORD as follows:

(DATATYPE Name (First Middle Last))

Note that TYPE? is automatically defined when the DATATYPE is declared, you do not need a TYPE? clause in the DATATYPE declaration statement.

Thereafter, Name will behave exactly as in all the examples above, except it will be a Name object rather than a list structure.

In particular, when you CREATE a Name, you will get back a Name object rather than a list.

87_ (CREATE Name First _ 'Sam Last _ 'Smith Middle _ 'A)

{Name}#51,142770

88_ (fetch (Name First) of IT)

Sam

89_ (TYPE? Name NewName)

T

90_ (TYPE? Name 'Foo)

NIL

System DATATYPEs

Many, many data structures in the Interlisp system are implemented in terms of DATATYPEs.

For example, Windows are just a DATATYPE data structure. The function CREATEW calls (CREATE WINDOW ...) and eventually returns the WINDOW object returned by this CREATE statement.

In fact, the following is the DATATYPE declaration for WINDOW from the Interlisp system code:

```
(DATATYPE WINDOW
  (DSP NEXTW SAVE REG BUTTONEVENTFN RIGHTBUTTONFN
  CURSORINFN CURSOROUTFN CURSORMOVEDFN REPAINTFN
  RESHAPEFN EXTENT USERDATA VERTSCROLLREG
  HORIZSCROLLREG SCROLLFN VERTSCROLLWINDOW
  HORIZSCROLLWINDOW CLOSEFN MOVEFN WTITLE
  NEWREGIONFN WBORDER PROCESS WINDOWENTRYFN))
```

A typical window might be filled in as follows:

```
{WINDOW} # 74,53544 Inspector
WINDOWENTRYFN    GIVE.TTY.PROCESS
PROCESS          {PROCESS}#71,45400
WBORDER         4
NEWREGIONFN     NIL
WTITLE          "EXEC Window"
MOVEFN          NIL
CLOSEFN         NIL
HORIZSCROLLWINDOW NIL
VERTSCROLLWINDOW NIL
SCROLLFN        NIL
HORIZSCROLLREG  NIL
VERTSCROLLREG   NIL
USERDATA        (\LINEBUF.OFD {STREAM}#71,105064
EXTENT          NIL
RESHAPEFN       NIL
REPAINTFN       NIL
CURSORMOVEDFN   NIL
CURSOROUTFN     NIL
CURSORINFN      NIL
RIGHTBUTTONFN   NIL
BUTTONEVENTFN   TOTOPW
REG             (74 283 571 336)
SAVE            {BITMAP}#62,21124
NEXTW           {WINDOW}#52,141144
DSP             {STREAM}#71,105150
```

Editing record declarations and record instance: EditRec and The Inspector

To be completed in class

References

Chapter 3 of the IRM.

Exercises

Redo the problem from LispCourse #23 using the Record Package.

LispCourse #26: Reprise; Continuation of Record Package from #25

Reprise: Sample Programming Exercise

The Problem

Consider an existing database containing the membership roster for the ACM.

The database is a list of entries.

Each entry corresponds to a person and contains the person's name, a list of dates relevant to that person's membership, a list of special interest groups(SIGs) that person belongs to, and an atom indicating whether the person's membership status.

The person's name is a list of three atoms corresponding to the persons first, middle, and last names in that order.

The date list contains two dates, the date of the person joined ACM and the date of the last renewal.

Each date is a list of three numbers: day, month, and year.

The list of SIGs consists of any number (including 0) of atoms like SIGOA and SIGART.

The membership status is one of the following atoms: Member, Associate, or Student.

Example Database:

```
((JoAnn A Jones)((23 10 75)(01 10 84))(SIGCHI SIGART SIGED)
```

```
Member)
```

```
((Bill B Smith)((12 03 82)(01 04 85))(SIGOA) Associate)
```

```
((Jill Jane Jerzy)((15 09 80)(14 09 84)) NIL Student))
```

Write functions to:

1. Add a new person entry to this database.
2. Return a list of all SIGCHI members in the ACM.
3. Given a last name, update that person's membership status to Member.
4. Given a last name and a date, update that person's last renewal date.

You can assume that the database is the value of the atom ACM.Members.

Step 1: Describe the database

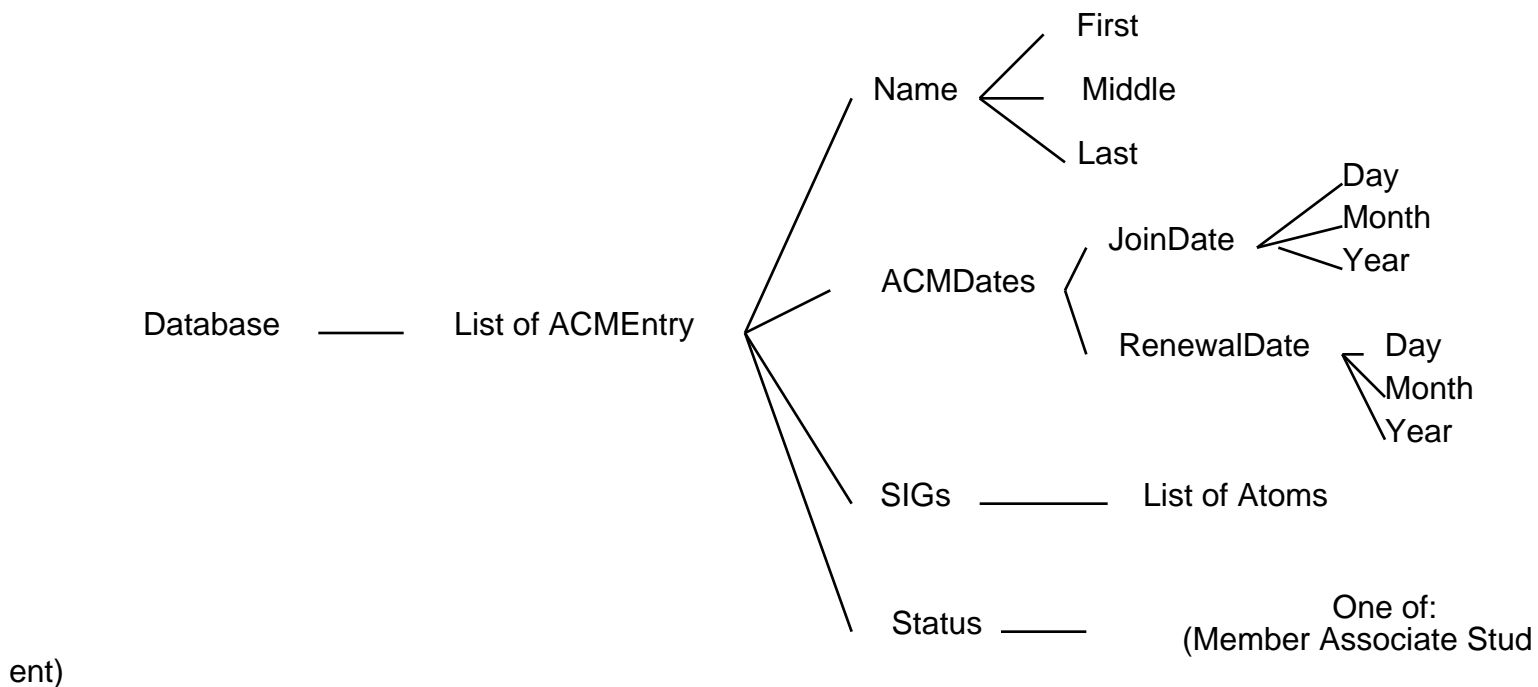
(RECORD ACMEntry (Name Dates SIGs Status))

(RECORD Name (First Middle Last))

(RECORD ACMDates (JoinDate RenewalDate))

(RECORD Date (Day Month Year))

Note: The whole database and the SIGs list are variable length and therefore cannot be described as RECORDs. In fact, they are likely to be the kind of data structures that you search looking for something rather than the kind of data structures you want to get a particular part (i.e., field) of.



Step 2: Write the constructors, selectors, and mutators for the data structures

A. Start with the lowest level structures (i.e., the one that consist of atoms)

i. Names: Deal with the Name RECORD

```

(DEFINEQ
  (ACM.CreateName
    (LAMBDA (First Middle Last)
      (CREATE Name First _ First Middle _ Middle last _
              Last)))
  (ACM.FirstName
    (LAMBDA (OldName NewFirst)
      (COND
        (NewFirst (replace (Name First) of OldName with
                           NewFirst))
        (T (fetch (Name First) of OldName))))))
  (ACM.MiddleName
    (LAMBDA (OldName NewMiddle)
      (COND

```

```

                                (NewMiddle (replace (Name Middle) of OldName
                                with NewMiddle))
                                (T (fetch (Name Middle) of OldName))))))
(ACM.LastName
  (LAMBDA (OldName NewLast)
    (COND
      (NewLast (replace (Name Last) of OldName with
        NewLast))
      (T (fetch (Name Last) of OldName))))))

```

ii. Dates: Deal with the Date RECORD

```

(DEFINEQ
  (ACM.CreateDate
    (LAMBDA (Day Month Year)
      (CREATE Date Day _ Day Month _ Month Year _ Year)))
  (ACM.Day
    (LAMBDA (OldDate NewDay)
      (COND
        (NewDay (replace (Date Day) of OldDate with
          NewDay))
        (T (fetch (Date Day) of OldDate))))))
  (ACM.Month
    (LAMBDA (OldDate NewMonth)
      (COND
        (NewMonth (replace (Date Month) of OldDate with
          NewMonth))
        (T (fetch (Date Month) of OldDate))))))
  (ACM.Year
    (LAMBDA (OldDate NewYear)
      (COND
        (NewYear (replace (Date Year) of OldDate with
          NewYear))
        (T (fetch (Date Year) of OldDate))))))

```

B. Continue with next level structures: the ones that use Names and Dates

i. ACMDates

```

(DEFINEQ
  (ACM.CreateACMDateList
    (LAMBDA (JoinDate RenewalDate)
      (CREATE ACMDates JoinDate _ JoinDate RenewalDate _
        RenewalDate)))
  (ACM.JoinDate
    (LAMBDA (DatesList NewJoinDate)
      (COND
        (NewJoinDate (replace (ACMDates JoinDate) of
          DatesList with NewJoinDate))
        (T (fetch (ACMDates JoinDate) of DatesList))))))
  (ACM.RenewalDate
    (LAMBDA (DatesList NewRenewalDate)
      (COND
        (NewRenewalDate (replace (ACMDates
          RenewalDate) of DatesList with
          NewRenewalDate))
        (T (fetch (ACMDates RenewalDate) of
          DatesList))))))

```

C. Highest level record structure: the ACMEntry

```

(DEFINEQ
  (ACM.CreateACMEntry
    (LAMBDA (Name Dates SIGs Status)
      (CREATE ACMEntry Name _ Name Dates _ Dates SIGs _ SIGs
        Status _ Status)))
  (ACM.Name
    (LAMBDA (Entry NewName)
      (COND
        (NewName (replace (ACMEntry Name) of Entry
          with NewName))
        (T (fetch (ACMEntry Name) of Entry))))))
  (ACM.DatesList

```

```

(LAMBDA (Entry NewDatesList)
  (COND
    (NewDatesList (replace (ACMEntry Dates) of
      Entry with NewDatesList))
    (T (fetch (ACMEntry Dates) of Entry))))))
(ACM.SIGs
  (LAMBDA (Entry NewSIGs)
    (COND
      (NewSIGs (replace (ACMEntry SIGs) of Entry with
        NewSIGs))
      (T (fetch (ACMEntry SIGs) of Entry))))))
(ACM.Status
  (LAMBDA (Entry NewStatus)
    (COND
      (NewStatus (replace (ACMEntry Status) of Entry
        with NewStatus))
      (T (fetch (ACMEntry Status) of Entry))))))

```

Step 3: Write the functions that do the work as required.

1. Add a new person entry to this database.

Plan: Create an entry record for the new person and CONS it onto the existing database list.

```

(DEFINEQ
  (ACM.AddPerson
    (LAMBDA (First Middle Last JoinDate RenewalDate SIGs Status)
      (SETQ ACM.Members
        (CONS
          (ACM.CreateACMEntry
            (ACM.CreateName First Middle
              Last)
            (ACM.CreateACMDatesList
              JoinDate
                RenewalDate)

```

```

          SIGs Status)
    ACM.Members))))))

```

2. Return a list of all SIGCHI members in the ACM.

Plan: Iterate through list of entries. For each entry that is a member of SIGCHI, add it to a list of the SIGCHI members (which is the value of the atom SIGCHI.Members).

```

(DEFINEQ
  (ACM.SIGCHIMembers
    (LAMBDA NIL
      (SETQ SIGCHI.Members NIL)
      (FOR Entry in ACM.Members DO
        (COND
          ((MEMBER 'SIGCHI (ACM.SIGS Entry))
           (SETQ SIGCHI.Members
                 (CONS Entry
                       SIGCHI.Members))))
          (T
           (SETQ SIGCHI.Members))))
      SIGCHI.Members)))

```

3. Given a last name, update that person's membership status to Member.

Plan: Iterate through list of entries. For the entry that matches on last name, change the Status field.

```
(DEFINEQ
  (ACM.ChangeStatusToMember
    (LAMBDA (LastName)
      (FOR Entry in ACM.Members DO
        (COND
          ((EQUAL LastName
            (ACM.LatName (ACM.Name
              Entry)))
            (ACM.Status Entry 'Member))))))
```

4. Given a last name and a date, update that person's last renewal date.

Plan: Iterate through list of entries. For the entry that matches on last name, change the renewal date.

```
(DEFINEQ
  (ACM.ChangeRenewalDate
    (LAMBDA (LastName NewRenewalDate)
      (FOR Entry in ACM.Members DO
        (COND
          ((EQUAL LastName
            (ACM.LatName (ACM.Name
              Entry)))
            (ACM.RenewalDate
              (ACM.DatesList Entry)
              NewRenewalDate))))))
```


Step 4: Save your work on a file and list it.

Decide to call the file ACMLIST.

Set up the COMS list for MAKEFILE

```
(SETQ ACMLISTCOMS (QUOTE (
  (RECORDS ACMEntry Name ACMDates Date)
  (VARS ACM.Members)
  (* * Constructors, Selectors, and Mutators)
  (FNS ACM.CreateName ACM.FirstName ACM.MiddleName
    ACM.LastName)
  (FNS ACM.CreateDate ACM.Day ACM.Month ACM.Year)
  (FNS ACM.CreateACMDateList ACM.JoinDate
    ACM.RenewalDate)
  (FNS ACM.CreateACMEntry ACM.Name ACM.DatesList
    ACM.SIGs ACM.Status)
  (* * Work-doing functions)
  (FNS ACM.AddPerson ACM.SIGCHIMembers
    ACM.ChangeStatusToMember ACM.ChangeRenewalDate)
)))
```

Do the MAKEFILE

```
(MAKEFILE '{PHYLUM}<HALASZ>LISPCOURSE>ACMLIST)
```

Do the LISTFILES

```
(LISTFILES {PHYLUM}<HALASZ>LISPCOURSE>ACMLIST)
```

Step 5: Test, Debug, and redo the MAKEFILE.

EFS.

Continuation from #25: Editing Record/Datatype Declarations and Instances

Examining and Editing Record/Datatype Declarations: RECLOOK and EDITREC

To see the declaration for record or datatype named *RecordName*, use **(RECLOOK *RecordName*)**.

```
61_ (RECORD ACMDate (Month Day Year) Month _ 01 Day _ 01 Year
_ 00)
```

ACMDate

```
62_ (RECLOOK 'ACMDate)
```

```
(RECORD ACMDate (Month Day Year) Month _ 01 Day _ 01 Year _ 00)
```

```
63_ (DATATYPE ACMDate (Month Day Year))
```

(record ACMDate redeclared)

ACMDate

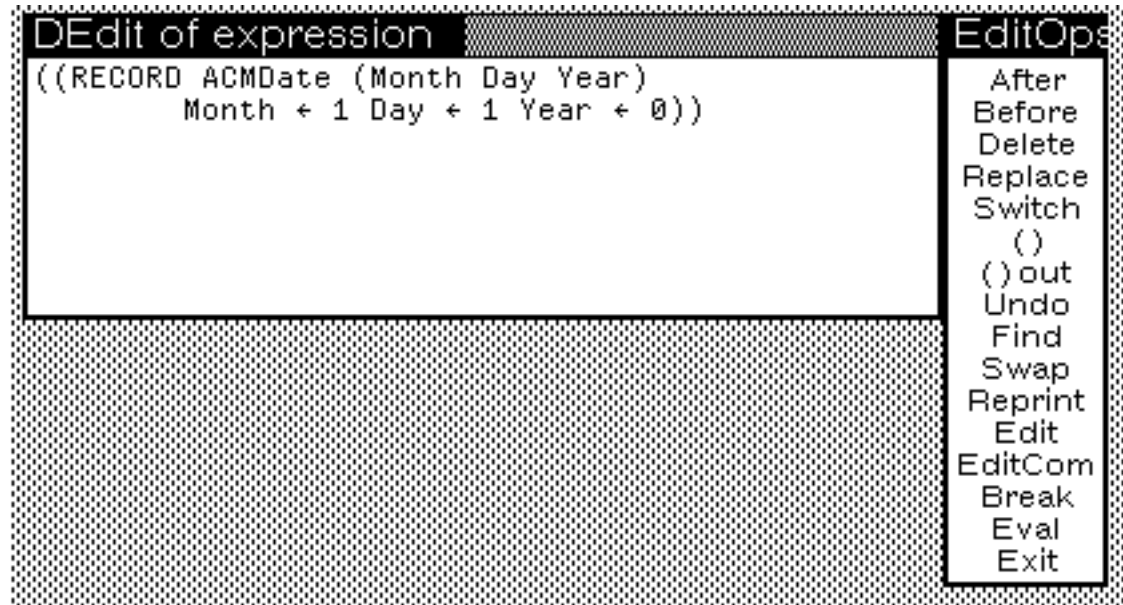
```
64_ (RECLOOK 'ACMDate)
```

```
(DATATYPE ACMDate (Month Day Year))
```

To edit (i.e., change) the declaration for record or datatype named *RecordName*, use **(EDITREC *RecordName*)**. [EDITREC is an NLAMBDA function]

This will pop you into a DEdit on the record/datatype declaration. If you exit DEdit with **OK**, the record/datatype will be redeclared. If you exit with **Stop**, no change in the declaration will happen.

```
(EDITREC ACMDate)
```



Examining and Editing Record/Datatype Instances: The Inspector

To examine or change the value of a field in a record/datatype instance, use the Inspector. The Inspector is called as follows:

(INSPECT *Instance*)

If *Instance* is a *record*, then a menu will be displayed with the following choices: *DisplayEdit*, *TtyEdit*, *Inspect*, *AsRecord*.

Choose: *AsRecord*

This will bring up a menu of all the RECORD types in the system. Choose the appropriate record type from this menu (e.g., choose ACMDate if the record is an ASCMDate).

Finally, this brings up an Inspect window (after prompting for a location) displaying the fields and values of the fields for the given *Instance* of the chosen record type.

If *Instance* is a *datatype*, then an Inspect window will be displayed immediately (after prompting for a location) showing the fields and values of the fields for the given *Instance*.

An Inspect window looks as follows:

```
75_ (RECORD ACMDate (Month Day Year) Month _ 01 Day _ 01 Year _ 00)
```

```
ACMDate
```

```
76_ (SETQ Test (CREATE ACMDate Month _ March Year _ 85 Day _ 25))
```

```
(March 25 1985)
```

```
77_ (INSPECT Test)
```

```
{WINDOW}#33,123456
```



The Inspect window has two columns:

The Left-hand column lists the *field names* of the ACMDate record type.

The Right-hand column lists the corresponding *values* of these fields in the instance being inspected.

You can select any item in the Inspect window by pointing to it and clicking the **LEFT** mouse button. The selected item will be inverted.



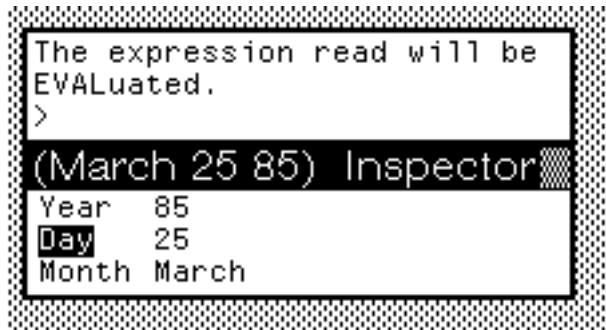
You can *act* on the selected item by clicking the **MIDDLE** mouse button anywhere inside the Inspect window. The action taken depends on whether the selected item is a field (left column) or a value (right column).

If the selected item is a *field*: You can change the *value* for that field as follows:

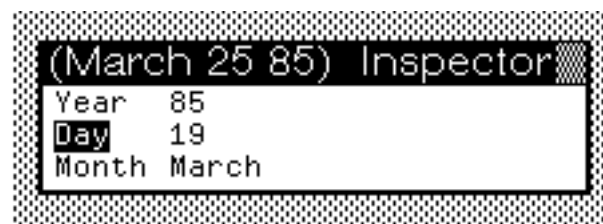
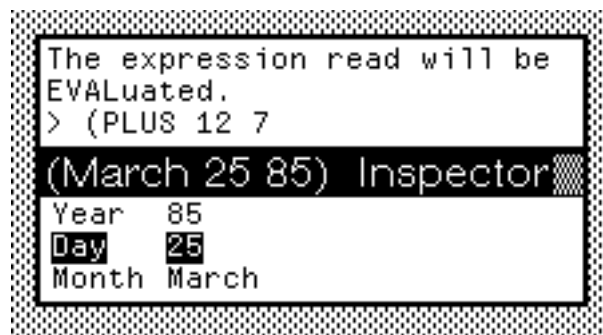
Clicking the **MIDDLE** button will bring up a menu of one item:
Set

Choose this item. (selecting off the menu aborts the interchange)

This will bring up a small prompt box above the Inspect window.

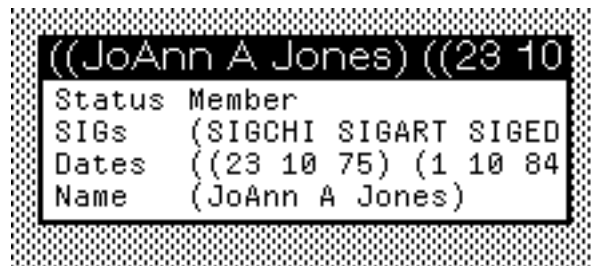


Type an S-expression that will evaluate to the new value you wish the field to have. This expression will be evaluated and the value of the selected field will be set to the result.

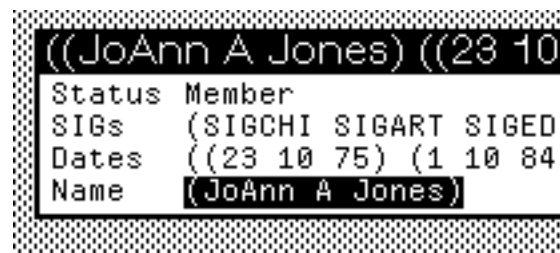


If the selected item is a *value*: (INSPECT value) will be evaluated. This is useful for Inspecting embedded record structures.

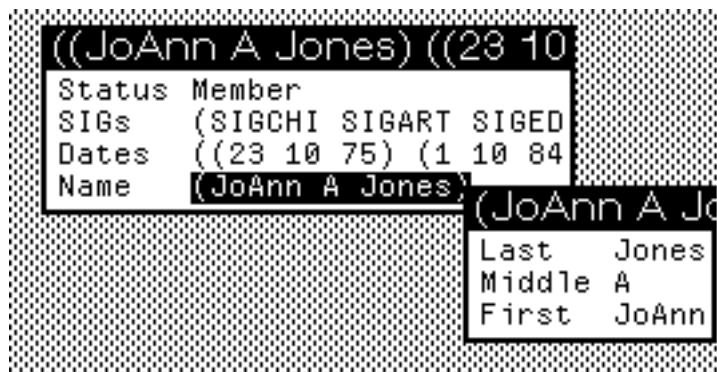
Example: Inspecting an ACMEEntry instance



Selecting the **value** of the Name field using the LEFT button



Clicking the MIDDLE button to INSPECT the name value.



Note: the interaction sequence to bring up the second inspector is the same as for the first inspector, i.e., I had to specify *AsRecord* and choose the *Name* record type from the menu of all record types.

To get rid of an Inspect window ǂ just close it using the RIGHT button menu.

Using the Inspector

The inspector is very useful to examine your data structures in an interpretable way while you are debugging.

For example:

Consider the RECORD definitions given in the ACM problem above.

Consider the sample database:

```
(SETQ ACM.Members
  '((Jones JoAnn A ) Member ((23 10 75)(01 10 84))(SIGCHI
    SIGART))
  ((Smith Bill B ) Associate ((12 03 82)(01 04 85))(SIGOA))
  ((Jerzy Jill Jane ) Student ((15 09 80)(14 09 84)) NIL)))
```

None of our functions we defined above would work on this sample database. Why? A quick inspect shows why.

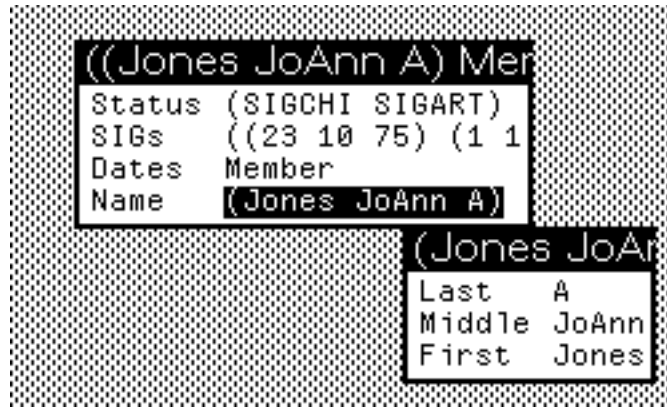
1. Inspect the first entry: (INSPECT (CAR ACM.Members))

```
((Jones JoAnn A) Mer
Status (SIGCHI SIGART)
SIGs ((23 10 75) (1 1
Dates Member
Name (Jones JoAnn A)
```

Notice that the fields and values don't quite match up; e.g., Status field is a list.

2. Notice that Name doesn't look quite right. So Inspect it.

```
((Jones JoAnn A) Mer
Status (SIGCHI SIGART)
SIGs ((23 10 75) (1 1
Dates Member
Name (Jones JoAnn A)
```



Notice that the fields and values don't quite match up in the name either; e.g., Last name field is a middle initial.

Bottom line: Either the sample database is in error or our RECORD declarations are in error. In this case, its the former.

References

The Inspector is described in Section 20.4 of the IRM.

EDITREC and RECLOOK are in Section 3.7 of the IRM.

Exercise

For the following exercise, it might be handy to know about the following embellishment of the FOR loop:

(FOR *Variable* IN *List* WHEN *Predicate* DO *S-Exprs*)

Instead of executing the *S-Exprs* following the DO on every iteration, this FOR loop evaluates the *Predicate* following the WHEN on every iteration. If *Predicate* evaluates to a non-NIL value, then the *S-exprs* following DO are evaluated. If *Predicate* evaluates to NIL, then the FOR loop skips the *S-exprs* after the DO on this iteration.

Note that the WHEN clause works with COLLECT as well as DO.

Examples:

```
76_(FOR A IN '((T 1)(NIL 2)(T 3)(NIL 4))
          WHEN (CAR A) COLLECT (CADR A))
(1 3)
```



```
77_(FOR A IN '(T 1)(NIL 2)(T 3)(NIL 4))  
      WHEN (NULL (CAR A)) COLLECT (CDR A))  
((2) (4))
```

Exercise from Touretzky, p. 179. (See following few pages)

LispCourse #27: Homework from #26; Completion of #25 & #26

Solution to Homework from #26 is attached.

LispCourse #28: Type Checking; Strings; Arrays

Type Checking in Interlisp

The Concept of Type Checking

Many programming languages insist that each parameter to a function be typed i.e., that it be declared to be a variable whose value is a given type of data (e.g., an atom, a number, or a list).

In these languages when you call a function, the language checks the type of each argument in the argument list to make sure that it matches the declared type of each parameter in the parameter list.

An error results if you call a function with arguments of the wrong type.

Example from PASCAL:

```
function SumOfSquares (X: INTEGER, Y: INTEGER): INTEGER  
BEGIN  
    SumOfSquares := (X * X) + (Y * Y)  
END
```

This PASCAL function definition says that *SumOfSquares* is a function that takes two integer arguments and returns an integer.

Therefore:

(SumOfSquares 2 3) is okay

But *(SumOfSquares "Foo" 4)* causes an immediate error because "Foo" is not an integer, but a string of characters.

The error will say something like "First argument to SumOfSquares is not an integer as required."

Note that *(SumOfSquares 1.234 4)* also causes an error because 1.234 is not an integer, but a floating point number.

Interlisp does not do this type checking. You do not have to declare the expected type of a parameter in your function definitions AND Interlisp does not check the type of the arguments when you call the function.

Example:

```
(DEFINEQ
  (SumOfSquares
    (LAMBDA (X Y)
      (PLUS (TIMES X X)(TIMES Y Y))))))
```

(SumOfSquares 2 3) works fine and returns an integer, 13.

(SumOfSquares 1.234 4) also works fine, but returns a floating point number 17.52276.

But, *(SumOfSquares 'Foo 4)* will cause an error (i.e., a BREAK).

The error will occur somewhere deep inside the SumOfSquares function and the error message will not reflect the real problem – i.e., that the argument to SumOfSquares was of the wrong type.

```
EXEC Window
17.52276
67+(SumOfSquares 'Foo 4)
NON-NUMERIC ARG
Foo
Foo - NON-NUMERIC ARG break: 1
(FTIMES broken)
68:
```

Note that the error occurs in FTIMES, not in SumOfSquares where it really should have happened.

In this case, its easy to trace back to see that in fact SumOfSquares has a bad argument. But if these were complex functions embedded many levels deep, it could be very difficult to figure out where the 'Foo that tripped up the FTIMES actually came from.

This example points out both the strengths and weaknesses of the Interlisp scheme of no type checking.

The strength is that you can easily write a single function that handles many different kinds of data ÿ e.g., both integers and floating numbers.

The weakness is that data of the wrong type may be passed through many layers of a program before an error is tripped up. When this happens, it is very difficult to trace where the bad data came from.

Adding type checking to your Interlisp functions

The solution to Interlisp's lack of explicit type checking is to judiciously add type checking to the functions you write.

The basic goal of type checking is to catch data of the wrong type at the earliest point at which you can detect that it is wrong.

In particular, at any point where new data enters the system – e.g., at user type-in or when reading from a database – make sure the data is of the type expected.

For example, if you are getting information from a user to add to the database, check the arguments to the add-to-database function to make sure that the user is entering the correct type of data.

Also, at any other point where there is a possibility of a wrong type of data being passed to a function, check the type of the arguments within that function.

It is important to type check only where necessary because type checking does take some time.

Too much type checking will make your program very inefficient.

On the other hand, with too little type checking your program may not run at all.

Type Checking Technique

To check the arguments being passed to a function, just put a COND clause at the beginning of the function. The COND should have a clause for each parameter that aborts the function and reports an error to the user if the value of the parameter is not an allowable type (i.e., if the calling function had an incorrectly typed argument).

Example:

```
(DEFINEQ
  (SumOfSquares (LAMBDA (X Y)
    (COND
      ((NOT (NUMBERP X))
        (ERROR "Incorrect first argument to
SumOfSquares -- must be a number. But it
is --" X))
      ((NOT (NUMBERP Y))
        (ERROR "Incorrect second argument to
SumOfSquares -- must be a number. But it
is --") Y)
      (T
        (PLUS (TIMES X X)(TIMES Y Y)))))))
```

The screenshot shows a terminal window titled "EXEC Window". The first part of the output shows a prompt "74+(SumOfSquares 'Foo 4)" followed by an error message: "Incorrect first argument to SumOfSquares -- must be a number. But it is --". Below this, the word "Foo" is printed. A second, larger error message box is overlaid on the bottom right, containing the text "(ERROR broken)" and "75:". The background of the terminal window has a halftone dot pattern.

Second Example:

Define a function to return the tail of a list starting from its second to last element.

```
(DEFINEQ
```

```
  (SecondFromLast (LAMBDA (List)
```

```
    (** Return the tail of a list starting from its second
       to last element.  Make sure List is in fact a list and
       is of length 2 or more.)
```

```
    (COND
```

```
      ((NOT (LISTP List))
```

```
        (ERROR "SecondFromLast:
                Argument must be a list.  It is --"
                List))
```

```
      ((LESSP (LENGTH List) 2)
```

```
        (ERROR "SecondFromLast: List
                argument must be at least 2 items
                long.  Its current length is --"
                (LENGTH List)))
```

```
      (T
```

```
        (NTH List (DIFFERENCE
                    (LENGTH List) 2))))))
```

```
EXEC Window
78+
78+(SecondFromLast (LIST 55))
SecondFromLast: List argument must be at least 2 items long.
Its current length is --
79:
SecondFromLast: List argument must be at least 2 it
(ERROR broken)
79:
ERROR
BREA
ERRC
COND
Seco
EVAL
LISP
ERRC
EVAL
ERRC
T
```


Type checking predicates

The predicates necessary for checking the type of a given piece of data have already been discussed.

LispLecture #4 (page 3), discusses the predicates **LITATOM**, **NUMBERP**, and **LISTP** that check if their argument is of type litatom, number, and list, respectively.

LispLecture #25 (page 13) discusses the **TYPE?** statement that allows one to build predicates that check if their argument is a particular type of **RECORD** or a particular **DATATYPE**.

The next section contains an overview of the types of data in Interlisp, including the predicates that check for each of the types.

Overview of the Types of Data in Interlisp

Below is a list of the types of data supported by Interlisp. In parentheses following each data type is the predicate that checks for that data type.

Primitive data types (those provided as part of the core Interlisp system).

Atoms (*ATOM*)

Litatoms (*LITATOM*)

Numbers (*NUMBERP*)

Integers (*FIXP*)

Floating Point Numbers (*FLOATP*)

Lists (*LISTP*)

[Prop Lists]

[Assoc Lists]

Strings (*STRINGPs*)

Arrays (*ARRAYPs*)

Compound data types (those built by combining primitive data types)

RECORDs

RECORDs defined by the Interlisp implementors

SKETCHs (*TYPE? SKETCH*)

GRAPHS (*no type checking predicate!*)

...

User defined RECORDs (*TYPE? RecordName*)

DATATYPEs

DATATYPEs defined by the Interlisp implementors

BITMAPs (*BITMAPP*)

HARRAYs (*HARRAYP*)

WINDOWs (*WINDOWP*)

PROCESSes (*PROCESSP*)

STREAMs (*STREAMP*)

...

User defined DATATYPES (*TYPE? DataTypeName*)

Print Names

Every data object in Interlisp has something called a *print name* (often called a *pname*).

A data object's print name is the thing that gets printed when Lisp needs to communicate with the outside world – e.g., in the PRINT phase of the READ-EVAL-PRINT loop.

Examples:

Atoms: the print name is the name of the atom. E.g., *FOOBAR*.

Lists: the print name starts with a "(", followed by the print names of all of the items in the list separated by spaces, followed by a)". E.g., *(1 (2 3) A)*.

DATATYPES: the print name is the DatatypeName in "{ }" followed by some numbers. E.g., *{WINDOW}#65,12345* and *{PROCESS}#12,12399*

Print names are NOT unique.

For example:

{WINDOW}#65,1234 is the print name for some window. But it is also the print name for an atom whose name (i.e., print name) is exactly *{WINDOW}65,1234*.

For some data types, the Lisp object can be referred to by typing its print name into the Lisp Exec.

For example, to refer to an atom, you just type its print name (i.e., its name) into the Lisp Exec.

For other data types, the print name is just for printing; you can't type it back into the Lisp exec to refer to the object.

For example, if you type in `{WINDOW}#65,12345` into the Lisp Exec, the Lisp Exec will think you mean the atom by that name, not the window that has that pname.

Arrays

An array is a primitive data type that represents a fixed number (N) of other Interlisp objects stored in a one-dimensional vector.

An Array with 7 Elements

1:	(1 2 3 (3 4))
2:	444
3:	FooBar
4:	{WINDOW}#1,2234
5:	(List of Elements)
6:	1.2345
7:	{PROCESS}#1,2234

Think of an array as a set of mailboxes arranged in one column and N rows. In each mailbox is some arbitrary Interlisp object (an atom, a list, a window, etc.) You can get at any object stored in the array of mailboxes only by specifying the row number of the mailbox it is stored in.

Alternatively, an array is like a RECORD with N fields, but the fields can be accessed by *number only* and *not by name*.

An array is also like a list, but it has a fixed length: you can't add or remove elements from an array.

Moreover, you can't deal with parts of the array as a single entity as you can a list (e.g., there is no operation like CDR for arrays).

For certain applications, arrays are much more efficient than lists.

In general, however, any program that uses arrays can be rewritten using lists with possible loss of efficiency and elegance.

Array manipulation functions

Interlisp has a number of functions that allow you to manipulate arrays, i.e., to create arrays, to access the objects stored in an array, etc.

Creating arrays

(ARRAY *Size*) ž Creates an array of size *Size* (i.e., with *Size* entries). Returns (a pointer to) the array. The array is initialized to have every element contain NIL.

Example:

```
1_(SETQ MyArray (ARRAY 10))
  {ARRAY}#65,51054
2_MyArray
  {ARRAY}#65,51054
```

Array predicate

(ARRAYP *Arg*) ž Returns *Arg* if *Arg* is an array, NIL otherwise.

Examples:

```
3_(ARRAYP MyArray)
  {ARRAY}#65,51054
4_(ARRAYP 10)
  NIL
5_(ARRAYP (LIST 1 2 3 4 5))
  NIL
6_(ARRAYP 'ARRAY)
```

NIL

Accessing the entries of an array

(SETA *Array N Value*) ž Sets the *N*th element of *Array* to have the value *Value* (i.e., puts the Lisp object specified by *Value* into the *N*th element of *Array*).

Examples:

```
7_(SETA MyArray 1 (LIST 1 2 3))
```

```
(1 2 3)
```

```
8_(SETA MyArray 2 (PLUS 2 3))
```

```
5
```

```
9_ (SETA MyArray 12 15)
```

```
ILLEGAL ARG
```

```
12
```

```
10_ (SETA MyArray 3 15)
```

```
15
```

(ELT *Array N*) ž Returns the Lisp object stored in the *N*th element of *Array*.

Examples:

```
11_(ELT MyArray 1)
```

```
(1 2 3)
```

```
12_(ELT MyArray 2)
```

```
5
```

```
13_ (ELT MyArray 12)
```

```
ILLEGAL ARG
```

```
12
```

```
14_ (ELT MyArray 5)
```

```
NIL
```

Finding out the size of an array

(ARRAYSIZE *Array*) ž Returns the size of array *Array*.

Example:

```
15_(ARRAYSIZE MyArray)
10
16_(ARRAYSIZE (ARRAY 50))
50
```

Using arrays

Problem:

Imagine you work for a company that has 20 products (numbered 1 thru 20). Each product has a "list price" and an "our price".

1. Write a function that takes a product number and returns the "list price".
2. Write a function that takes a product number and returns the "our price".
3. Write a function that replaces the price entry field with the atom OutOfStock for a given product number.

Solution:

Store the data in an array of size 20, where each entry is a RECORD called *Prices* with 2 fields named *ListPrice* and *OurPrice*.

```
(SETQ PriceArray (ARRAY 20))
```

```
(SETA PriceArray 1 (CREATE Prices ListPrice _ 1.00 OurPrice
1.25))
```

... *{Fill in rest of proce array with values}*

```
(DEFINEQ
```

```
(LC.ListPrice (LAMBDA (ProductNumber)
```

```
(fetch (Prices ListPrice) of
      (ELT PriceArray ProductNumber)))
(LC.OurPrice (LAMBDA (ProductNumber)
             (fetch (Prices OurPrice) of
                   (ELT PriceArray ProductNumber))))
(LC.MarkOutOfStock (LAMBDA (ProductNumber)
                   (SETA PriceArray ProductNumber 'OutOfStock))))
```

The advantage of using an array in this case is that you need to get to and CHANGE any element of the data structure at any time.

This is easy with arrays using ELT and SETA.

It is harder with lists.

First, (CAR (NTH List N)) takes longer than (ELT Array N).

Second, there is no easy way to do SETA with list structures.
(Though we will learn how to do so not easily later!!!).

Strings

Strings are a primitive data type in Interlisp used for representing sequences of characters. (As opposed to atoms which are used as symbols for arbitrary objects.)

A string is an arbitrary sequence of characters, including spaces and tabs.

The print name of a string encloses the characters in the string in double quotes.

Examples:

"B"

"abc"

"Frank G. Halasz"

"This is a very long string. It consists of several sentences. The sentences are separated by spaces."

Strings can be from 0 to any number of characters in length.

The string "" is the empty string having 0 characters.

Strings can contain any characters except the double quote character and %.

To include these characters they must be preceded by the % escape as in atom names.

Example:

"String with single %% percent sign"

String manipulation functions

Interlisp has a number of functions that allow you to manipulate strings, i.e., to create strings, to concatenate strings, to decompose strings, to search through strings, etc.

String predicate

(STRINGP *Arg*) ž Returns *Arg* if *Arg* is a string, NIL otherwise.

Examples:

```

1_(STRINGP "ABC")
"ABC"
2_(STRINGP 'ABCDEF)
NIL
3_(STRINGP (LIST 1 2 3 4))
NIL
8_(STRINGP (TEDIT))
NIL

```

Creating strings

(MKSTRING *Arg*) ž If *Arg* is already a string, returns *Arg*. Otherwise, makes and returns a string containing the print name of *Arg*.

Examples:

```

5_(MKSTRING "ABC")
"ABC"
6_(MKSTRING 'ABCDEF)
"ABCDEF"
7_(MKSTRING (LIST 1 2 3 4))
"(1 2 3 4)"
8_(MKSTRING (TEDIT))
"{PROCESS}#61,130000"

```

(ALLOCSTRING *N Character*) ž Returns a string *N* characters long where each character is *Character*. *Character* can be a single character string/atom or a character code (see LispCourse #10, page 5)

Examples:

```

9_(ALLOCSTRING 5 "A")
"AAAAA"
10_(ALLOCSTRING 15 'B)
"BBBBBBBBBBBBBBBB"
11_(ALLOCSTRING (PLUS 3 4) 63)
"???????"

```

```
12_(ALLOCSTRING 7 (CHARCODE ?))
"???????"
```

Comparing strings

(STREQUAL *Str1 Str2*) ž Returns T if *Str1* and *Str2* are both strings and contain the same sequence of characters.

Examples:

```
13_(STREQUAL "ABCDEF" "ABCDEF")
```

T

```
14_(STREQUAL (ALLOCSTRING 5 'A) "AAAAA")
```

T

```
15_(STREQUAL 'A "A")
```

NIL

```
16_(STREQUAL (MKSTRING 'AAA)(ALLOCSTRING 3
"A"))
```

T

Concatenating strings

(CONCAT *Str1 Str2 ...*) ž Returns a new string that consists of the concatenation of the characters in *Str1*, *Str2*, *Str3* If any *StrI* is not a string, the MKSTRING of that *STR1* is used instead of *StrI*.

Examples:

```
17_(CONCAT "ABCDEF" "GHIJKL")
```

"ABCDEFGHIJKL"

```
18_(CONCAT (ALLOCSTRING 5 'A) "FOO BAR")
```

"AAAAAFOO BAR"

```
19_(CONCAT 1234 " " 5678 " " (LIST 9 0))
```

"1234 5678 (9 0)"

```
20_(CONCAT "This is the kind of value that TEdit returns
-- " (TEDIT))
```

*"This is the kind of value that TEdit returns --
{PROCESS}#61,130000"*

Decomposing strings

(SUBSTRING *Str Start End*) ž Returns a new string that consists of the characters of *Str* starting at character number *Start* and ending at character number *End*.

If *End* is NIL, the last character of *Str* is used.

If *Start* or *End* are negative, they are interpreted as being positions from the end of *Str*.

Examples:

21_(SUBSTRING "ABCDEF" 2 4)

"BCD"

22_(SUBSTRING "FOO BAR" 4)

"BAR"

23_(SUBSTRING "FOO BAR" -3)

"BAR"

24_(SUBSTRING "FOO BAR" 2 -2)

"OO BA"

Searching strings

(STRPOS *Pattern String Start SkipChar*) ž Searches through string *String* looking for any sequences of characters that matches the characters in string *Pattern*. If a match is found, STRPOS returns the number of the character in *String* where the match starts. If no match is found, STRPOS returns NIL.

If *Start* is specified, the search begins at character number *Start* in *String*.

If *SkipChar* is specified, any instance of *SkipChar* in the *Pattern* string will match any character in *String*. (*SkipChar* is the wildcard character).

If *Pattern* and/or *String* are not strings, their MKSTRINGs will be used instead.

Examples:

```
25_(STRPOS "Q" "ABCDEF")
```

```
NIL
```

```
26_(STRPOS "D" "ABCDEF")
```

```
4
```

```
27_(STRPOS "C*E" "ABCDEF" NIL "**")
```

```
3
```

```
28_(STRPOS "O" "FOO BAR" 4)
```

```
NIL
```

Using Strings

Given a list of strings of the format: "Name: *Last,First*".

Write a function to extract all names with "sz" in them. The function should return a list of strings with the format "*First Last*".

```
(DEFINEQ
```

```
  (LC.szP (LAMBDA (String)
```

```
    (* * Does String have an sz in it?)
```

```
    (OR (STRPOS "sz" String)
```

```
        (STRPOS "Sz" String)
```

```
        (STRPOS "sZ" String)
```

```
        (STRPOS "SZ" String))))
```

```
  (LC.GetLastName (LAMBDA (String)
```

```
    (* * Extract the last name from the string)
```

```
    (SUBSTRING String
```

```
      (PLUS 1 (STRPOS " " String))
```

```
      (DIFFERENCE (STRPOS ", " String) 1))))
```

```
  (LC.GetFirstName (LAMBDA (String)
```

```

(* * Extract the first name from the string)
  (SUBSTRING String (PLUS 1 (STRPOS "," String))))

(LC.FindSzNames (LAMBDA (List)
  (* * Extract all names with sz in the last name)
  (FOR Entry in List
    WHEN (LC.szP Entry)
    COLLECT
      (CONCAT
        (LC.GetFirstName Entry)
        " "
        (LC.GetLastName Entry))))))

6_(SETQ TestList (QUOTE
  ("Name: Halasz, Frank" "Name: Smith, Sam"
   "Name: Beals, Szmata" "Name: Schatz, Sheila")))
7_(LC.szP (CAR TestList))
11
8_(LC.GetFirstName (CAR TestList))
"Frank"
9_(LC.GetLastName (CAR TestList))
"Halasz"
10_(LC.FindSzNames TestList)
("Frank Halasz" " Szmata Beals")

```

References

In general, primitive data types are covered in Chapter 2 of the IRM.

Arrays are covered in Section 2.7.

Strings are covered in Section 2.6.

LispCourse #29: Solutions for Homework #28

Sample solution functions attached.

Sample Run:

```
20_(LC.CreateDatabase 'TestDB]
NIL
21_(LC.AddEntry 'TestDB 'Frank 'G. 'Halasz 331404999 15 04
84 11000)
(((Frank G. Halasz) 331404999 (15 4 84) 11000))
22_(LC.AddEntry (QUOTE TestDB)
  (QUOTE George)
  (QUOTE B.)
  (QUOTE Smith)
  100455467 5 6 83 17000)
(((George B. Smith) 100455467 (5 6 83) 17000) ((Frank G. Halasz)
331404999 (15 4 84) 11000))
23_(LC.AddEntry (QUOTE TestDB)
  (QUOTE Ann)
  (QUOTE Q.)
  (QUOTE Merrick)
  345223431 14 2 80 29000)
(((Ann Q. Merrick) 345223431 (14 2 80) 29000) ((George B. Smith)
100455467 (5 6 83) 17000) ((Frank G. Halasz) 331404999 (15 4 84)
11000))
24_(LC.AddEntry (QUOTE TestDB)
  (QUOTE Zoltan)
  (QUOTE Z.)
  (QUOTE Zolka)
  995761234 01 1 27 98000)
(((Zoltan Z. Zolka) 995761234 (1 1 27) 98000) ((Ann Q. Merrick)
345223431 (14 2 80) 29000) ((George B. Smith) 100455467 (5 6 83)
17000) ((Frank G. Halasz) 331404999 (15 4 84) 11000))
25_(LC.AddEntry (QUOTE TestDB)
  (QUOTE Arnie)
  (QUOTE A.)
  (QUOTE Aardvark)
  100101000 12 12 85 10000)
```

```
((Arnie A. Aardvark) 100101000 (12 12 85) 10000) ((Zoltan Z.
Zolka) 995761234 (1 1 27) 98000) ((Ann Q. Merrick) 345223431 (14 2
80) 29000) ((George B. Smith) 100455467 (5 6 83) 17000) ((Frank
G. Halasz) 331404999 (15 4 84) 11000))
```

```
26_(LC.AddEntry (QUOTE TestDB)
```

```
(QUOTE Melinda)
```

```
(QUOTE F.)
```

```
(QUOTE Twiddle)
```

```
446891876 25 8 76 34000)
```

```
((Melinda F. Twiddle) 446891876 (25 8 76) 34000) ((Arnie A.
Aardvark) 100101000 (12 12 85) 10000) ((Zoltan Z. Zolka) 995761234
(1 1 27) 98000) ((Ann Q. Merrick) 345223431 (14 2 80) 29000) ((
George B. Smith) 100455467 (5 6 83) 17000) ((Frank G. Halasz)
331404999 (15 4 84) 11000))
```

```
27_(LC.SortDatabaseByField 'TestDB 'SS#]
```

```
((Arnie A. Aardvark) 100101000 (12 12 85) 10000) ((George B.
Smith) 100455467 (5 6 83) 17000) ((Frank G. Halasz) 331404999 (15
4 84) 11000) ((Ann Q. Merrick) 345223431 (14 2 80) 29000) ((
Melinda F. Twiddle) 446891876 (25 8 76) 34000) ((Zoltan Z. Zolka)
995761234 (1 1 27) 98000))
```

```
28_PP IT
```

```
((Arnie A. Aardvark)
```

```
100101000
```

```
(12 12 85)
```

```
10000)
```

```
((George B. Smith)
```

```
100455467
```

```
(5 6 83)
```

```
17000)
```

```
((Frank G. Halasz)
```

```
331404999
```

```
(15 4 84)
```

```
11000)
```

```
((Ann Q. Merrick)
```

```
345223431
```



```
(14 2 80)
29000)
((Melinda F. Twiddle)
446891876
(25 8 76)
34000)
((Zoltan Z. Zolka)
995761234
(1 1 27)
98000))
```

(IT)

```
29_(LC.SortDatabaseByField (QUOTE TestDB)
```

```
(QUOTE Salary]
```

```
((Arnie A. Aardvark) 100101000 (12 12 85) 10000) ((Frank G.
Halasz) 331404999 (15 4 84) 11000) ((George B. Smith) 100455467 (5
6 83) 17000) ((Ann Q. Merrick) 345223431 (14 2 80) 29000) ((
Melinda F. Twiddle) 446891876 (25 8 76) 34000) ((Zoltan Z. Zolka)
995761234 (1 1 27) 98000))
```

```
30_PP IT
```

```
((Arnie A. Aardvark)
100101000
(12 12 85)
10000)
((Frank G. Halasz)
331404999
(15 4 84)
11000)
((George B. Smith)
100455467
(5 6 83)
17000)
((Ann Q. Merrick)
```

```

345223431
(14 2 80)
29000)
((Melinda F. Twiddle)
446891876
(25 8 76)
34000)
((Zoltan Z. Zolka)
995761234
(1 1 27)
98000))
(IT)

31_(LC.SortDatabaseByField (QUOTE TestDB)
      (QUOTE Name]
(((Arnie A. Aardvark) 100101000 (12 12 85) 10000) ((Frank G.
Halasz) 331404999 (15 4 84) 11000) ((Ann Q. Merrick) 345223431 (14
2 80) 29000) ((George B. Smith) 100455467 (5 6 83) 17000) ((
Melinda F. Twiddle) 446891876 (25 8 76) 34000) ((Zoltan Z. Zolka)
995761234 (1 1 27) 98000))

```

```
32_PP IT
```

```

(((Arnie A. Aardvark)
100101000
(12 12 85)
10000)
((Frank G. Halasz)
331404999
(15 4 84)
11000)
((Ann Q. Merrick)
345223431
(14 2 80)
29000)
((George B. Smith)

```

```
100455467
(5 6 83)
17000)
((Melinda F. Twiddle)
446891876
(25 8 76)
34000)
((Zoltan Z. Zolka)
995761234
(1 1 27)
98000))
(IT)

33_(LC.SortDatabaseByField (QUOTE TestDB)
      (QUOTE StartDate]
(((Zoltan Z. Zolka) 995761234 (1 1 27) 98000) ((Melinda F. Twiddle
) 446891876 (25 8 76) 34000) ((Ann Q. Merrick) 345223431 (14 2 80)
29000) ((George B. Smith) 100455467 (5 6 83) 17000) ((Frank G.
Halasz) 331404999 (15 4 84) 11000) ((Arnie A. Aardvark) 100101000
(12 12 85) 10000))

34_PP IT

(((Zoltan Z. Zolka)
995761234
(1 1 27)
98000)
((Melinda F. Twiddle)
446891876
(25 8 76)
34000)
((Ann Q. Merrick)
345223431
(14 2 80)
29000)
((George B. Smith)
```

100455467

(5 6 83)

17000)

((Frank G. Halasz)

331404999

(15 4 84)

11000)

((Arnie A. Aardvark)

100101000

(12 12 85)

10000))

(IT)

LispCourse #30: Conceptual Models for Atoms, Lists *et al.*

Introduction

So far, we've been talking about data objects in Lisp in terms of how they appear to us as programmer's, i.e., in terms of how we type them in and how they are printed on the screen.

Example: We defined a list as a thing that begins with a "(", followed by one or more atoms or lists, and terminated by a ")".

This way of talking is not wrong, but it does not reflect how Lisp itself "thinks" about the various data objects.

In particular, Lisp translates the data objects that we type-in into an internal representation. All functions dealing with these data objects then work on this internal representation of the objects. Only when it comes time to print out some result, does Lisp translate the internal representation back into the external representation (i.e., into the *print name* described in LispCourse #28, page 9).

You can't get very far in Lisp programming without understanding something about the underlying internal representations for atoms, lists, etc.

For example, the semantics of many Lisp functions can be expressed only in terms of this underlying representation. There are many examples of this below.

There are many levels at which one could describe the internal representation of Lisp data objects.

For example, the hardware actually processes bits that represent numbers and addresses in its memory.

However, here we will discuss internal representation at a *conceptual* level.

The goal is to provide a conceptual model of Lisp data that provides all of the necessary concepts and mechanism for understanding the semantics of Lisp functions, BUT without going into the grubby details of how the data is actually implemented at the hardware/microcode level.

Everything in Lisp is a *Pointer*

The first lesson is that everything in Lisp is a *pointer*.

A *pointer* is simply a one-way connection between two data objects. If A points to B, then we can get to B from A, but not vice versa.

When we say that *atom A has the value 5*, Lisp represents this by a pointer between the thing that is the atom **A** and the thing that is the atom **5**.

The atom A points to the atom 5

A \longrightarrow **5**

Similarly, when we represent the fact that *the value of the atom NewList is the list (1 2 3)*, Lisp represents that by a pointer between the atom **NewList** and the thing that represents the list **(1 2 3)**.

The atom NewList points to the list (1 2 3)

NewList \longrightarrow **The thing: (1 2 3)**

Also, when we say *function Foo returns a list*, we actually mean that function Foo returns a pointer to a list.

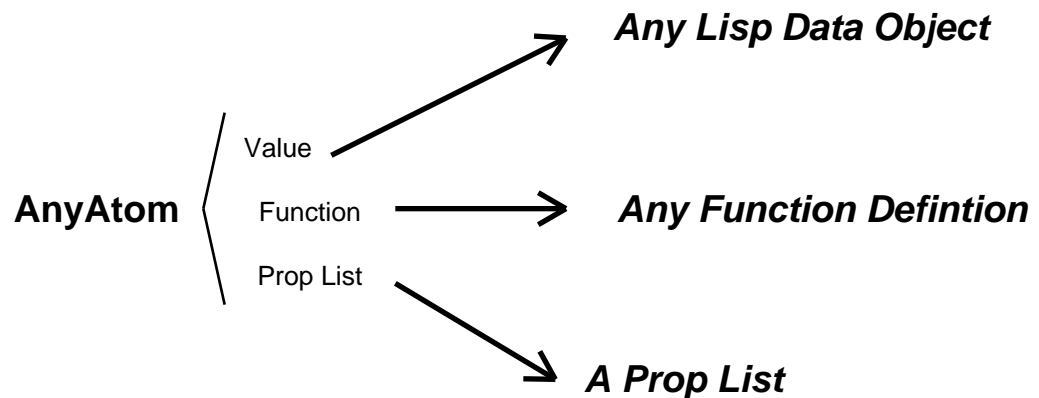
In some sense, you seldom actually hold a Lisp data object in your hands when you program in Lisp. It's more like you are holding one end of a rope in your hand. The other end of the rope is a Lisp data object. You operate in Lisp by passing around the free end of this rope, e.g., functions return to you a free end of the rope which you then pass onto other functions, etc.

Representing Atoms

In our conceptual model, atoms will be represented by their print name. So to represent the atom *ManyWordAtom*, we simply type use **ManyWordAtom**.

An atom can have three pointers coming from it, representing its *value*, its *prop list* and its *function definition*.

Example:



The SET functions (SET, SETQ, SETQQ) can be used to establish the pointer between an atom and its value.

For example, (SETQ A 'B) sets up a (value) pointer between the atom A and the atom B.

The function DEFINEQ establishes the pointer between an atom and a function definition.

For example, (DEFINEQ (MyFunc (LAMBDA NIL (PLUS 1 3)))) sets up a (function) pointer between the atom MyFunc and the given function definition (LAMBDA ...).

The function SETPROPLIST establishes the pointer between an atom and a prop list object.

For example, (SETPROPLIST 'MyAtom '(Size 5)) sets up a (prop list) pointer between the atom MyAtom and the given prop list.

Note: The functions PUTPROP and GETPROP add to and retrieve from from the prop list pointed to the given atom.

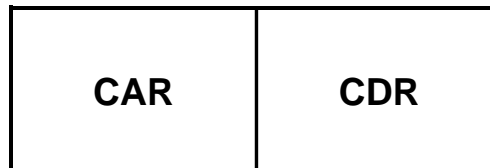
Representing Lists

The CONS cell

Lists are constructed from basic building blocks called *CONS cells*.

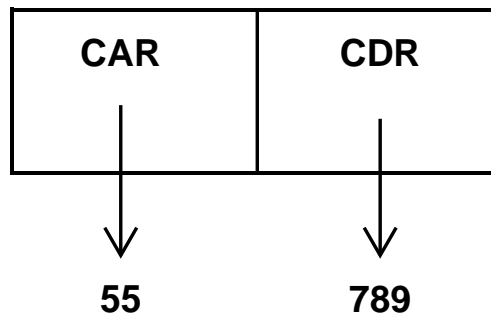
A *CONS cell* can be represented by a box divided into two halves. The left half is called the *CAR* and the right half is called the *CDR*.

A CONS cell



Both the CAR and the CDR portions of a CONS cell contain pointers to other Lisp data objects.

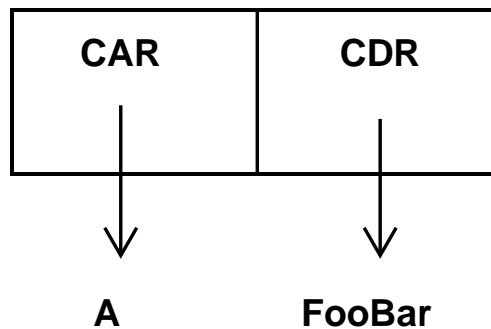
A CONS cells is printed as a *dotted pair*, (*X . Y*), where *X* is the print name of the thing pointed to by the CAR of the CONS cell and *Y* is the print name of the thing pointed to by the CDR.

The CONS cell: (55 . 789)

Creating a CONS cell is done using the function **CONS**.

Example: **(CONS 55 789)** prints the result **(55 . 789)** and actually creates a CONS cell like the one shown above.

Second example: **(CONS 'A 'FooBar)** prints the result **(A . FooBar)** and actually creates a CONS cell like the one shown below.

The result of (CONS 'A 'FooBar)**Lists are built from CONS cells**

Lists are constructed from CONS cells in the following manner:

There is a CONS cell for each item in the list.

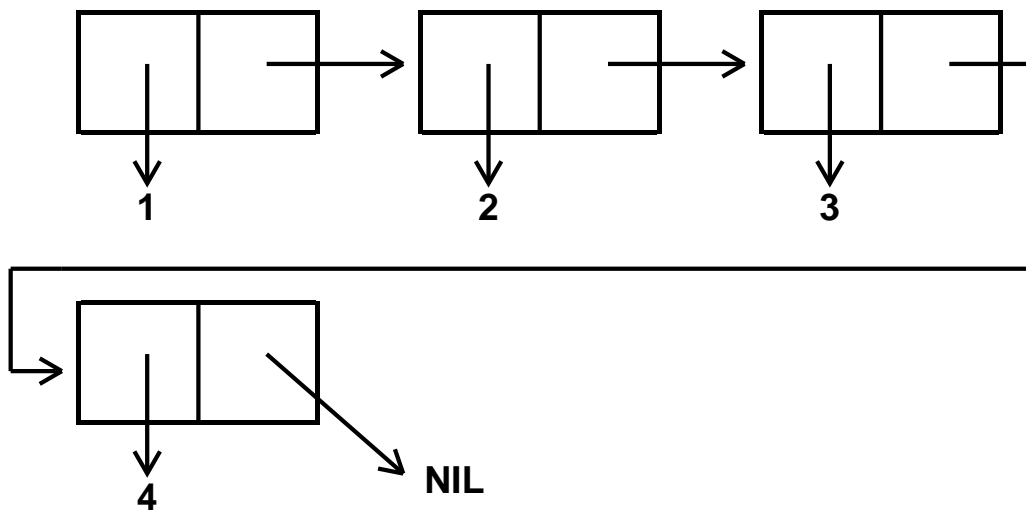
The CAR of the CONS cell points to the item.

The CDR of the CONS cell points to the CONS cell for the next item in the list.

The CDR of the CONS cell for the last item in the list (i.e., where there is no next item) points to the atom NIL.

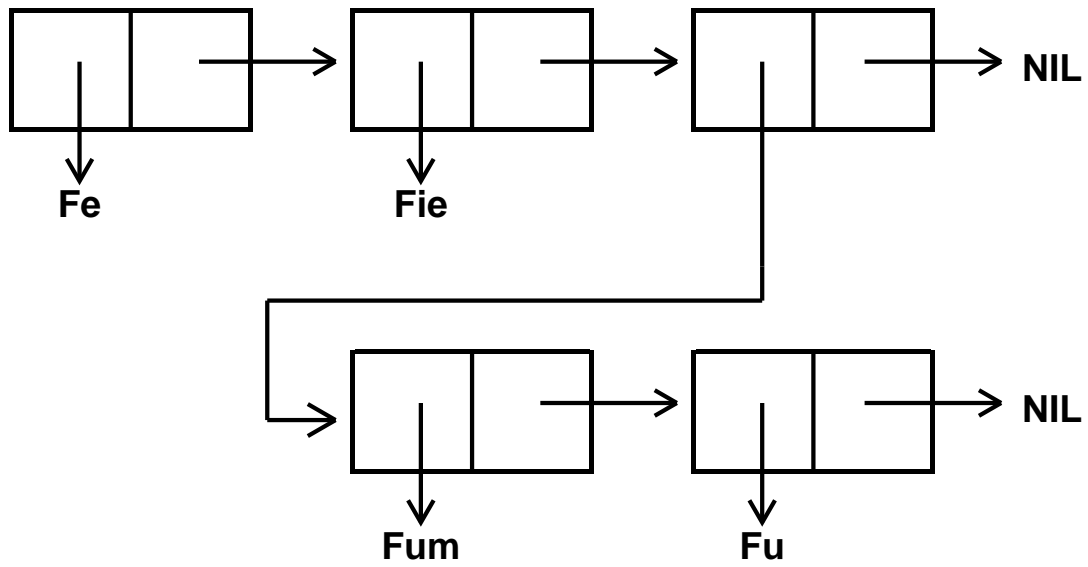
Example: Representation of the list (1 2 3 4)

The list (1 2 3 4)



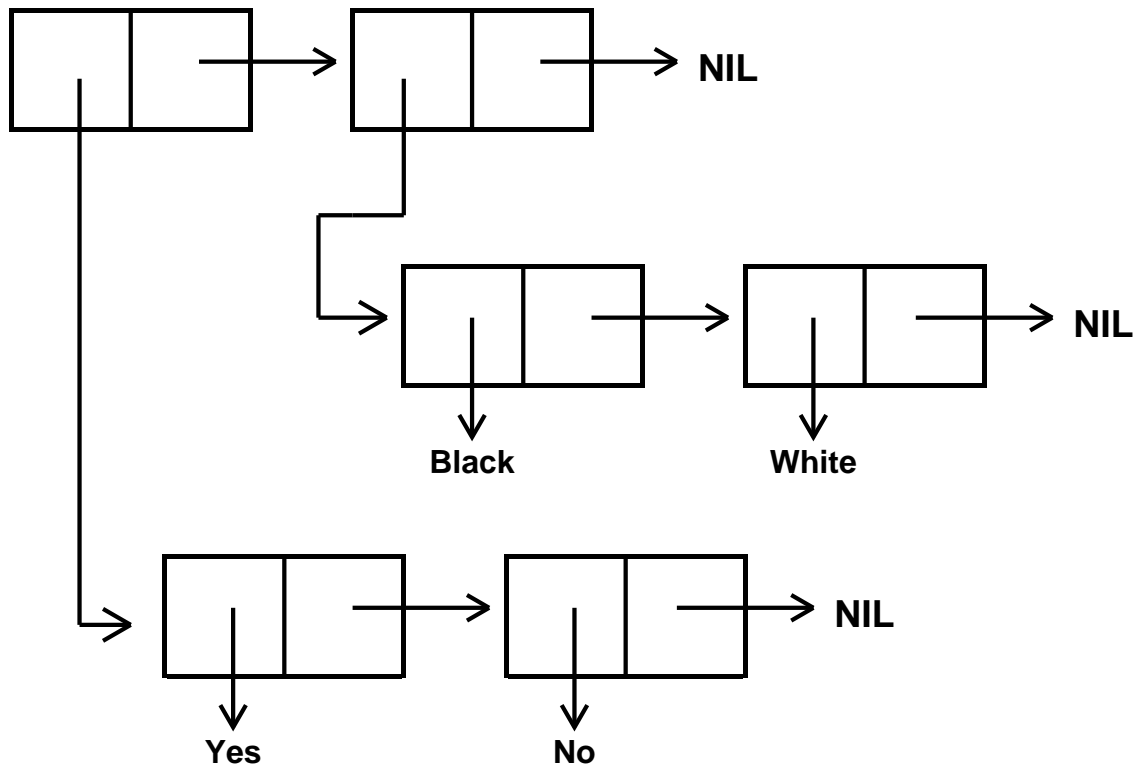
Second example: Representation of the list (*Fe Fie (Fum Fu)*)

The list (*Fe Fie (Fum Fu)*)



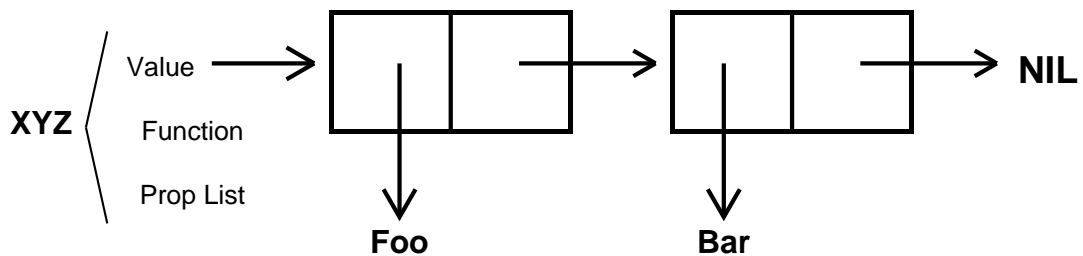
Third example: Representation of the list $((Yes\ No)(Black\ White))$

The list $((Yes\ No)(Black\ White))$



Fourth example: The result of $(SETQ\ XYZ\ '(Foo\ Bar))$

The result of $(SETQ\ XYZ\ '(Foo\ Bar))$



CAR and CDR revisited

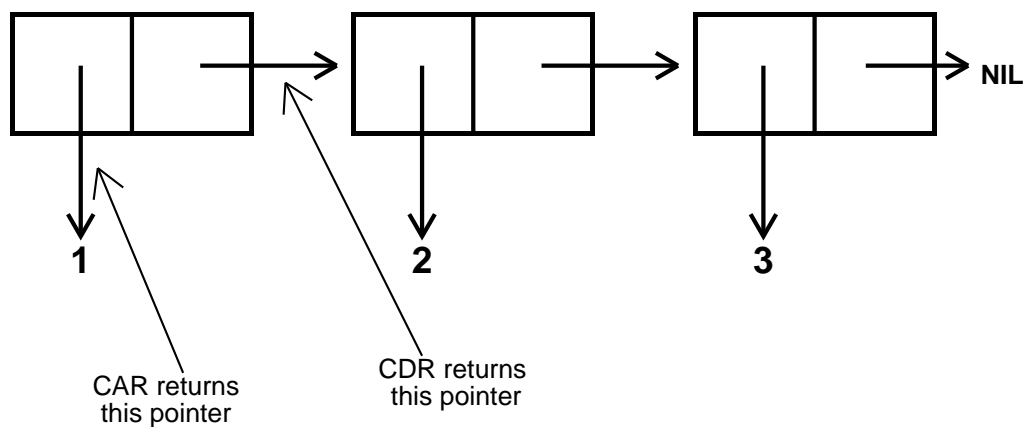
CAR and CDR can be defined in terms of CONS cells as follows:

CAR \checkmark returns contents of the CAR part of the CONS cell pointed to by its argument.

CDR \checkmark returns contents of the CDR part of the CONS cell pointed to by its argument.

Note: when the CONS cell is part of list, then the CDR part points to the CONS cell that begins the rest of the list as per our previous definition of CDR.

The list (1 2 3)



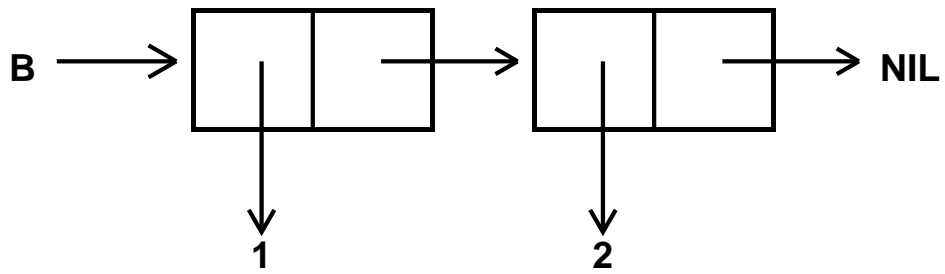
CONSing onto a list and LIST revisited

Recall that CONS creates a new CONS cell with the first argument as its CAR and the second argument as its CDR.

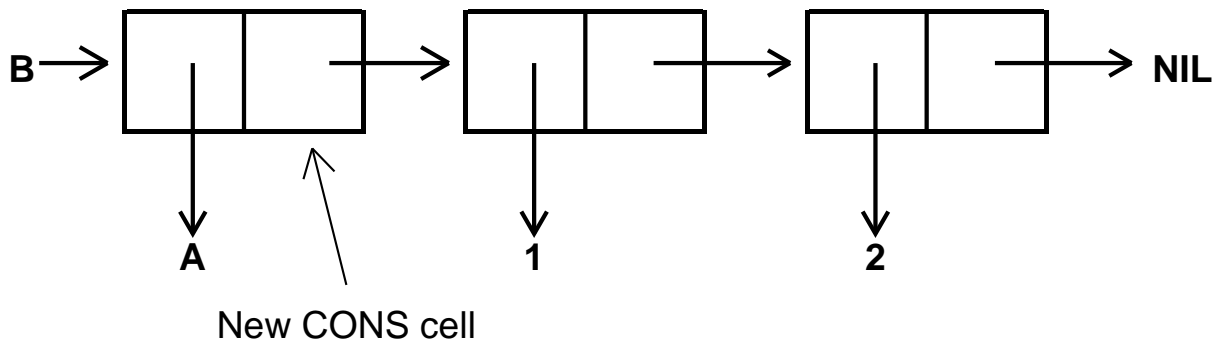
When the second argument is a list (i.e., a pointer to the first CONS cell in a list) then the result is a new list with a different CONS cell at the head of the list.

Example: `(SETQ B (1 2))` then `(SETQ B (CONS 'A B))`

After (SETQ B (1 2))

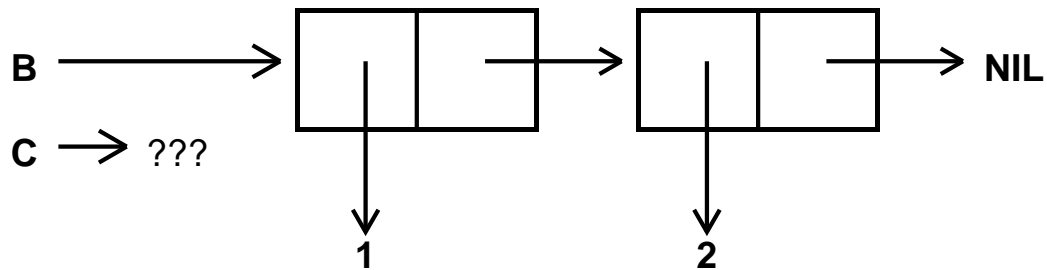


After (SETQ B (CONS 'A B))

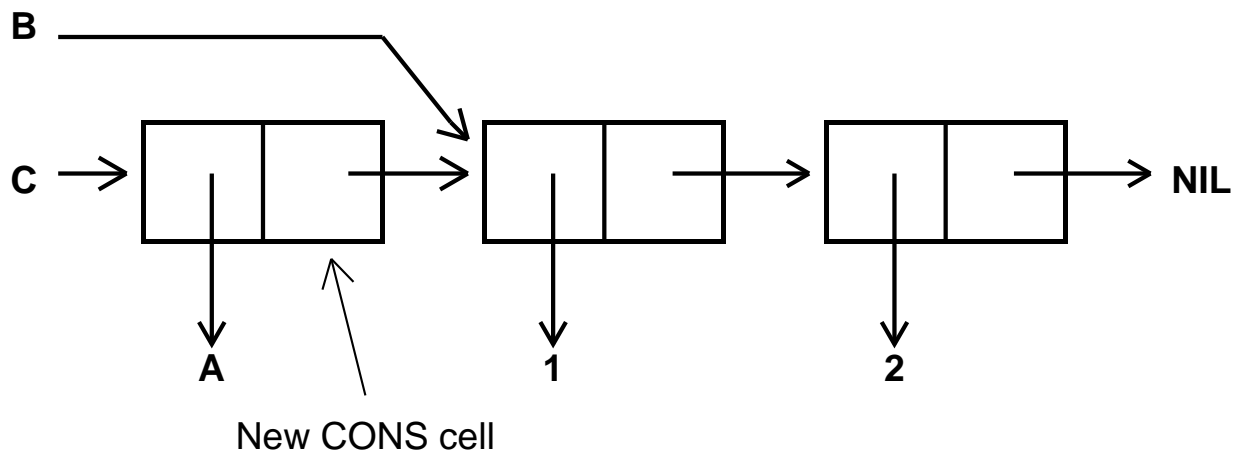


Second example: `(SETQ B (1 2))` then `(SETQ C (CONS 'A B))`

After `(SETQ B (1 2))`



After `(SETQ C (CONS 'A B))`



The function *LIST* constructs a list with its arguments as the items in the list. Example: `(LIST 1 2 3 '(3 4))` returns the list `(1 2 3 (3 4))`. *LIST* works by successive *CONSES*, i.e., by building the list one *CONS* cell at a time.

EFS: Define the function `NewLIST` using the *CONS* function.

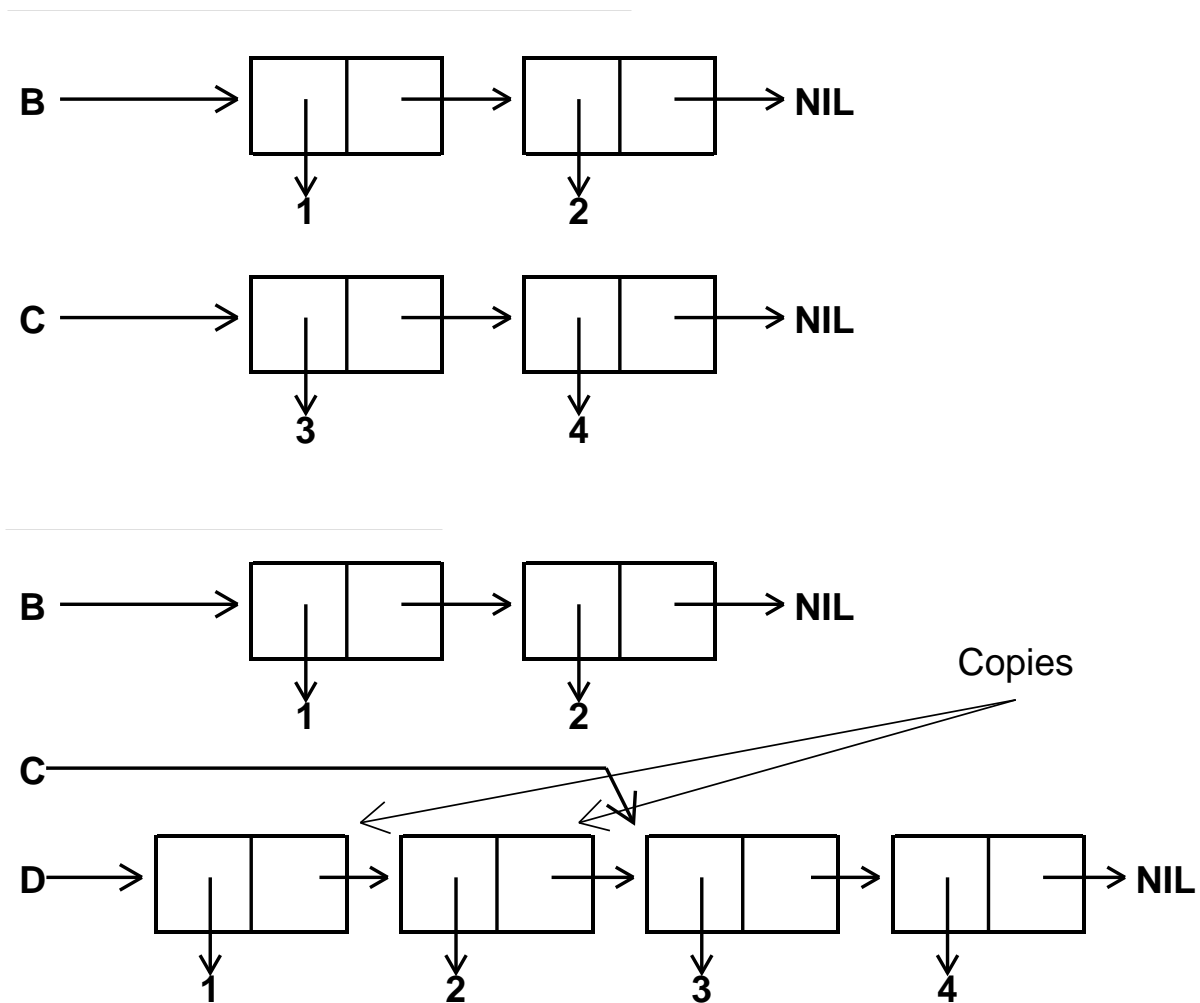
APPEND revisited

APPEND works very differently from *CONS*.

In particular, *APPEND* *copies* all but the last list it is appending before it does the append.

It then changes the last CDR cell in each copy to point to the head of the next copy rather than to NIL.

Example: $(SETQQ B (1 2))$, $(SETQQ C (3 4))$ then $(SETQ D (APPEND B C))$



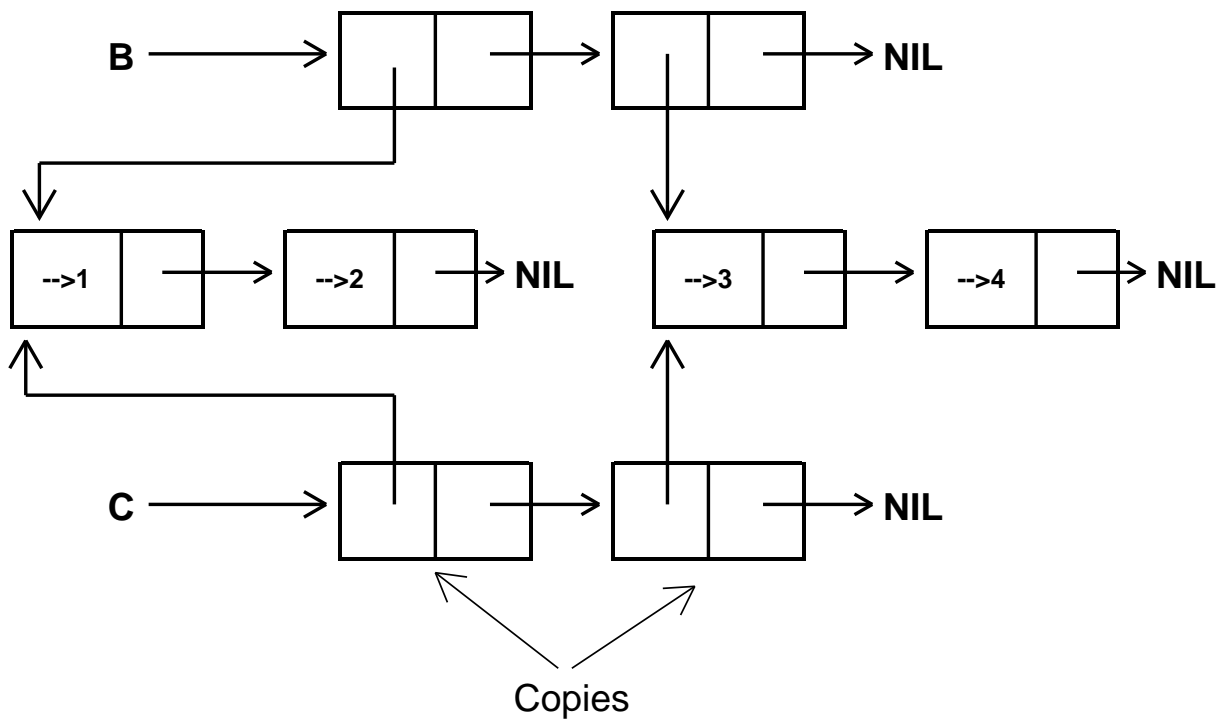
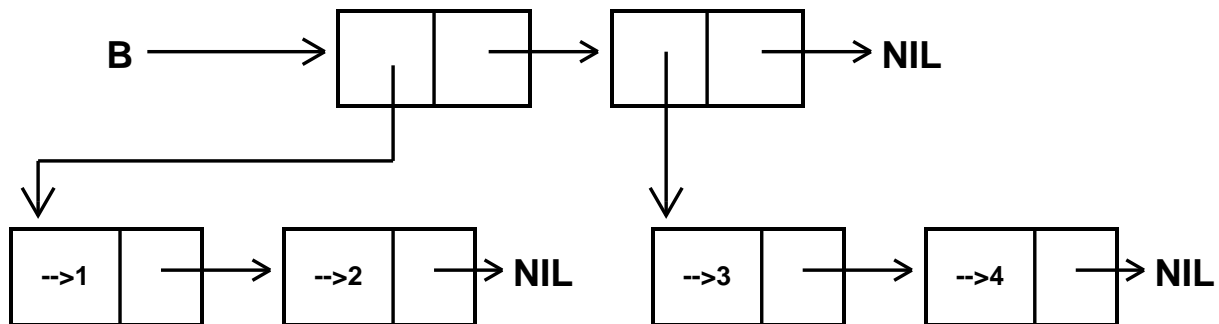
The COPY function

The function `COPY` makes a copy of a list by copying all the `CONS` cells in the list.

`APPEND` uses `COPY` to make its copies.

Note: COPY copies only the top-level of a list. If the CAR of any CONS cell points to another list, then that list is NOT copied.

Example: (SETQQ B ((1 2) (3 4))) then (SETQ C (COPY B))



RPLACA, RPLACD, and NCONC -- The destructive functions

RPLACA, RPLACD, and NCONC are three functions that "do surgery on lists." Unlike APPEND, they don't make copies and then carry out actions on the copies, rather they actually change the list they are passed as an argument. Because of this they are dangerous functions and should be used with some care!!!!

(RPLACA X Y) replaces the CAR of the CONS cell X with a pointer Y.
RPLACA returns X.

Example:

```
1_(SETQ A (1 2 3))
```

```
(1 2 3)
```

```
2_(RPLACA A '(7 8))
```

```
((7 8) 2 3)
```

Note: No SETQ is necessary because the actual list is changed.

```
3_ A
```

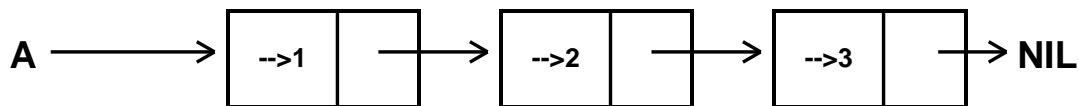
```
((7 8) 2 3)
```

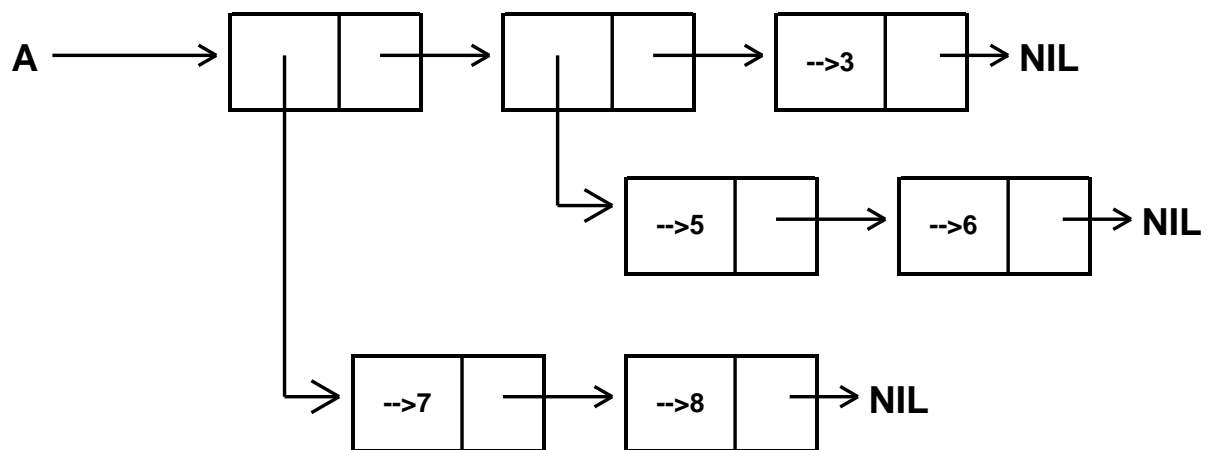
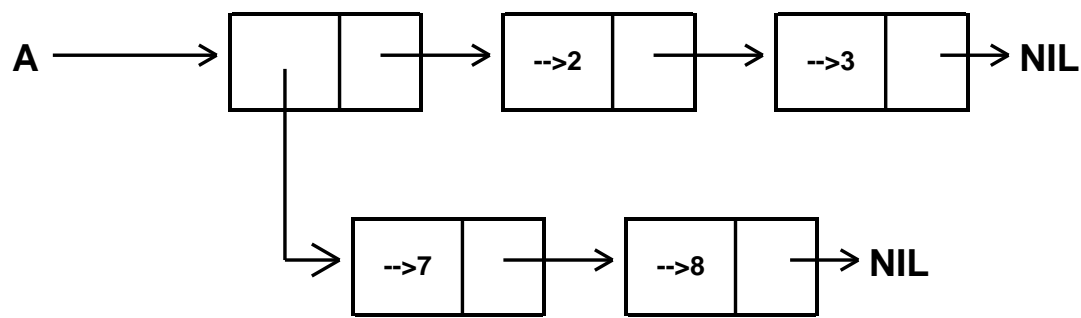
```
4_(RPLACA (CDR A) '(5 6))
```

```
((5 6) 3)
```

```
5_ A
```

```
((7 8) (5 6) 3)
```





(RPLACD X Y) replaces the CDR of the CONS cell *X* with a pointer *Y*.
RPLACD returns *X*.

Example:

```
6_(SETQ A (1 2 3))
```

```
(1 2 3)
```

```
7_(RPLACD A '(7 8))
```

```
(1 7 8)
```

Note: No SETQ is necessary because the actual list is changed.

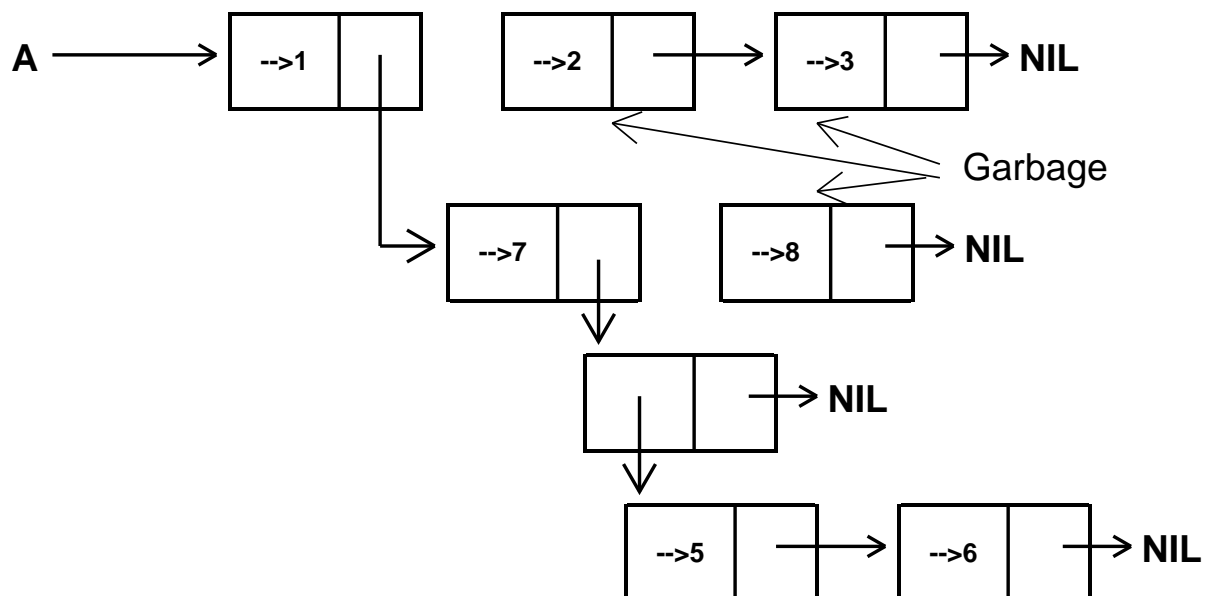
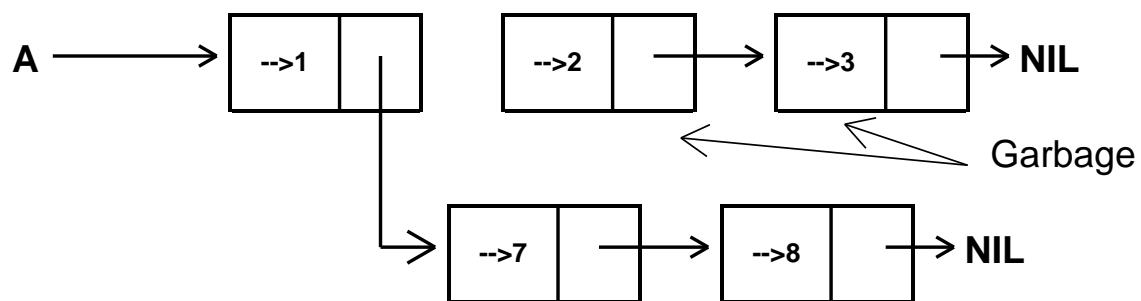
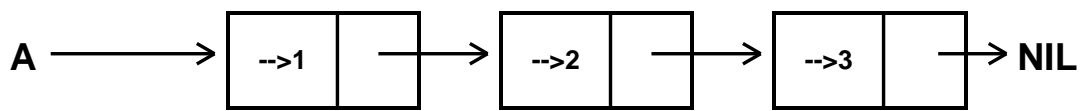
```
8_ A
```

```
(1 7 8)
```

```
9_ (RPLACD (CDR A) '((5 6)))
```

```
(7 (5 6))
```

```
10_ A
```

$(1\ 7\ (5\ 6))$ 

(NCONC X1 X2 X3 ...) Like APPEND, but does not copy the lists before doing the list splicing.

Example:

11_(SETQQ B (1 2))

(1 2)

12_(SETQQ C (3 4))

(3 4)

13_(NCONC B C)

(1 2 3 4)

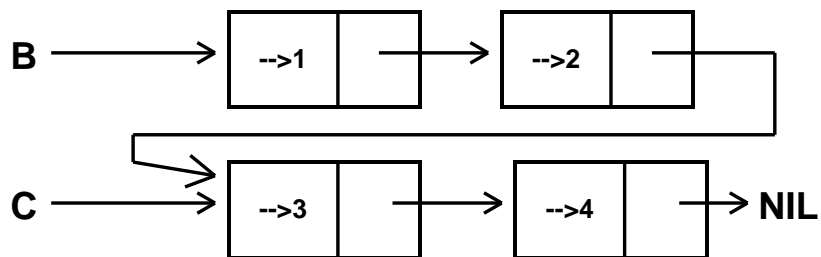
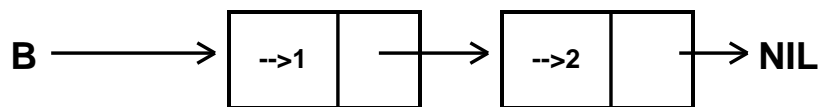
Note: No SETQ is necessary because the actual B list is changed.

14_ B

(1 2 3 4)

15_ C

(3 4)



Some dangers in using the destructive functions

If you are not careful, RPLACA, RPLACD, and NCONC can cause many strange things to happen to your list structures.

Two of typical strange things are changing things you didn't mean to change and circular list structures.

Example: Changing what you didn't mean to change.

Plan: you are going to change A, so you hold on to the original by SETQing B to the original value of A.

```
23_(SETQQ A (1 2 3))
```

```
(1 2 3)
```

```
24_(SETQ B A)
```

```
(1 2 3)
```

```
25_(NCONC A '(4 5))
```

```
(1 2 3 4 5)
```

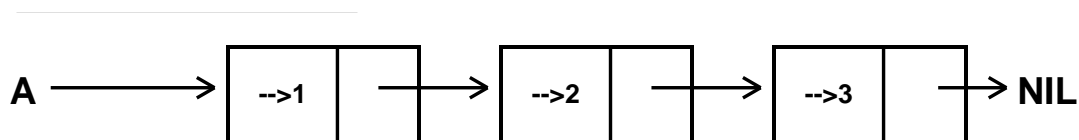
```
26_ A
```

```
(1 2 3 4 5)
```

```
27_ B
```

```
(1 2 3 4 5)
```

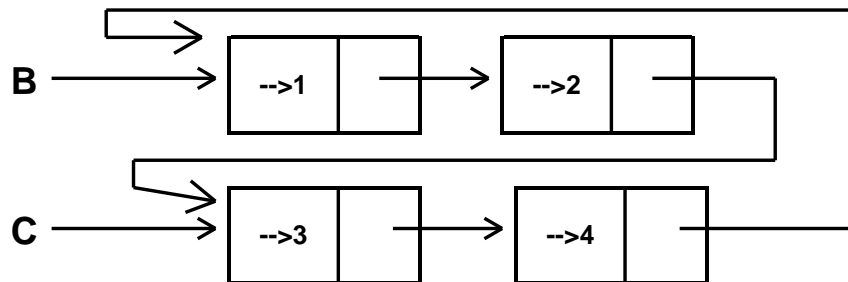
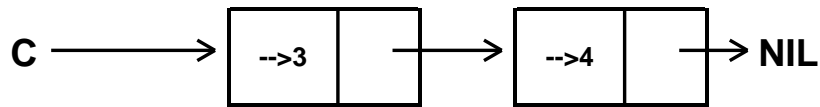
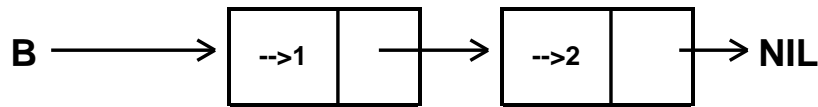
Note that the NCONC effects the value of B even though it is not mentioned at all in the NCONC function call.



4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
 4 1 2 3 4 ... forever!!!!!!

46_C

(3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1
 2 ... forever!!!!!!



Representing Datatypes and Arrays

Datatypes and Arrays can be modelled as a fixed-length one-dimensional vector of cells, each of which contains a pointer to some Lisp data object. *Note: the cells are NOT CONS cells because they contain only a single pointer.*

For arrays, the cells are indexed by number. For Datatypes, the cells are indexed by field name.

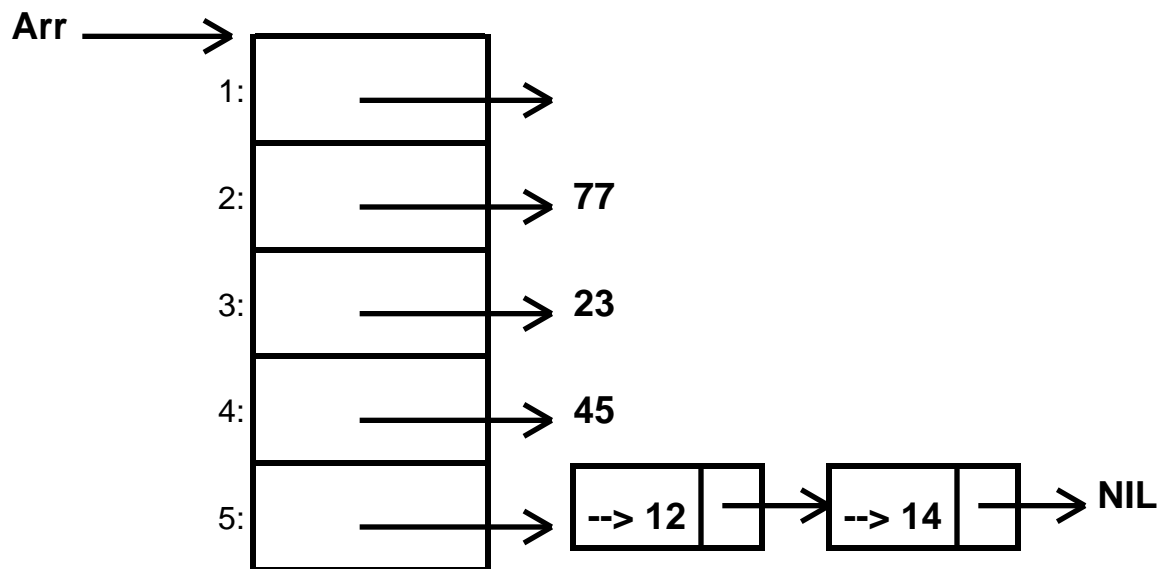
Example:

```
55_(SETQ Arr (ARRAY 5))
```

```
{Array}#6,12345
```

```
56_(FOR X IN '(55 77 23 45 (12 14)) AS I FROM 1 DO (SETA Arr I X))
```

```
NIL
```



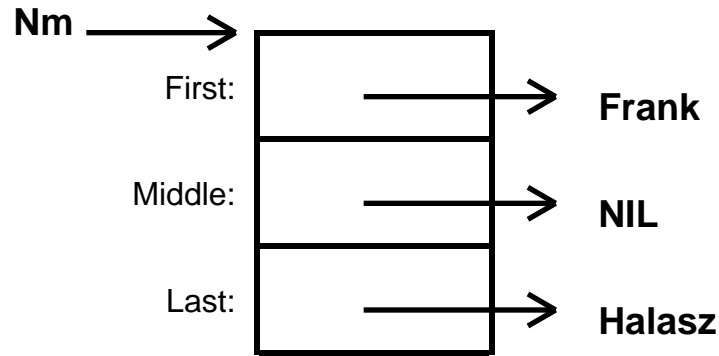
Second example:

```
34_(DATATYPE LC.Name (First Last Middle))
```

```
LC.Name
```

```
35_(SETQ Nm (CREATE LC.Name First _ Frank Last _ Halasz))
```

```
{LC.Name}#34,12378
```



Representing Strings

The internal representation of a string has two parts: a *string pointer* and a *character vector*.

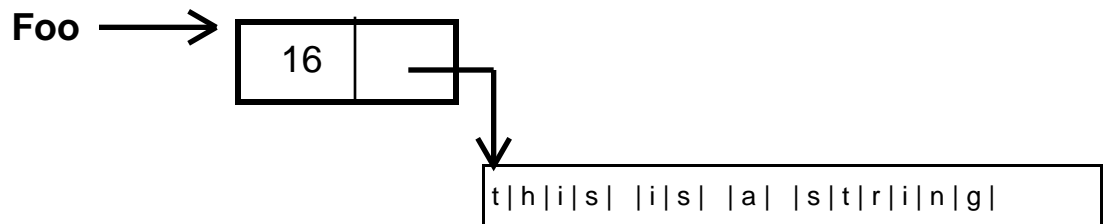
The *string pointer* is a cell (not a CONS cell) containing two things:

1. The number of characters in the string.
2. A pointer to the start of the characters for this string in the character vector.

The *character vector* is just a sequence of characters.

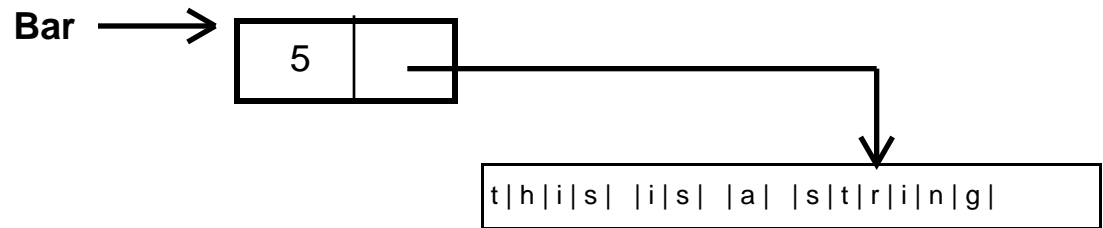
A pointer to a string is a pointer to the string's string pointer.

Example: `(SETQ Foo "this is a string")`



There can be more characters in the character vector than there are in the string.

Example: `(SETQ Bar (SUBSTRING Foo 11))` returns *"string"*.



Some functions that create and return strings create only a new string pointer and use an old character vector. Other functions create both a new string pointer and a new character vector.

As shown above, `SUBSTRING` creates just a new string pointer and uses the old character vector.

`MKSTRING` and `ALLOCSTRING` create both string pointers and character vectors.

There are destructive string functions analogous to `RPLACA`, etc. for lists. These are `GNC`, `GLC`, and `RPLSTRING`.

GNC returns the first character of a string (as an atom) and then removes that character from the string by updating the string pointer. If the argument is not a string, it is made into a string using its print name.

Example:

```
12_(SETQ S "abcd")
```

```
"abcd"
```

```
13_(GNC S)
```

```
a
```

```
14_S
```

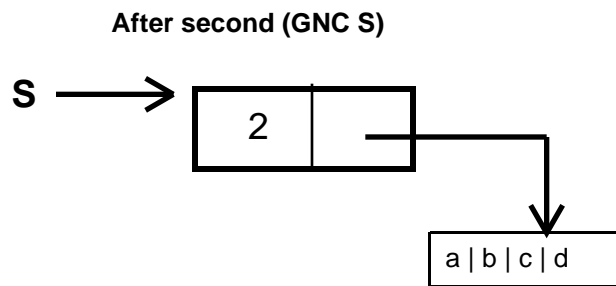
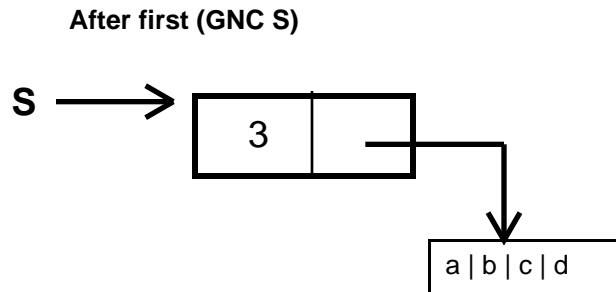
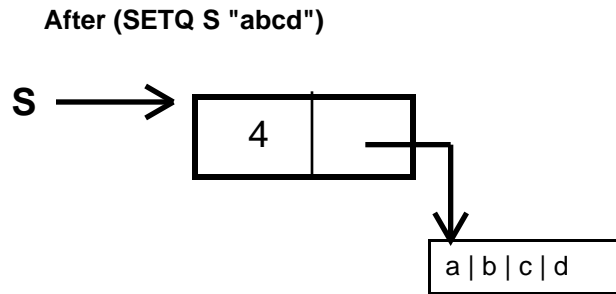
```
"bcd"
```

```
15_(GNC S)
```

```
b
```

```
16_S
```

```
"cd"
```



GLC ž returns the last character of a string (as an atom) and then removes that character from the string by updating the string pointer. If the argument is not a string, it is made into a string using its print name.

Example:

```
12_(SETQ S "abcd")
```

```
"abcd"
```

```
13_(GLC S)
```

```
d
```

```
14_S
```

```
"abc"
```

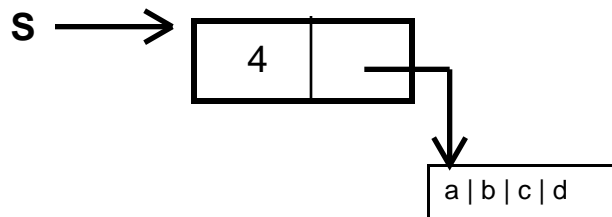
```
15_(GLC S)
```

c

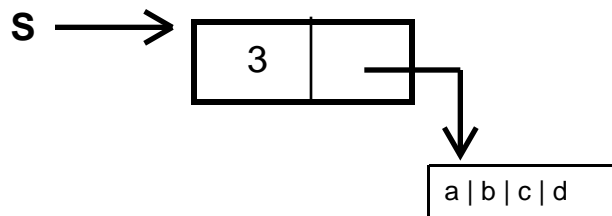
16_ S

"ab"

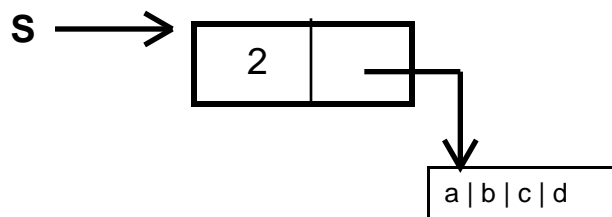
After (SETQ S "abcd")



After first (GLC S)



After second (GLC S)



(RPLSTRING *String1* *N* *String2*) *z* changes the character vector of *String1* to include the characters in *String2*, starting at position *N* in the *String1*. *N* can be negative just as in `SUBSTRING`. If *String1* and/or *String1* are not strings, they are made into strings using their print names.

Example:

12_(SETQ S "abcd")

"abcd"


```
13_(SETQ R (SUBSTRING S 2))
```

```
"bcd"
```

```
14_(RPLSTRING S 2 "xy")
```

```
"axyd"
```

```
15_ S
```

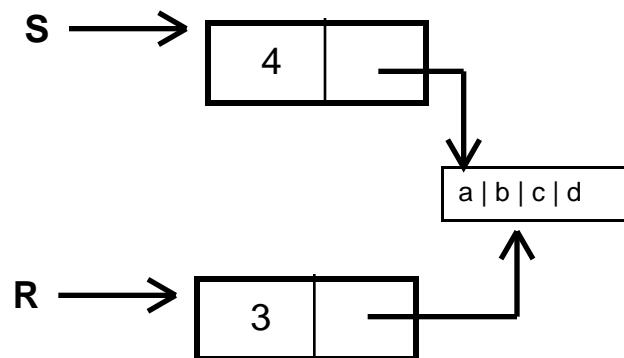
```
"axyd"
```

Note: since RPLSTRING is destructive, R is messed up too.

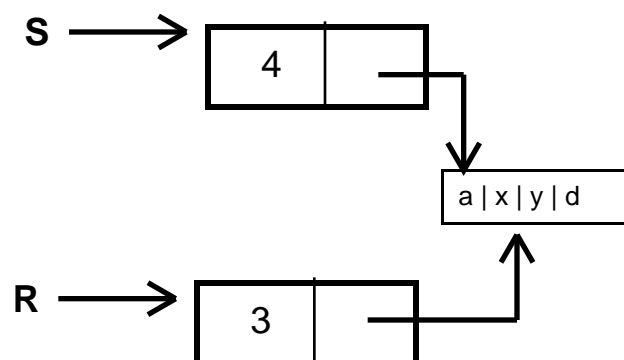
```
16_ R
```

```
"xyd"
```

After (SETQ S "abcd") and (SETQ R (SUBSTRING S 1))



After (RPLSTRING S 2 "xy")



Sameness and Equality in Lisp: EQ versus EQUAL

Consider the following:

```
1_(SETQ A (LIST 1 2 3))
```

```
(1 2 3)
```

```
2_(SETQ B (LIST 1 2 3))
```

```
(1 2 3)
```

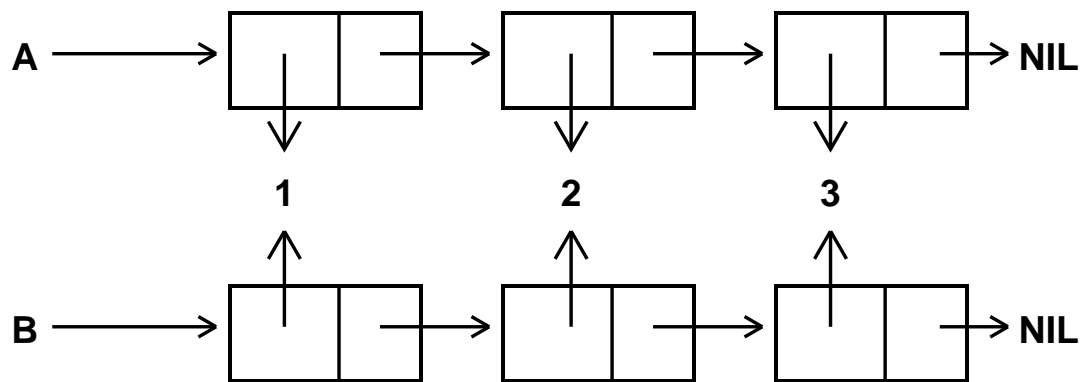
Do A and B have the same value? Are the values of A and B equal?

By most peoples definition, the answer is *Yes*. But in Lisp, the answer is it depends what you mean by *equal*.

The values of A and B are equal in the sense that they are both lists containing the identical sequence of items.

The values of A and B are NOT equal in the sense that they are made up of entirely different CONS cells. This is because the function LITS works by creating new CONS cells to make a list out of its arguments (see above).

The situation is clear if you diagram the two lists.



Contrast the previous situation with the following situation:

```
1_(SETQ A (LIST 1 2 3))
```

```
(1 2 3)
```

```
2_(SETQ B A)
```

```
(1 2 3)
```

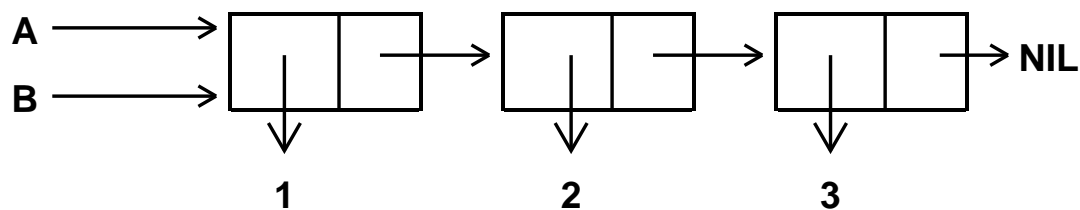
Do A and B have the same value? Are the values of A and B equal?

By most peoples definition, the answer is *Yes*. And in Lisp, the answer is *Yes*.

The values of A and B are equal in the sense that they are both lists containing the identical sequence of items.

The values of A and B are also equal in the sense that they are made up of exactly the same CONS cells. This is because the function SETQ just sets the value of its first argument to be its second argument without creating any new structures (see above).

The situation is clear if you diagram the list.



Lisp has two concepts of sameness or equality:

- 1) containing the same "information"
- 2) being exactly the same data objects

Data objects in Lisp can be equal in the first sense without being equal in the second sense. The opposite is NOT true – identical data object always contain the same "information".

EQ ž returns T if its two arguments are pointers to the exact same internal data object. NIL, otherwise.

EQUAL ž returns T if its two arguments are the same type of data object and contain the *same information*. NIL, otherwise.

Same information is determined as follows:

- 1) The two arguments are EQ
- 2) The two arguments are equal numbers

3) The two arguments are strings that are STREQUAL (have the same sequence of characters).

4) The two arguments are lists, where the CARs are EQUAL and the CDRs are EQUAL.

Basically, two lists are EQUAL if they contain the same set of atoms within the same list format.

Examples:

The results of (LIST 1 2 3) and (LIST 1 2 3) are EQUAL but not EQ.

The results of (LIST 1 (LIST 2 3)) and (LIST 1 (LIST 2 3)) are EQUAL but not EQ.

The results of (MKSTRING 'AB) and (MKSTRING 'AB) are EQUAL but not EQ.

Following (SETQ A B), (EQ A B) and (EQUAL A B) return T.

If A points to a list, (EQ (CDDR A) (CDR (CDR A))) is T.

If A points to a list, (EQ (CDR (CONS 1 A)) A) is T.

If A points to a list, (EQ (RPLACA A 1) A) is T.

If A points to a list, (EQ (CONS 22 A)(CONS 22 A)) is NIL, but (EQUAL (CONS 22 A)(CONS 22 A)) is T.

(EQ 'Atom 'Atom) is always T.

For integers less than 65,000, (EQ *SmallInteger SmallInteger*) is T.

For larger integers and for floating numbers, (EQ *Number Number*) is generally NIL, but (EQUAL *Number Number*) is T.

As a general rule, use EQUAL unless you know you want to test for the same exact data structure or you are comparing atoms.

References

CONS cells and lists represented as CONS cells is covered in:

Winston & Horn, Chapter 9

Touretzky, Chapter 2

RPLACA, et al. are covered in Section 2.5 of the IRM.

GNC et al. are covered in Section 2.6 of the IRM.

EQ et al. are covered in Section 2.2 of the IRM and in Touretzky, page 155 and in Winston and Horn, page 142.

Exercises

Attached.

LispCourse #31: Solutions for Homework #30, Problems 2 and 3

Sample solutions attached.

LispCourse #32: Homework on circular queues

Exercise

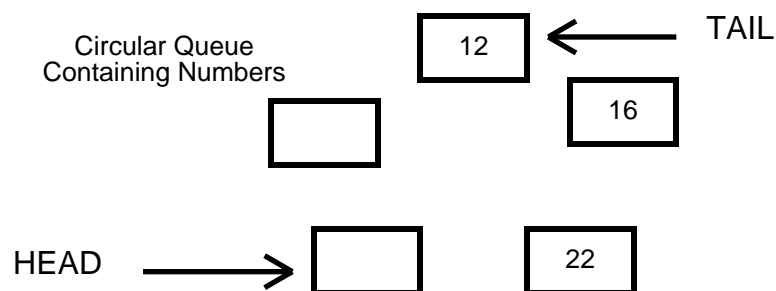
Overall problem

Write the functions to manage a circular queue. Randomly add and delete random numbers to and from this queue. Print out the contents of the queue after each addition or deletion.

Defintions

A *queue* is a data structure that can hold from 1 to N other data objects. The queue has a beginning (called its *head*) and an end (called its *tail*). You can add new data data items to the queue (as long as you don't exceed N items) and you can remove data items from the queue. New data objects are always added to the tail of the queue and old data items are always removed from the head of the queue, just like in the queues at banks, airline counters, etc.

A *circular queue* is one where the N places for data objects are arranged in a circle. There is a pointer for the head and a pointer for the tail. When you add a new data object, you put it into the place indicated by the head pointer, and then move the head pointer clockwise by one. When you remove a data object, you take the one indicated by the tail pointer, and then move the tail pointer clockwise by one. You can keep adding and deleteing elements until the head pointer equals the tail pointer, in which case the queue is either empty or full, depending on whether you just removed or added an item.



Subproblems

- 1) Write a function that produces a list of N NILs.
- 2) Write a function that creates a circular list out of a non-circular list by making the CDR of the last CONS cell in the list point to the CAR of the first CONS cell in the list. (Caution: You will have to ^D when Lisp starts printing the result of this function otherwise it will go on forever.)
- 3) Use the functions from 1 and 2 to create a circular queue (i.e., list) containing all NILs. (See caution from 2). SETQ LC.Queue to this circular list.
- 4) Set up the pointers LC.Head and LC.Tail. Originally both should point to the same CONS cell that LC.Queue does, indicating that the queue is empty.
- 5) Diagram the CONS cells in the queue and the LC.Queue, LC.Head, and LC.Tail pointers. Use this diagram to help you in the following problems.
- 6) Write a function that adds a number to the tail of the queue. Be sure to check first that the queue is not full. If it is return NIL, otherwise return T. (Hint: RPLACA and CDR are critical here).
- 7) Write a function that removes a number from the head of the queue. Be sure to check first that the queue is not empty. If it is, return NIL. (Hint: CAR and CDR are critical here).
- 8) Write a function the prints all the numbers in the queue. (Hint: Produce a list starting at the tail and going to the head, then reverse this list and print it.)
Note: Use the function (**PRIN1 X**) to print a X in the Exec window. Use (**TERPRI**) to print end a line.
- 9) Write a function that randomly excercises the queue as follows:

Repeat the following 100 times.

Print the queue.

Generate a random integer between 0 and 1, inclusive.

If this random integer is 0, then remove an item from the queue and print it preceded by an appropriate message. If the delete

fails because the queue is empty, just print an appropriate message.

If this random integer is 1, then generate a second random integer between 1 and 99. Add this second random integer to the queue and print the number preceded by an appropriate message. If the add fails because the queue is full, just print an appropriate message.

Note: The function call (**RAND** N M) generates a random integer between N and M .

LispCourse #33: Solutions for Homework #32

Note

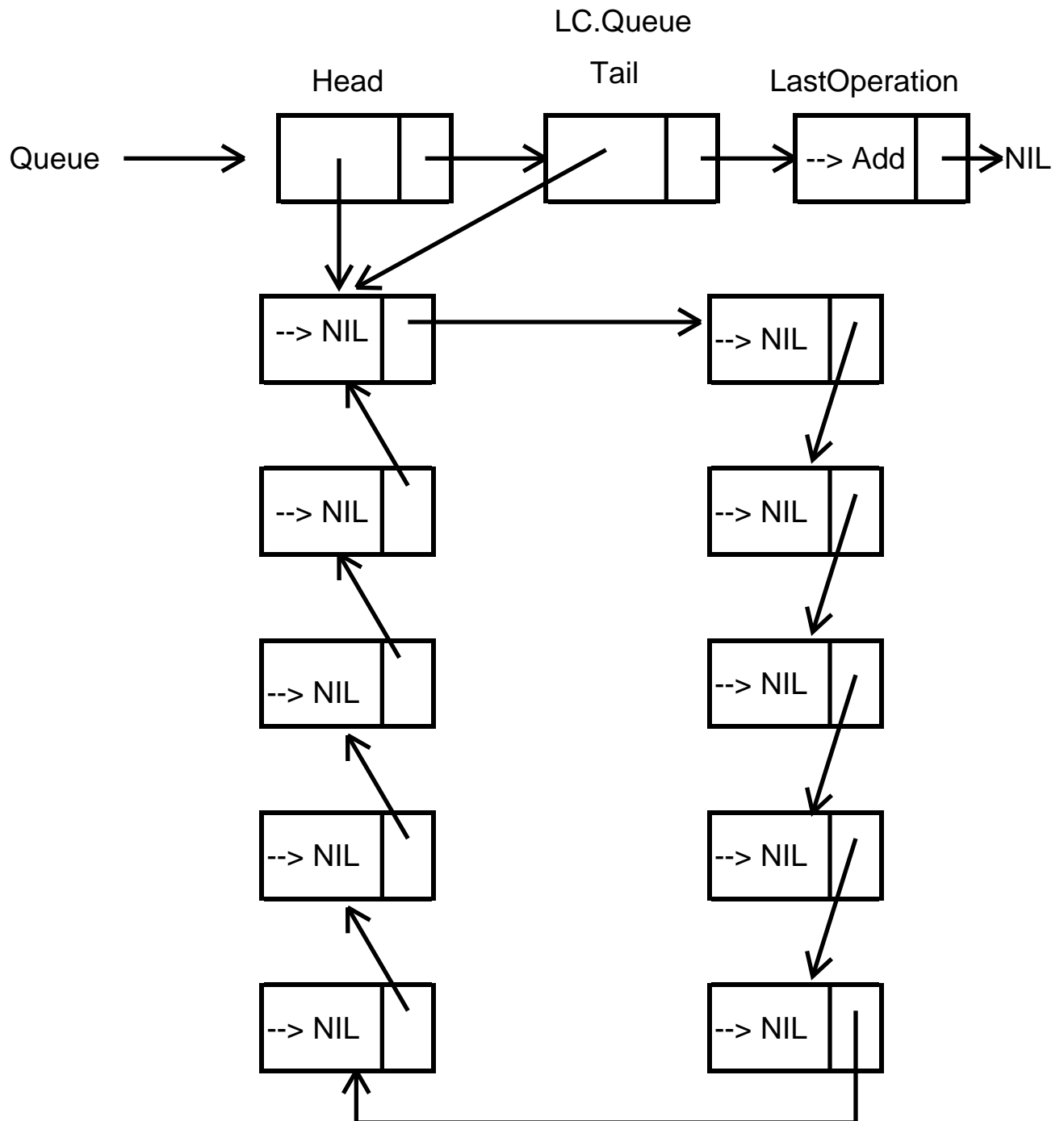
This solution differs slightly from the assignment. Instead of keeping global pointers called LC.Head and LC.Tail, I created a record called LC.Queue with Head and Tail fields (as well as a field to indicate the last operation done on the queue to be used in the empty and full predicates). This scheme allows multiple queues to be created at once, each with its own head and tail pointers.

Solutions to #1 thru 4, and 6 thru 9

Attached

Solution to #5

Result of (SETQ Queue (LC.MakeCircularQueue 10))



Test Run of Exerciser

```
83_(LC.ExerciseQueue 100 (LC.MakeCircularQueue 30))
```

```
ITERATION # 1
Queue is empty!
Adding Item: 17
ITERATION # 2
Head of Queue: 17 :Tail of Queue
Adding Item: 71
ITERATION # 3
Head of Queue: 17 71 :Tail of Queue
Adding Item: 9
ITERATION # 4
Head of Queue: 17 71 9 :Tail of Queue
Adding Item: 13
ITERATION # 5
Head of Queue: 17 71 9 13 :Tail of Queue
Adding Item: 5
ITERATION # 6
Head of Queue: 17 71 9 13 5 :Tail of Queue
Removing Item: 17
ITERATION # 7
Head of Queue: 71 9 13 5 :Tail of Queue
Removing Item: 71
ITERATION # 8
Head of Queue: 9 13 5 :Tail of Queue
Removing Item: 9
ITERATION # 9
Head of Queue: 13 5 :Tail of Queue
Adding Item: 46
ITERATION # 10
Head of Queue: 13 5 46 :Tail of Queue
Adding Item: 89
ITERATION # 11
Head of Queue: 13 5 46 89 :Tail of Queue
Adding Item: 63
ITERATION # 12
Head of Queue: 13 5 46 89 63 :Tail of Queue
Adding Item: 43
ITERATION # 13
Head of Queue: 13 5 46 89 63 43 :Tail of Queue
Adding Item: 84
ITERATION # 14
Head of Queue: 13 5 46 89 63 43 84 :Tail of Queue
Removing Item: 13
ITERATION # 15
Head of Queue: 5 46 89 63 43 84 :Tail of Queue
Adding Item: 51
ITERATION # 16
Head of Queue: 5 46 89 63 43 84 51 :Tail of Queue
Adding Item: 43
ITERATION # 17
Head of Queue: 5 46 89 63 43 84 51 43 :Tail of Queue
```

```
Removing Item: 5
ITERATION # 18
Head of Queue: 46 89 63 43 84 51 43 :Tail of Queue
Adding Item: 68
ITERATION # 19
Head of Queue: 46 89 63 43 84 51 43 68 :Tail of Queue
Adding Item: 8
ITERATION # 20
Head of Queue: 46 89 63 43 84 51 43 68 8 :Tail of Queue
Removing Item: 46
ITERATION # 21
Head of Queue: 89 63 43 84 51 43 68 8 :Tail of Queue
Removing Item: 89
ITERATION # 22
Head of Queue: 63 43 84 51 43 68 8 :Tail of Queue
Adding Item: 71
ITERATION # 23
Head of Queue: 63 43 84 51 43 68 8 71 :Tail of Queue
Adding Item: 19
ITERATION # 24
Head of Queue: 63 43 84 51 43 68 8 71 19 :Tail of Queue
Adding Item: 30
ITERATION # 25
Head of Queue: 63 43 84 51 43 68 8 71 19 30 :Tail of Queue
Adding Item: 3
ITERATION # 26
Head of Queue: 63 43 84 51 43 68 8 71 19 30 3 :Tail of Queue
Removing Item: 63
ITERATION # 27
Head of Queue: 43 84 51 43 68 8 71 19 30 3 :Tail of Queue
Adding Item: 97
ITERATION # 28
Head of Queue: 43 84 51 43 68 8 71 19 30 3 97 :Tail of Queue
Adding Item: 69
ITERATION # 29
Head of Queue: 43 84 51 43 68 8 71 19 30 3 97 69 :Tail of Queue
Removing Item: 43
ITERATION # 30
Head of Queue: 84 51 43 68 8 71 19 30 3 97 69 :Tail of Queue
Removing Item: 84
ITERATION # 31
Head of Queue: 51 43 68 8 71 19 30 3 97 69 :Tail of Queue
Adding Item: 4
ITERATION # 32
Head of Queue: 51 43 68 8 71 19 30 3 97 69 4 :Tail of Queue
Adding Item: 41
ITERATION # 33
Head of Queue: 51 43 68 8 71 19 30 3 97 69 4 41 :Tail of Queue
Removing Item: 51
ITERATION # 34
Head of Queue: 43 68 8 71 19 30 3 97 69 4 41 :Tail of Queue
Adding Item: 37
ITERATION # 35
Head of Queue: 43 68 8 71 19 30 3 97 69 4 41 37 :Tail of Queue
```

```
Adding Item: 61
ITERATION # 36
Head of Queue: 43 68 8 71 19 30 3 97 69 4 41 37 61 :Tail of Queue
Adding Item: 90
ITERATION # 37
Head of Queue: 43 68 8 71 19 30 3 97 69 4 41 37 61 90 :Tail of Queue
Removing Item: 43
ITERATION # 38
Head of Queue: 68 8 71 19 30 3 97 69 4 41 37 61 90 :Tail of Queue
Removing Item: 68
ITERATION # 39
Head of Queue: 8 71 19 30 3 97 69 4 41 37 61 90 :Tail of Queue
Adding Item: 16
ITERATION # 40
Head of Queue: 8 71 19 30 3 97 69 4 41 37 61 90 16 :Tail of Queue
Removing Item: 8
ITERATION # 41
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 :Tail of Queue
Adding Item: 21
ITERATION # 42
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 :Tail of Queue
Adding Item: 52
ITERATION # 43
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 52 :Tail of Queue
Adding Item: 89
ITERATION # 44
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 52 89 :Tail of Queue
Adding Item: 18
ITERATION # 45
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 52 89 18 :Tail of Queue
Adding Item: 43
ITERATION # 46
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 52 89 18 43 :Tail of Queue
Adding Item: 91
ITERATION # 47
Head of Queue: 71 19 30 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91
:Tail of Queue
Removing Item: 71
ITERATION # 48
Head of Queue: 19 30 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 :Tail of Queue
Adding Item: 12
ITERATION # 49
Head of Queue: 19 30 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12
:Tail of Queue
Removing Item: 19
ITERATION # 50
Head of Queue: 30 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 :Tail of Queue
Removing Item: 30
ITERATION # 51
Head of Queue: 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 :Tail of Queue
Adding Item: 65
ITERATION # 52
Head of Queue: 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 :Tail of Queue
Adding Item: 7
```

```
ITERATION # 53
Head of Queue: 3 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7
:Tail of Queue
Removing Item: 3
ITERATION # 54
Head of Queue: 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 :Tail of Queue
Adding Item: 39
ITERATION # 55
Head of Queue: 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39
:Tail of Queue
Adding Item: 1
ITERATION # 56
Head of Queue: 97 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1
:Tail of Queue
Removing Item: 97
ITERATION # 57
Head of Queue: 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1
:Tail of Queue
Adding Item: 36
ITERATION # 58
Head of Queue: 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36
:Tail of Queue
Adding Item: 34
ITERATION # 59
Head of Queue: 69 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34
:Tail of Queue
Removing Item: 69
ITERATION # 60
Head of Queue: 4 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34
:Tail of Queue
Removing Item: 4
ITERATION # 61
Head of Queue: 41 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34
:Tail of Queue
Removing Item: 41
ITERATION # 62
Head of Queue: 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34 :Tail of Queue
Adding Item: 72
ITERATION # 63
Head of Queue: 37 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72
:Tail of Queue
Removing Item: 37
ITERATION # 64
Head of Queue: 61 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72 :Tail of Queue
Removing Item: 61
ITERATION # 65
Head of Queue: 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72 :Tail of Queue
Adding Item: 85
ITERATION # 66
Head of Queue: 90 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72 85 :Tail of Queue
Removing Item: 90
ITERATION # 67
Head of Queue: 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72 85 :Tail of Queue
Adding Item: 53
```



```
ITERATION # 68
Head of Queue: 16 21 52 89 18 43 91 12 65 7 39 1 36 34 72 85 53 :Tail of Queue
Removing Item: 16
ITERATION # 69
Head of Queue: 21 52 89 18 43 91 12 65 7 39 1 36 34 72 85 53 :Tail of Queue
Removing Item: 21
ITERATION # 70
Head of Queue: 52 89 18 43 91 12 65 7 39 1 36 34 72 85 53 :Tail of Queue
Removing Item: 52
ITERATION # 71
Head of Queue: 89 18 43 91 12 65 7 39 1 36 34 72 85 53 :Tail of Queue
Adding Item: 80
ITERATION # 72
Head of Queue: 89 18 43 91 12 65 7 39 1 36 34 72 85 53 80 :Tail of Queue
Adding Item: 97
ITERATION # 73
Head of Queue: 89 18 43 91 12 65 7 39 1 36 34 72 85 53 80 97 :Tail of Queue
Removing Item: 89
ITERATION # 74
Head of Queue: 18 43 91 12 65 7 39 1 36 34 72 85 53 80 97 :Tail of Queue
Removing Item: 18
ITERATION # 75
Head of Queue: 43 91 12 65 7 39 1 36 34 72 85 53 80 97 :Tail of Queue
Removing Item: 43
ITERATION # 76
Head of Queue: 91 12 65 7 39 1 36 34 72 85 53 80 97 :Tail of Queue
Removing Item: 91
ITERATION # 77
Head of Queue: 12 65 7 39 1 36 34 72 85 53 80 97 :Tail of Queue
Adding Item: 36
ITERATION # 78
Head of Queue: 12 65 7 39 1 36 34 72 85 53 80 97 36 :Tail of Queue
Adding Item: 40
ITERATION # 79
Head of Queue: 12 65 7 39 1 36 34 72 85 53 80 97 36 40 :Tail of Queue
Adding Item: 1
ITERATION # 80
Head of Queue: 12 65 7 39 1 36 34 72 85 53 80 97 36 40 1 :Tail of Queue
Removing Item: 12
ITERATION # 81
Head of Queue: 65 7 39 1 36 34 72 85 53 80 97 36 40 1 :Tail of Queue
Adding Item: 28
ITERATION # 82
Head of Queue: 65 7 39 1 36 34 72 85 53 80 97 36 40 1 28 :Tail of Queue
Removing Item: 65
ITERATION # 83
Head of Queue: 7 39 1 36 34 72 85 53 80 97 36 40 1 28 :Tail of Queue
Removing Item: 7
ITERATION # 84
Head of Queue: 39 1 36 34 72 85 53 80 97 36 40 1 28 :Tail of Queue
Adding Item: 85
ITERATION # 85
Head of Queue: 39 1 36 34 72 85 53 80 97 36 40 1 28 85 :Tail of Queue
Adding Item: 22
```

```
ITERATION # 86
Head of Queue: 39 1 36 34 72 85 53 80 97 36 40 1 28 85 22 :Tail of Queue
Removing Item: 39
ITERATION # 87
Head of Queue: 1 36 34 72 85 53 80 97 36 40 1 28 85 22 :Tail of Queue
Removing Item: 1
ITERATION # 88
Head of Queue: 36 34 72 85 53 80 97 36 40 1 28 85 22 :Tail of Queue
Adding Item: 9
ITERATION # 89
Head of Queue: 36 34 72 85 53 80 97 36 40 1 28 85 22 9 :Tail of Queue
Adding Item: 77
ITERATION # 90
Head of Queue: 36 34 72 85 53 80 97 36 40 1 28 85 22 9 77 :Tail of Queue
Adding Item: 50
ITERATION # 91
Head of Queue: 36 34 72 85 53 80 97 36 40 1 28 85 22 9 77 50 :Tail of Queue
Adding Item: 59
ITERATION # 92
Head of Queue: 36 34 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 :Tail of Queue
Removing Item: 36
ITERATION # 93
Head of Queue: 34 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 :Tail of Queue
Removing Item: 34
ITERATION # 94
Head of Queue: 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 :Tail of Queue
Adding Item: 21
ITERATION # 95
Head of Queue: 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 21 :Tail of Queue
Adding Item: 73
ITERATION # 96
Head of Queue: 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 :Tail of Queue
Adding Item: 49
ITERATION # 97
Head of Queue: 72 85 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 49
:Tail of Queue
Removing Item: 72
ITERATION # 98
Head of Queue: 85 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 49 :Tail of Queue
Removing Item: 85
ITERATION # 99
Head of Queue: 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 49 :Tail of Queue
Adding Item: 64
ITERATION # 100
Head of Queue: 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 49 64 :Tail of Queue
Adding Item: 83
Head of Queue: 53 80 97 36 40 1 28 85 22 9 77 50 59 21 73 49 64 83
:Tail of Queue
NIL
84_
```

LispCourse #34: Variable Binding and the Interlisp Stack

Variable Reference Inside Functions: Bound and Free Variables

Consider the following function definition from the solution to Homework #30:

```
(DEFINEQ
  (LC.ParseNameString
    (LAMBDA (String)
      (SETQ String (CONCAT String))
      (SETQ Comma (STRPOS "," String))
      (SETQ Space (STRPOS " " String Comma))
      (create LC.Name
        Last _
          (MKATOM
            (SUBSTRING String 1 (SUB1 Comma)))
        First _
          (MKATOM
            (SUBSTRING String (ADD1 Comma) (SUB1 Space)))
        Middle _
          (MKATOM
            (SUBSTRING String (ADD1 Space))))))
```

In this function, there are three variables: *String*, *Comma*, and *Space*.

String differs from *Comma* and *Space* in that before the function is entered during a function call evaluation, the value of *String* is temporarily set (i.e., **bound**) to the value of the argument in the function call.

Moreover, the old value of *String* is reset to its previous value when the function is exited, despite the new value assigned to *String* by the SETQ in the function.

In contrast, *Comma* and *Space* have unknown values when the function is entered and the SETQ operations on *Comma* and *Space* permanently change the value of these variables.

String is known as a ***bound variable*** within the function `LC.ParseNameString`.

A bound variable is one whose value is set when the function is entered and reset to the previous value when the function is exited.

In particular, a bound variable has the following property: You can change the name of the variable, and the operation of the function will not change.

Substituting *NameString* for *String* throughout the definition of `LC.ParseNameString` would not change how the function worked.

Comma and *Space* are known as ***free variables*** within the function `LC.ParseNameString`.

A variable is a free variable in a function definition if it is not bound within that function definition.

The value of a free variable when the function is entered cannot be specified *a priori*. The free variable may have a value or it may not.

Moreover, when the function is exited, the free variable is not reset to its previous value.

In particular, a free variable has the following property: If you change the name of the variable, and the operation of the function may change, depending on the context of the evaluation.

For example, if the variables *ListSize* and *StringSize* were used by the Lisp Exec to hold important information, then changing *Comma* to *ListSize* and *Space* to *StringSize* in the definition of `LC.ParseNameString` might have serious side-effects on the Lisp Exec, since evaluating a call to `LC.ParseNameString` would change the value of these variables.

The problem with free variables in a function definition is that there is some ambiguity as to what is being referred to by the free variable. Interpreting the free variable reference depends on some context outside of the function itself.

Consider the following example:

```
1_ (DEFINEQ
    (LC.FindComma
```

```

(LAMBDA (String SearchLetter)
  (SETQ SearchLetter ",")
  (LC.FindLetter String)))
(LC.FindLetter
  (LAMBDA (String)
    (STRPOS SearchLetter String))))
(LC.FindComma LC.FindLetter)
2_ (SETQ SearchLetter "+")
"+ "
3_ (LC.FindLetter "AB,CD+EF")
6
4_ (LC.FindComma "AB,CD+EF")
????

```

Problem: What is the value of this function call?

There are two possible answers, depending on how the free variable *SearchLetter* is resolved in the call to *LC.FindLetter*.

1. If *SearchLetter* in *LC.FindLetter* refers to the global value in the Exec environment, then the result would be 6 since *LC.FindLetter* would be searching for a "+" as determined by event 2.
2. If *SearchLetter* in *LC.FindLetter* refers to the most recent binding of *SearchLetter* anywhere, then the result would be 3 since *LC.FindLetter* would be searching for a "," as determined by the binding and *SETQ* statements in *LC.FindComma*.

In fact, in Interlisp the value returned by *(LC.FindComma "AB,CD+EF")* in event 4 would be 3, because a free variable reference in a function always references the most recent binding of that variable.

The following several sections, explain in detail how Interlisp handles variable references, both free and bound, within function definitions.

The main point of these sections is that the model of variables we have been using until now is far too simple.

Until now, we have assumed that the value of a variable in a function definition is the value of the atom with the same name as the variable.

This is true "at the top level", i.e. for variable references typed directly to the Lisp Exec.

However, for variable references within a function, Interlisp uses a much more complex scheme to set and determine the value of an variable.

Review: Evaluating S-expressions using EVAL and APPLY

Recall that all of the significant work in Lisp is done by the Lisp evaluator during the EVAL part of the read-EVAL-print loop.

Recall also that the Lisp evaluator is built around two functions: *EVAL* and *APPLY*.

Hence, understanding Lisp function evaluation requires understanding the two functions EVAL and APPLY.

EVAL

The following defines a function that could be used by the Lisp evaluator to evaluate arbitrary S-expressions (i.e., EVAL):

```
(DEFINEQ
  (EVAL
    (LAMBDA (SExp)
      (COND
        ((NLISTP SExp) (LookUpValue SExp))
        (T
         (APPLY (CAR SExp)
                  (FOR Arg IN (CDR SExp)
                       COLLECT (EVAL Arg))))
```

Note: The function *LookUpValue* is some mysterious function that can look up the value of atoms, arrays, etc. in the Lisp environment.

In natural language, EVAL does the following:

If *SExp* is not a list, then *look up the value* of *SExp* and return it.

If *SExp* is a list, then *APPLY* the function named by the first element (i.e., CAR) of *SExp* to the list obtained by collecting the evaluation each item in the rest (i.e., CDR) of *SExp*.

Aside from *LookUpValue* (which will remain mysterious for a while), the central function used in defining EVAL is the function *APPLY*.

APPLY

APPLY could be defined as follows:

```
(DEFINEQ
  (APPLY
    (LAMBDA (Function Arguments)
      (FOR Parameter IN (CADR (GetFunctionDefn Function))
        AS Argument IN Arguments
        DO (Bind Parameter Argument))
      (FOR SExpr IN (CDDR (GetFunctionDefn Function))
        AS Ctr FROM 1 TO
          (DIFFERENCE
            (LENGTH (GetFunctionDefn
              Function)) 3)
            DO (EVAL SExpr))
      (SETQ Result
        (EVAL (CAR (LAST (GetFunctionDefn Function))))))
      (FOR Parameter IN (CADR (GetFunctionDefn Function))
        DO (Unbind Parameter))
      Result)))
```

Note: The functions *GetFunctionDefn*, *Bind*, and *Unbind* are some mysterious functions that can look up the definition of a function, bind a parameter to a value, and unbind a parameter, respectively.

In natural language, *APPLY* does the following:

For each parameter in the parameters list in the function definition of *Function*, *bind* that parameter to the corresponding argument in the *Arguments* list.

Then, for each S-expression in the body of the function definition except for the last, *EVAL* the S-expression.

Then, *EVAL* the last S-expression in the body of the function definition and hold on to the resulting value.

Then, for each parameter in the parameters list in the function definition of *Function*, ***unbind*** that parameter.

Finally, return the result already derived by EVALing the last S-expression.

EXAMPLE (from LispCourse #3, page 6)

```
(DEFINEQ
  (MOVEFILE
    (LAMBDA (FromFile ToFile)
      (COPYFILE FromFile ToFile)
      (DELFILE FromFile)
      (QUOTE AllDone))))
```

Evaluating (*MOVEFILE 'OLD.LISP 'NEW.LISP*) proceeds as follows:

EVAL:

1. 'OLD.LISP evaluates to OLD.LISP
2. 'NEW.LISP evaluates to NEW.LISP
3. APPLY the function MOVEFILE to (OLD.LISP NEW.LISP)

APPLY:

1. FromFile is bound to OLD.LISP
2. ToFile is bound to NEW.LISP
3. (COPYFILE FromFile ToFile) is evaluated using current bindings of FromFile and ToFile (i.e., FromFile will evaluate to OLD.LISP and ToFile will evaluate to NEW.LISP).
4. (DELFILE FromFile) is evaluated similarly.
5. (QUOTE AllDone) is evaluated to AllDone.
6. FromFile and ToFile are reset to their previous values (if any).
7. APPLY returns AllDone.

Missing Pieces

The foregoing explanation references several functions that have not yet been explained.

In particular, *LookUpValue*, *Bind*, *Unbind*, and *GetFunctionDefn* are critical functions used by EVAL and/or APPLY in the process of evaluating S-expressions.

The following sections explain the operation of these functions in the Interlisp-D evaluator.

The Lisp Stack, Variable Binding, Variable Lookup, and Related Topics

In the APPLY phase of evaluating a Lisp function call, the first step is to *bind* the parameters of the function to the corresponding arguments in the function call.

In LispCourse #3 (page 6), binding a parameter to an argument was described as "temporarily SETQing the parameter to the argument value". The implication was that binding simply temporarily resets the value assigned to the atom of the same name as the parameter.

In actuality, binding is much more complex and relies on a special data structure called the *stack*.

The Lisp Stack

The stack is a data structure that contains a variable number of *stack frames* arranged in an ordered one-dimensional vector.

Each stack frame contains a variable number of *binding records*.

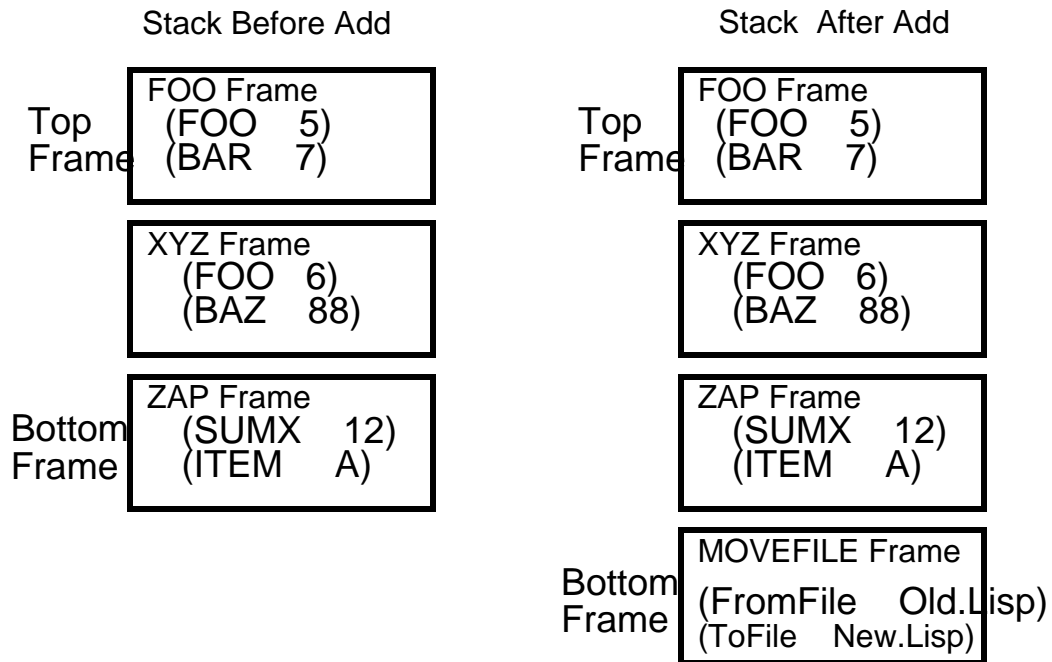
Each binding record is a pair consisting of a *variable name* (i.e., a litatom) and a *value*.

The stack has a *top* and a *bottom*.

Stack frames can only be added to the **bottom** of the stack.

Stack frames can only be removed from the **bottom** of the stack.

Adding a stack frame to the bottom of the stack



The stack implements a *first-in/last-out* access scheme.

If we add Frame X to the bottom of the stack and subsequently add N more frames to the bottom of the stack, then we have to remove the last N frames before we can remove Frame X.

The analogy is to a stack of boxes. To remove the Nth box down in the stack, you have to first remove the N-1 boxes sitting on top of that box.

(Contrast the stack with the queue data structure in Homework #32, which implemented a *first-in/first-out* access scheme.)

Variable Binding

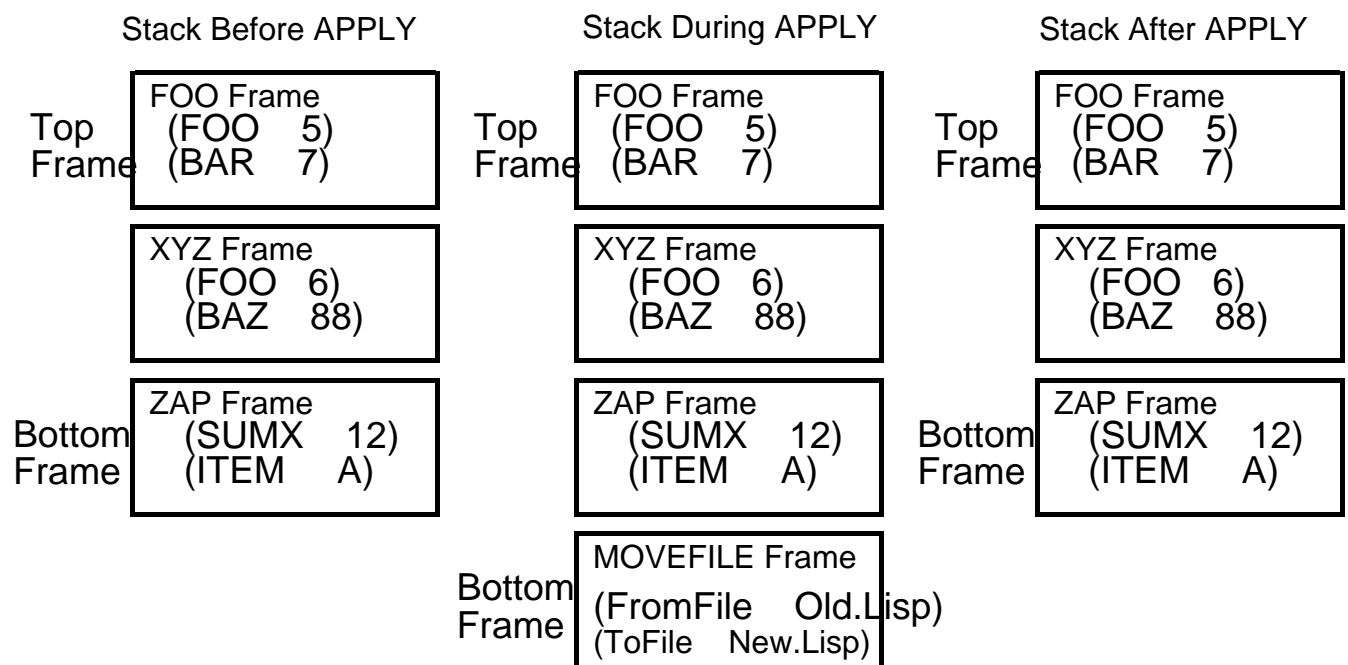
Every time the Lisp evaluator evaluates a new function call, APPLY creates a new stack frame and adds it to the bottom of the Interlisp stack.

When APPLY **binds** the parameters of the function to the arguments of the function call, it simply adds a binding record to this stack frame for each parameter.

When APPLY **unbinds** the parameters at the end, it simply removes this stack frame from the stack.

Example:

Evaluating the function call: (MOVEFILE 'Old.Lisp 'New.Lisp)



Looking up the Value of a Variable

After binding all of the parameters, APPLY calls EVAL on each S-expression in the function body. These S-expression often refer to variables.

Example: Inside the body of MOVEFILE is the S-expression (*COPYFILE FromFile ToFile*), which refers to 2 variables *FromFile* and *ToFile*.

EVALuating these S-expressions involves EVALuating these variables.

Looking at EVAL, evaluating a variable (i.e., a NLISTP) involves a function, *LookUpValue*, that looks up the value of the variable.

The *LookUpValue* function works as follows:

Starting at the bottom of the stack, search each stack frame for a binding record with a variable name EQ to the variable being looked up.

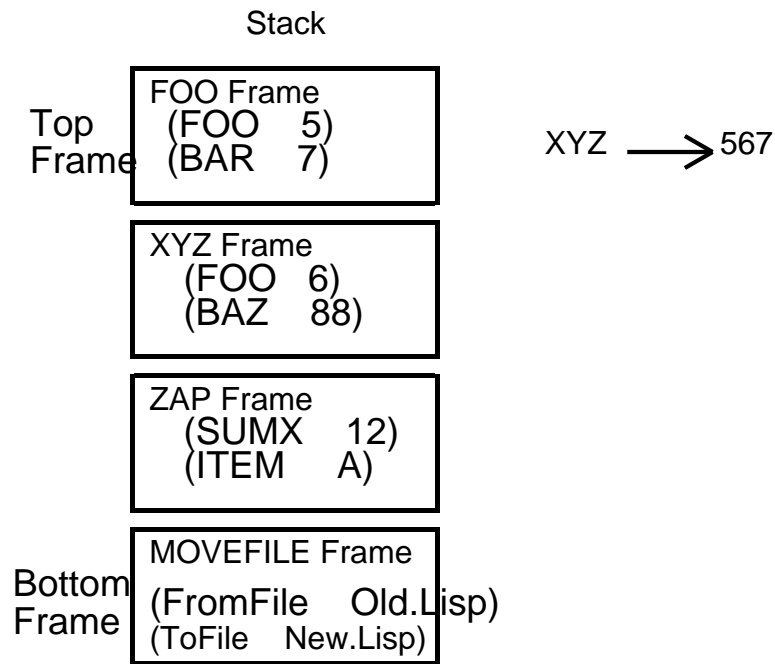
Return the value associated with the FIRST such binding record found on the stack.

If there is no binding record on the stack, then get the value attached to the atom with the same name as the variable being looked up.

If the atom has no value, then break with an "unbound atom" error.

Examples:

Given the following stack and atom values:



(LookUpValue 'FromFile) would return *Old.Lisp*

(LookUpValue 'FOO) would return *6*

(LookUpValue 'BAR) would return *7*

(LookUpValue 'XYZ) would return *567*

(LookUpValue 'PRQ) would return *u.b.a. error*

Setting the Value of a Variable

Within a function body being processed by APPLY, the SET functions work in a manner analogous to variable lookup.

In particular, **(SETQQ *Variable Value*)** will search up the stack starting at the bottom for a binding record for the variable *Variable*. If it finds one, it will change the value portion of this binding record to be *Value*.

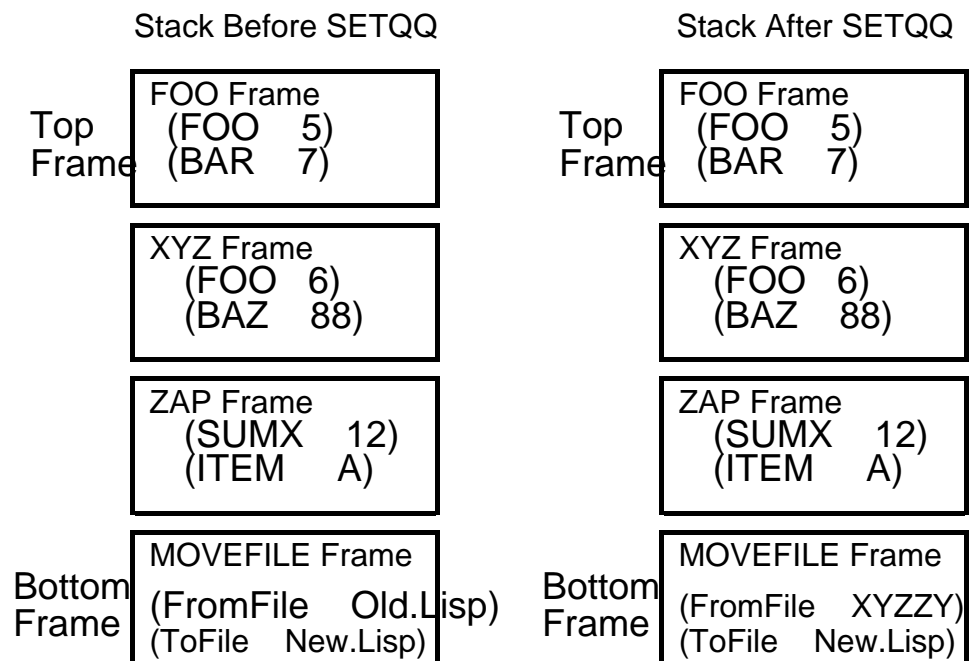
If no binding record is found on the stack, then SETQQ will set the value of the atom *Variable*.

SET and SETQ work analogously.

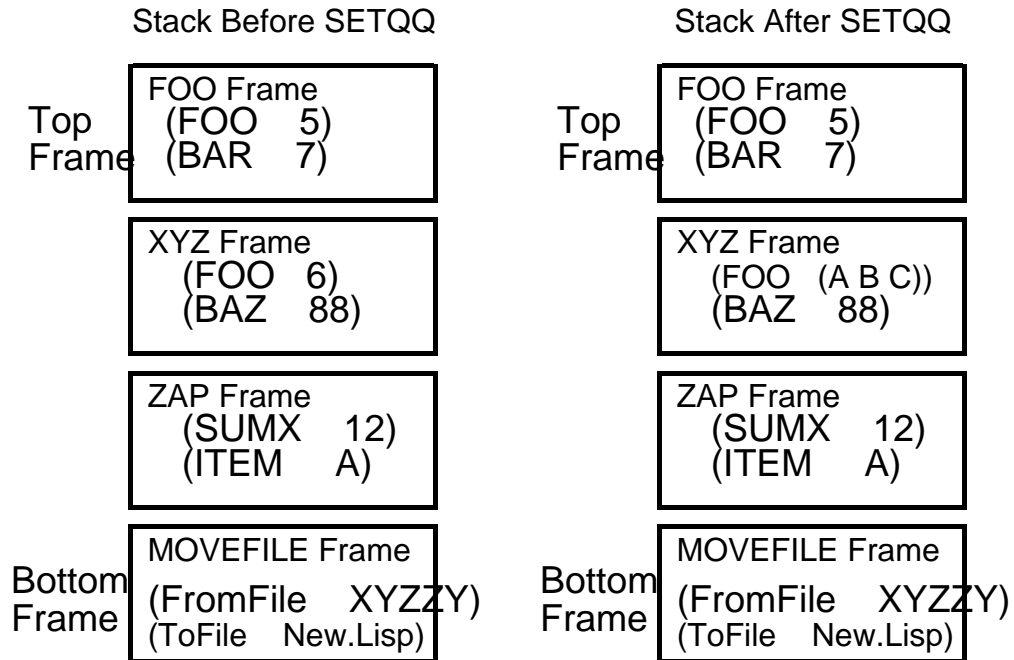
Examples of variable setting:

All of the following assume that the SET statement is part of the MOVEFILE function definition.

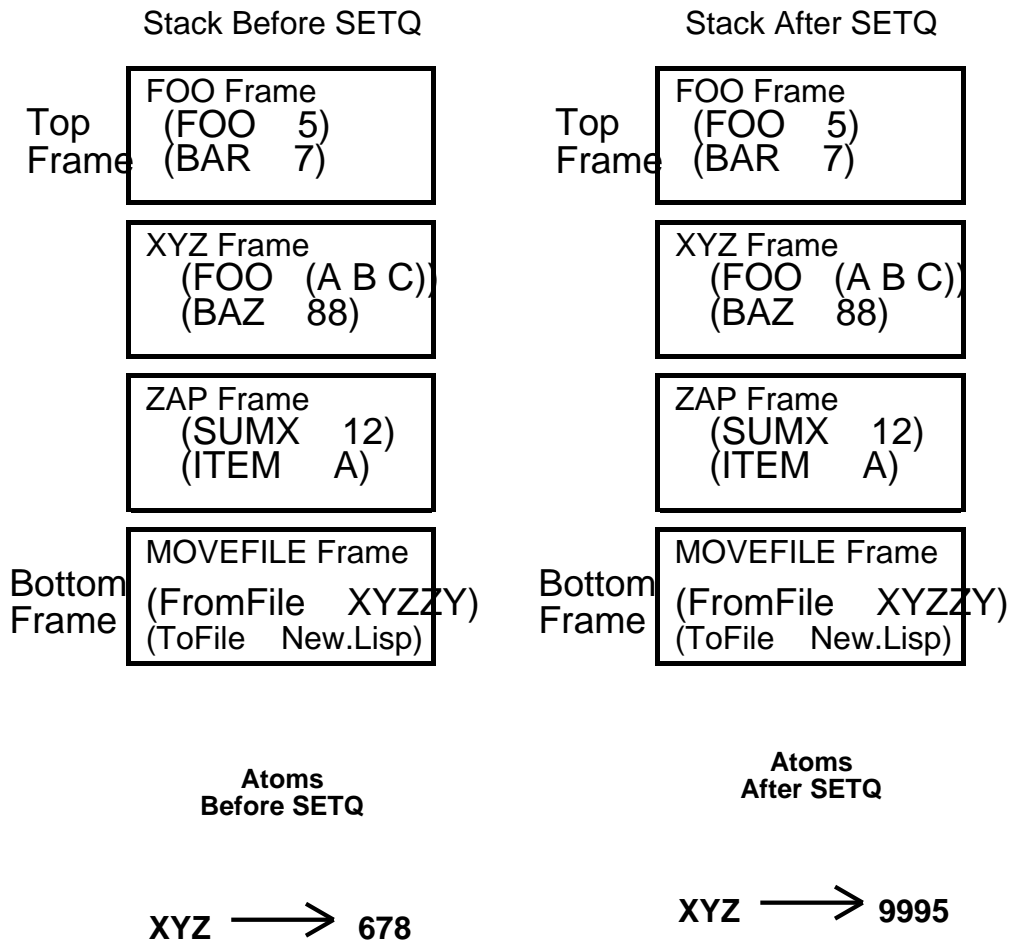
Evaluating: (SETQQ FromFile XYZZY)



Evaluating: (SETQQ FOO (A B C))



Evaluating: (SETQ XYZ 9995)



Function Definition Lookup

There is no such thing as binding a function name in Interlisp.

All function names in Interlisp refer to the function definition attached to the atom of the same name.

The function *GetFunctionDefn* in the definition of APPLY simply accesses the function definition for the named atom.

In particular, it does not search the stack for new bindings of the function name.

Note: This is different in other dialects of Lisp. Some Lisps allow binding of function names as well as variables names on the stack.

Variable Reference in Practice: LET and PROG are used to bind variables

Avoid Free Variables!

In practice the rule is: *Bound variables are good, free variables are bad.*

The effect of evaluating a function with bound variables is always predictable.

In contrast, evaluating a function that uses free variables can lead to differing results, depending on the context of the evaluation.

Examples:

Interference between variable names

```

1_ (DEFINEQ
      (Circumference (LAMBDA (Radius)
                      (TIMES 2 PI Radius))))
(Circumference)
2_ (SETQ PI 3.1416)
3.1416
3_ (DEFINEQ
      (MakeProgrammersInterface (LAMBDA NIL
                                  (SETQ PI (CREATE ProgInt ....)))))
(MakeProgrammersInterface)
4_ (MakeProgrammersInterface)
{ProgInt}#32,33412
5_ (Circumference 5)
NON-NUMBERIC ARG
{ProgInt}#32,33412

```

Inadvertantly altering system parameters

```

6_ (DEFINEQ
      (FontNameToFontNumber (LAMBDA (FontName)
        (SETQ DEFAULTFONT 1)
        (FOR Name in '(Helvetica TimesRoman
          Gacha Modern)
          AS Number FROM 1
          WHEN (EQ FontName Name)
          DO (SETQ DEFAULTFONT
            Number))
        DEFAULTFONT)

```

(FontNameToFontNumber)

```
7_ (FontNameToFontNumber 'Gacha)
```

```
3
```

```
8_ (TEDIT "XXXXXX")
```

ILLEGAL ARG

{FONTCLASS}#71,10500

(because DEFAULTFONT has been reset to an illegal value)

The Funarg Problem

```

9_ (DEFINEQ
      (Sum (LAMBDA (List Transform)
        (FOR N IN List SUM (APPLY* Transform
          N))))

```

(Sum)

```

10_ (DEFINEQ
      (SumSquares#1 (LAMBDA (List)
        (Sum List (FUNCTION SQUARE))))
      (SumSquares#2 (LAMBDA (List)
        (SETQ N 2)
        (Sum List (FUNCTION NthPower))))

```

```

(SumCubes (LAMBDA (List)
  (SETQ N 3)
  (Sum List (FUNCTION NthPower))))
(NthPower (LAMBDA (X)
  (EXPT X N))))
(SumSquares#1 SumSquares#2 SumCubes NthPower)
11_ (SumSquares#1 (LIST 1 2 3 4 5))
55
12_ (SumSquares#2 (LIST 1 2 3 4 5))
3413
13_ (SumCubes (LIST 1 2 3 4 5))
3413

```

The examples illustrate the problems involved in using free variables in defining functions. Because of these problems, it is best to avoid using free variables.

The LET Special Form

Consider the following function that computes the average of all the numbers in a list of numbers and literals:

```

(DEFINEQ
  (AverageOfNumbers (List)
    (SETQ Sum 0.0)
    (SETQ N 0)
    (FOR Item IN List
      WHEN (NUMBERP Item)
      DO
        (SETQ Sum (PLUS Sum Item))
        (SETQ N (ADD1 N)))
    (COND
      ((NOT (ZEROP N))
       (QUOTIENT Sum N))

```

(T NIL))))))

It would not be possible to write this function without the **Sum** and **N** variables, since they are used to store intermediate results as the function iterates through the list.

There is no reason, however, for **Sum** and **N** to be free variables. Their function should be limited to the scope of the `AverageOfNumbers` function body.

On the other hand, **Sum** and **N** are not a parameters either and therefore should not be made into a bound variables by placing them in the parameter list of the function.

The **LET** special form provides the means to bind variables without making them part of the parameter list.

LET has the format:

(LET *BindingList S-Expression1 S-Expression2 ...*)

BindingList is a list of variable-value pairs, i.e., (*VariableName InitialValue*). A variable can also be expressed by just its *VariableName*, which is equivalent to the list (*VariableName NIL*).

The *S-Expression_i* are arbitrary Lisp S-expressions to be evaluated.

LET works as follows:

The variables specified in the binding list are bound on the stack and set to the specified initial values.

The S-expressions are evaluated in order.

The LET form returns the value of the last S-expression, unbinding the variables in the binding list before it exits.

Variables in the binding list are bound "in parallel" and thus the order of mention in the binding list is unimportant.

If the binding list is `((X 55) (Y X))`, the value of `X` within the `LET` will be 55 and the value of `Y` will be whatever was the value of `X` before the `LET` (and **not** 55 as might be expected).

The same effect could have been achieved by the binding list `((Y X) (X 55))`.

The proper definition for `AverageOfList` would thus be:

```
(DEFINEQ
  (AverageOfList (List)
    (LET ((Sum 0.0)(N 0))
      (FOR Item IN List
        WHEN (NUMBERP Item)
        DO
          (SETQ Sum (PLUS Sum
                          Item))
          (SETQ N (ADD1 N)))
      (COND
        ((NOT (ZEROP N))
         (QUOTIENT Sum N))
        (T NIL))))))
```

In this definition, `List`, `Sum`, and `N` are all bound variables.

When the function is entered, `List` is bound since it is in the parameter list.

When the `LET` is entered, `Sum` and `N` are bound on the stack and set to their initial values of zero.

When the `LET` is exited, `Sum` and `N` are unbound.

When the function is exited, `List` is unbound.

As a second example, consider a function (slightly modified) from the solution to Homework#32:

```
(DEFINEQ
  (LC.PrintQueue
    (LAMBDA (Q)
      (COND
        ((NOT (LC.QueueEmptyP Q))
          (SETQ NextPtr (fetch (LC.Queue Head)
of Q))
          (PRINT (CAR NextPtr))
          (until (EQ
                  (SETQ NextPtr (CDR
NextPtr))
                  (fetch (LC.Queue Tail) of
Q)
                )
                DO (PRIN1 (CAR NextPtr)))
                (TERPRI))))))
```

Note the unnecessary use of **NextPtr** as free variable.

The proper definition for LC.PrintQueue is:

```
(DEFINEQ
  (LC.PrintQueue
    (LAMBDA (Q)
      (LET (NextPtr)
        (COND
          ((NOT (LC.QueueEmptyP Q))
            (SETQ NextPtr (fetch (LC.Queue Head)
of Q))
            (PRINT (CAR NextPtr))
            (until (EQ
                    (SETQ NextPtr (CDR
NextPtr))
                    (fetch (LC.Queue Tail) of
Q)
                  )
                  DO (PRIN1 (CAR NextPtr)))
                  (TERPRI))))))
```

```

                                (fetch (LC.Queue Tail) of
                                Q)
                                DO (PRIN1 (CAR NextPtr)))
                                (TERPRI))))

```

In this definition, **NextPtr** is bound within the context of the LET statement (and hence within the entire function body).

The PROG Special Form

The *PROG* special form is very much like the LET special form, but it allows a little more flexibility in how and when the special form is exited.

PROG has the format:

(PROG *BindingList S-Expression1 S-Expression2 ...*)

BindingList is as in the LET special form.

The *S-Expression_i* are arbitrary Lisp S-expressions to be evaluated. One or more of these S-expressions may contain a sub-expression of the form **(RETURN *S-expression*)**

PROG works as follows:

The variables specified in the binding list are bound on the stack and set to the specified initial values.

The S-expressions are evaluated in order until an expression (or sub-expression) of the form **(RETURN *S-expression*)** is evaluated.

Evaluating this RETURN expression evaluates the embedded *S-expression*, then causes the PROG special form to be exited, returning the value of this evaluation. On exit, the variables in the binding list are unbound.

If no RETURN statement is encountered while evaluating the S-expressions, PROG unbinds the variables in the binding list and returns NIL.

As in LET, variables in the PROG binding list are bound "in parallel" and thus the order of mention in the binding list is unimportant.

PROG should be used where one might want to exit at one of several places in a function, depending on certain conditions.

For example, the following is a function that transfers a list of numbers to an array and then places the sum of the numbers in the last cell of the array. The program immediately exits with NIL if the Array is not on larger than the length of the list.

```
(DEFINEQ
  (TransferListToArray (LAMBDA (List Array)
    (PROG
      ((LstLen (LENGTH List))
        (ArrSize (ARRAYSIZE Array))
        (Sum 0.0))
      (COND
        ((OR
          (ZEROP LstLen))
          (NEQ LstLen (SUB1
            ArrSize))))
          (RETURN NIL))
      (FOR Item IN List
        AS Index FROM 1
        DO
          (ELT Array Index
            Item)
          (SETQ Sum (PLUS
            Sum Item)))
      (ELT Array ArrSize Sum)
      (RETURN Sum)))
```

FOR and WHILE are implemented using PROG

The FOR and WHILE clisp forms are implemented using the PROG special form.

An important side-effect of this is that the (RETURN S-expr) form can be used to exit from FOR or WHILE loops before their normal termination.

For example, the following will sum the item in a list until a negative number is reached.

```
(DEFINEQ
  (SumTillMinus (LAMBDA (List)
    (PROG ((Sum 0.0)
      (FOR Item IN List
        DO (COND
          ((MINUSP
            Item)(RETURN
              NIL)))
          (SETQ Sum (PLUS Sum
              Item)))
        (RETURN Sum))))))
```

In this example, the RETURN inside the FOR loop will exit only out of the PROG implicit in the FOR. It will not exit out of the PROG that contains the FOR loop.

This is true for all RETURN statements: each RETURN exits out only the lowest level enclosing PROG and not out of any PROGs that in turn enclose the lowest level PROG.

Note also that the previous example, would probably best be written using a LET as follows:

```
(DEFINEQ
  (SumTillMinus (LAMBDA (List)
```

```

(LET ((Sum 0.0))
  (FOR Item IN List
    DO (COND
        ((MINUSP
          Item)(RETURN
            NIL)))
      (SETQ Sum (PLUS Sum
        Item))))
  Sum))))

```

As another example, the following function takes a list of numbers and returns a list of as many of the initial numbers as necessary to sum to just of 100. If any of the items in the initial list is non-numeric the function exits and returns the bad item.

```

(DEFINEQ
  (FirstHundred (LAMBDA (List)
    (LET ((Sum 0.0))
      (WHILE (LESSP Sum 100)
        FOR Item IN List
        COLLECT
          (COND
            ((NOT (NUMBERP
              Item))
              (RETURN Item)))
            (SETQ Sum (PLUS Sum
              Item))
            Item))))))

```

Note: The RETURN statement inside of a COLLECT loop will cause the COLLECT loop to return with the value of the S-expr in the RETURN clause instead of the list being COLLECTed.

```
3_ (FirstHundred (LIST 1 2 3 44 55 66 77 88))
```

```
(1 2 3 44 55)
```

```
4_ (FirstHundred (LIST 1 2 'A 44 55 66 77))
```

A

Global Variables: Free variables used as system or package parameters

Free variables are sometimes necessary.

In particular, system or package parameters (e.g., `DEFAULTPRINTINGHOST`, `CHAT.FONT`, `DEFAULTFONT`, `LAFITEDEFAULTHPOST&DIR`, etc.) are set outside of any function (e.g., in an Init file or in the Lisp Exec) but must be used (and sometimes set) within various system or user functions.

These system/package parameters must be globally accessible, i.e., available to all functions in all contexts.

Therefore, they cannot be bound on the stack inside of some particular function.

When setting or retrieving the value of one of these global parameters, you want to go directly to the value attached to the atom, skipping the search up the stack for other bindings.

For example, when a function definition includes the form *(SEND.FILE.TO.PRINTER '{DSK}FOO (CAR DEFAULTPRINTINGHOST))*, the *DEFAULTPRINTINGHOST* variable should reference the global value of this variable. If one of the calling functions of this function has stupidly rebound *DEFAULTPRINTINGHOST*, the rebinding should probably be ignored.

The functions *GETTOPVAL* and *SETTOPVAL* can be used to directly access the global value of a variable (i.e., the value attached to the atom) skipping the search for rebindings on the stack.

GETTOPVAL takes a single LITATOM as an argument and returns the value attached to that LITATOM.

SETTOPVAL takes a LITATOM and a value and sets the value of the LITATOM to be value, returning the value.

Example:

```
1_ (SETQ LineLength 107)
```

```

107
2_ (DEFINEQ
    (Tester
      (LAMBDA NIL
        (PROG ((LineLength 223))
          (PRINT
            (CONCAT "Initial values: " LineLength " "
              (GETTOPVAL 'LineLength)))
          (SETQ LineLength 55)
          (SETTOPVAL 'LineLength 88)
          (PRINT
            (CONCAT "LineLength: " LineLength "
              TopVal of LineLength: " (GETTOPVAL
                'LineLength)))))))
    )
)

```

(Tester)

```

3_ (Tester)
Initial values: 223 107
LineLength: 55 TopVal of LineLength: 88
NIL

```

References

In the IRM:

PROG is on Page 4.3. LET is not in the IRM, but is essentially the same as PROG, except for the RETURN and GO features.

GETTOPVAL and SETTOPVAL are on page 2.5

Free variables and binding are covered in both Winston & Horn and Touretzky. But be careful. CommonLisp uses a different sort of binding scheme than does Interlisp, so not everything said in these books applies to Interlisp.

Look at Chapter 5 of Touretzky and page 53 of Winston and Horn.

Also LET is explained starting on page 57 of W&H and on page 250 of T.

Exercise

Overall Task: Write a simple Lisp evaluator.

For each of the functions you write, put a print statement at the end of the function that prints on the screen some information about what the function just did. This way, you can watch your evaluator in action when it all gets put together.

1. Write a function to do variable binding on a stack.

The stack should just be a list: use CONS and CDR to add and remove items.

The binding function should take two equal length lists, one of variables and one of values.

It should put a marker on the stack (e.g., the list atom MARKER) and then put a binding pair on the stack for each variable and its corresponding value in the lists.

2. Write a function that unbinds variables. It should basically remove items from the stack up to and including the next marker.

3. Write a function that looks up the value of a variable on the binding stack. If it's not on the stack, get its top level value using GETTOPVAL.

4. Write a function that sets the value of a variable. If the variable is on the binding stack, then just reset the value in that binding. Otherwise, set its top level value using SETTOPVAL.

5. Rewrite the EVAL and APPLY procedures from the course notes.

- A. Rename the functions so as not to mess up the original EVAL and APPLY in Interlisp.
- B. Use your bind, unbind and lookup functions.
- C. In your apply, if the function name is SET, SETQ, or SETQQ, then use your set value of variable function to carry out the appropriate setting action instead of looking the definition of SET, SETQ, or SETQQ.
- D. In your apply use GETD to get a function definition. If the value of GETD is a list, then proceed as in the course notes. If the value of GETD is not a

list, then assume it is a primitive function and just invoke the standard `APPLY` function instead of the rest of your apply function.

6. Write a procedure (called `CountAtoms`) that recursively counts all the atoms in a list. (See page 12 of LispCourse #5).
7. Use your `eval` to evaluate `(CountAtoms '(A (B C D) (E (F (G (H) I) J K) L)))` and watch the evaluator in action as it prints out its action summaries.

LispCourse #35: Solutions to Homework #34

Apologies

Homework #34 was poorly planned out. In doing the homework myself, I ran into a number of conceptual rough spots that I did not point out in the problem specifications.

Some of these problems are:

- 1) I forgot to account for numbers (and arrays, etc.) that evaluate to themselves. In LC.Eval, LITATOMs should have their value looked up on the stack, LISTPs should be evaluated as per the rules, and everything else (including numbers) should evaluate to themselves. Its this last clause I left out of my description.
- 2) I did not account for NLAMBDA functions. In the LC.Eval function, NLAMBDA functions have to be handled specially, since their arguments should not be evaluated before LC.Apply is called. On page 8 of the solution printout is a magic function LC.NlambdaP that determines if its argument is the name of an NLAMBDA function.
- 3) As part of the LC.Apply function, I specified that if the function definition was not a list, then you should use the standard Interlisp APPLY procedure instead of your own. This is true, except it doesn't quite work for functions that use free variables.

What happens is this: For functions with LISTP definitions, you bind variables on your own stack. For non-LISTP definitions, you use Interlisp which has its own stack. When Interlisp wants to look up a free variable, it looks on its own stack. But the variable might have last been bound on your stack. Therefore, Interlisp will either not find the binding or find an old binding.

Solution: Before you call the Interlisp APPLY, you have to bind on the interlisp stack all of the variables that are bound on your stack. One way to do this is to construct a LET or PROG statement with the necessary bindings and containing the appropriate APPLY statement. You can then Interlisp EVAL this LET/PROG statement. This solution is captured in the function LC.LispApply on page 8 of the solution printout. In my LC.Apply, I called this function rather than calling APPLY directly to get around this problem.

- 4) Things get dull after you call `LC.LispApply`, since thereafter Lisp is doing all the EVALs and APPLYs. Therefore, you want to call standard Interlisp function as late as possible so that you can watch functions with LISTP definitions (i.e., one you have defined) being evaluated.

`LC.CountAtoms` has a `COND` at its top level. Therefore, most of the work in evaluating `LC.CountAtoms` ends up being done by Lisp and not by your `LC.Eval` and `LC.Apply`. I have one function, `LC.CountAtoms1`, that works this way. But I also wrote a second function, `LC.CountAtoms2`, that uses my own version of `COND` called `LC.Cond`. When I `LC.Eval` this second `CountAtoms` much more of the work is done by my evaluator because I bomb into Lisp much later in the evaluation process.

Solutions

Attached.

Sample Runs

Attached.

LispCourse #36: More on Variable Binding; Control Structures in Lisp

NLAMBDA Functions

Recall that NLAMBDA functions are treated specially by the Lisp interpreter.

In particular, when evaluating a function call containing an NLAMBDA function, the interpreter does *not* evaluate the arguments before the function is applied.

To define an NLAMBDA function, just use the keyword NLAMBDA instead of LAMBDA in the function definition.

For example:

```
1_ (DEFINEQ
      (E (NLAMBDA (FileName)
                (* * Call TEdit on the named file)
                (TEDIT FileName))))
(E)
2_ (E <LISPCOURSE>OUTLINE01.TED)
{PROCESS}#32,12345 (Starts up a TEdit)
3_ (SETQQ File <LISPCOURSE>OUTLINE01.TED)
<LISPCOURSE>OUTLINE01.TED
4_ (E File)
{DSK}File not found
```

Contrast the preceding NLAMBDA with an analogous LAMBDA function:

```
5_ (DEFINEQ
      (EX (LAMBDA (FileName)
            (* * Call TEdit on the named file)
            (TEDIT FileName))))
(EX)
6_ (EX ' <LISPCOURSE>OUTLINE01.TED)
{PROCESS}#32,12337 (Starts up a TEdit)
7_ (SETQQ File <LISPCOURSE>OUTLINE01.TED)
<LISPCOURSE>OUTLINE01.TED
```

8_ (EX File)

{PROCESS}#32,12678 (Starts up a TEdit)

The NLAMBDA version of this function has the *advantage* that user/programmer does not have to QUOTE the file name argument when calling the function since the argument is not evaluated.

I.e., compare events 2 and 6 in the preceding examples.

However, the NLAMBDA version has the *disadvantage* that it is less easily integrated into the workings of the Lisp Exec.

For example, you cannot set a variable to the name of the desired file and then use this variable to refer to the file when calling the NLAMBDA function. This is shown in the comparison between events 4 and 8 above.

The bottom line is that NLAMBDA function have their uses, especially when writing functions that interface with the user, but they also have their drawbacks in terms of flexibility.

For example: LISTFILES is an NLAMBDA function.

It is nice to be able to type: (LISTFILES {DSK}HOMEWORK34)
without worrying about the QUOTE.

On the other hand, you can do: (*FOR File in ListOfFiles DO (LISTFILES File)*) to print out each file in a list of files.

You would have to type (*FOR File in ListOfFiles DO (APPLY 'LISTFILES (LIST File))*), which is more than a bit clumsy.

The other place that NLAMBDA function are very handy is when you want to write functions that evaluate a bunch of SExpressions in a non-standard order.

For example, the following function evaluates the first of two SExpressions only if the first evaluates to non-NIL:

```
(DEFINEQ
  (FirstOnlyIfSecond
    (NLAMBDA (SEExpr1 SEExpr2)
```

```
(COND ((EVAL SExpr2) (EVAL SExpr1))))))
```

Note that COND, AND, OR, PROG, etc. are all standard Interlisp functions that use this trick to change the order in which a set of SExpressions are evaluated.

Spread and NoSpread Functions

The *Spread/NoSpread* distinction is a second classification of functions in Interlisp that is orthogonal to the LAMBDA/NLAMBDA distinction.

So far, we have been looking only at *spread* functions.

A *spread* function is one whose parameter list is in fact a LISTP.

Thus, a spread function definition has the form (**LAMBDA List SExpr1 ... SExprN**) or the form (**NLAMBDA List SExpr1 ... SExprN**).

When APPLYing spread functions, the Interlisp interpreter matches each parameter in this parameter list with the corresponding argument in the function call.

Note that the effect of this scheme is that each spread function has a fixed number of parameters. If there are more arguments than parameters, the arguments are just ignored.

In contrast, a *nospread* function has a parameter "list" consisting of a single LITATOM.

Thus, a nospread function definition has the form (**LAMBDA Litatom SExpr1 ... SExprN**) or the form (**NLAMBDA Litatom SExpr1 ... SExprN**).

When APPLYing a nospread function, Interlisp sets up this LITATOM to point to the list of all arguments in the function call. Using the LITATOM as a reference (as described below), you can access this list from within the function.

This scheme allows a single parameter to refer to an arbitrarily long list of arguments. Thus, nospread functions have no fixed number of parameters.

The manner in which one can reference the arguments of a nospread function differs for LAMBDA and NLAMBDA functions.

NLAMBDA nospread functions

Interlisp simply sets the LITATOM that is the parameter "list" to the list of arguments (unevaluated). You can then get to any element of the this list using CAR, CDR, LAST, etc.

For example, the following function calls TEdit for each of an arbitrary number of files:

```

10_(DEFINEQ
      (TEDs
        (NLAMBDA Files
          (FOR File in Files DO (TEDIT File))))))
(TEDs)
11_(TEDs {DSK}FOO)
NIL (Starts TEdit on a single file {DSK}FOO)
12_(TEDs {DSK}BAR {ERIS}<HALASZ>BAZ {DSK}ARG)
NIL (Starts 3 TEdits on three files as specified)
13_(TEDs A B C D E F G H J K)
NIL (Starts 10 TEdits on ten files as specified)

```

LAMBDA nospread functions

LAMBDA nospread functions are a bit more complicated.

The LITATOM that is the parameter "list" is bound to the number of arguments being passed in the current function call.

The arguments can be accessed individually using the functions **ARG** and **SETARG**.

(ARG *Litatom M*) ž In a LAMBDA nospread function whose parameter specification is the LITATOM *Litatom*, **ARG** returns the *M*th argument of the current function call. ARG is itself an NLAMBDA function that doesn't evaluate its argument but DOES evaluate its second argument.

(SETARG *Litatom M Value*) ž In a LAMBDA nospread function whose parameter specification is the LITATOM *Litatom*, **SETARG** sets the *M*th argument of the current function call to *Value*. SETARG is itself an NLAMBDA function that doesn't evaluate its argument but DOES evaluate its second and third arguments.

For example, the following (nonsense) function collects the results of evaluating an arbitrary number of SExprs, where each result is made into a string:

```
14_(DEFINEQ
  (ResultsAsStrings
    (LAMBDA SExprs
      (FOR Index FROM 1 to SExprs
        COLLECT
          (MKSTRING (ARG SExprs
                    Index))))))
```

(ResultsAsStrings)

```
15_ (ResultsAsStrings (PLUS 1 2) 'ABC (REVERSE '(A B C)))
```

```

("3" "ABC" "(C B A)")
16_ (ResultsAsStrings (PLUS 2 3) '(PLUS 2 3) (EVAL (PLUS 2
3)))
("5" "(PLUS 2 3)" "5")

```

What are nospread functions good for?

Nospread functions are handy whenever you want to write a function that handles an arbitrary number of arguments.

COND, PROG, AND, OR, and PLUS are all implemented as nospread functions since they all handle an arbitrary number of arguments.

Another example might be the print function, LC.Print, from Homework #35.

LC.Print was defined as a spread function as follows:

```

(DEFINEQ (LC.Print
  (LAMBDA (#Items Thing1 Thing2 Thing3 Thing4
    Thing5)
    (FOR Thing
      IN (LIST Thing1 Thing2 Thing3 Thing4
        Thing5)
      AS Ctr FROM 1 TO #Items
      DO (PRIN1 Thing))
    (TERPRI))))

```

This version of LC.Print could handle at most 5 things to be printed and required a sixth argument specifying how many actual arguments there were if there were less than 5.

But LC.Print should have been defined as a nospread function as follows:

```

(DEFINEQ
  (LC.Print (LAMBDA Things
    (FOR Index FROM 1 TO Things

```



```
DO (PRIN1 (ARG Things Index)))
(TERPRI))))
```

Note that this nospread version of LC.Print can print an arbitrary number of things and does not require an explicit parameter specifying how many arguments there are.

Summary of 4 function types

Since Spread/Nospread and LAMBDA/NLAMBDA are orthogonal distinctions, there are a total of 4 different kinds of functions in Interlisp:

```
LAMBDA spread
LAMBDA nospread
NLAMBDA spread
NLAMBDA nospread
```

You can find out the type of an arbitrary function in Interlisp using the `?=` facility in the Lisp Exec.

In the Lisp Exec, just type a `"(`, the *function name*, a *space*, the characters `"?="` and a *carriage return*.

Once the return is typed, the `?=` will be erased and on the next line there will be a print out the parameter "list" of the named function.

Following the parameter list will be a indication of the type of the function as follows:

```
LAMBDA spread -- Blank
LAMBDA nospread -- {L*}
NLAMBDA spread -- {NL}
NLAMBDA nospread -- {NL*}
```

Examples:

```
LAMBDA spread
```

```
EXEC Window
31+
NIL
31+(DIFFERENCE
(DIFFERENCE X Y)
```

LAMBDA nospread

```
EXEC Window
33+
NIL
33+(LC.Print
(LC.Print Things...) {L*}
```

NLAMBDA spread

```
EXEC Window
32+
((NIL))
(DDD)
32+(SETQQ
(SETQQ X Y) {NL}
```

NLAMBDA nospread

```
EXEC Window
32+
((NIL))
(DDD)
32+(LISTFILES
(LISTFILES FILES...) {NL*}
```

Functions as Arguments in Function Calls

In Lisp, using a function as an argument in a function call is no big deal. Functional arguments follow the same rules as any other argument.

Example:

```
1_(DEFINEQ
  (DoItToFiveAndFour
    (LAMBDA (ItFn)
      (APPLY ItFn (LIST 5 4))))))
```

```

(DoItToFiveAndFour)
2_(DoItToFiveAndFour 'PLUS)
9
3_(DoItToFiveAndFour 'DIFFERENCE)
1
4_(DoItToFiveAndFour (MKATOM (CONCAT "TI" 'MES)))
20
5_(DoItToFiveAndFour (LIST 'PLUS))
UNDEFINED FUNCTION
(PLUS)

```

Note: you can generally use a function definition (i.e., a list beginning with a LAMBDA) where a function name is required. This is because most Lisp functions that require a function name, will accept a function definition instead.

Example:

```

6_(DoItToFiveAndFour '(LAMBDA (X Y) (SUB1 (PLUS X
Y))))
8
7_(DoItToFiveAndFour '(LAMBDA (X Y) (TIMES (PLUS X Y)
Y)))
36 [= (5+4)*4]

```

When passing functions as arguments, it is important to pass down function with the appropriate characteristics, e.g., the required number of parameters.

Example:

```

8_(DoItToFiveAndFour '(LAMBDA (X Y Z) (PLUS X Z Y)))
NON-NUMERIC ARG
NIL
[Because Z is bound to NIL when (PLUS X Y Z) is evaluated.]

```

FUNCTION ž in the preceding examples, I used QUOTE to prevent evaluation of function names or function definitions used as arguments in a function call.

I actually should have used the function `FUNCTION` in place of `QUOTE` in these cases.

`FUNCTION` behaves much like `QUOTE`, but tells Lisp that it is dealing with a function rather than a data object. In some cases (e.g., in the compiler) this will allow Lisp to behave much more efficiently.

Example:

```
9_ (DoItToFiveAndFour (FUNCTION (LAMBDA (X Y) (SUB1 (PLUS
X Y))))))
8
10_ (DoItToFiveAndFour (FUNCTION PLUS))
9
```

A more realistic example

```
1_ (DEFIN EQ
      (WindowOp (LAMBDA (Op Window)
                 (OR (WINDOWP Window) (SETQ Window (GetWindow)))
                 (APPLY* Op Window)))
      (WindowOp)
2_ (WindowOp (FUNCTION CLOSEW) W)
   CLOSED [Closes the window that is the value of W]
3_ (WindowOp (FUNCTION MOVEW))
   (100 . 100) [Gets a window from the user, then moves that window]
```

Control Structures in Lisp

Procedural Abstraction

If there's one important concept in programming, it's *abstraction*.

Abstraction starts with the process of breaking down a large task into a set of sub-tasks, breaking down each sub-task into a set of sub-sub-tasks, and so on until you get to atomic tasks that can't be further decomposed.

You then write the primitive functions to carry out the lowest-level, most detailed sub-sub-... tasks. Once these functions are written, you can write the functions that carry out the next higher level in terms of these more primitive, lower level functions. And so on, until the function that accomplishes the top-level task can be written in terms of the functions that carry out its component sub-tasks.

The important concept here is that the functions at each level should be as independent as possible from both the lower level functions it uses and the higher level function that use it.

The only assumptions that should be made are about these higher or lower level functions concern the syntax and semantics of their arguments and returned values, and some information about their side-effects on the environment.

In particular, NO assumptions should be made about the details of how they are implemented.

The goal of abstraction is to be able to write programs in which bugs, required changes, etc. will be isolated to only those few functions that are directly relevant to the bug or change or whatever.

For example:

Consider a program that does some complex computations, printing out intermediate results in the Exec window along the way.

The task of printing out the intermediate results on the screen should be isolated into a separate function (or functions) and not be part of the functions that do the calculations.

Then, if you want to change the printout to use special fonts or to go to a file rather than the screen, you have to change only the printing function(s). No changes will have to be made to all of the functions that do the computational work.

The advantages of properly abstracted programs are too many to enumerate!!! Ease of programming, ease of debugging, ease of modification, and so on.

We covered *data abstraction*, i.e. abstraction when dealing with data structures, in glorious detail earlier (see LispCourse #s 24 thru 26).

Procedural abstraction is the analogous concept in the realm of writing procedures to carry out arbitrary tasks.

Procedural abstraction is the art of writing functions by *combining* calls to other, more detailed functions.

There are various schemes for doing this *combination* of function calls to make interesting and useful functions in Lisp.

The various schemes are called *control structures* and are the topic of the next section.

Interlisp Control Structures

A *control structure* is a scheme for controlling the order of a set of function calls being made in order to accomplish some task.

In Lisp, there are a number of basic control structures available for use in writing your functions.

These control structures are generally implemented as functions that take an arbitrary number of SExpressions (i.e., function calls) and evaluate them in some specific order.

Review: Control Structures Already Covered

The following are basic control structures that we've already covered in previous LispCourses.

Sequential evaluation

As a default rule, when you can specify an arbitrary number of SExpressions for evaluation, these

SExpressions are evaluated in sequential order and the value of the last SExpression is returned as the value of the whole set of SExpressions.

The most salient instance of this is in the body of a function definition.

Another instance is after the DO or COLLECT in a FOR loop.

Embedding

Perhaps, the most prevalent control structure in Lisp is embedding function calls as arguments to other (LAMBDA) function calls.

Example:

```
(SQRT
  (PLUS
    (SQUARE (DIFFERENCE
              X1 X2))
    (SQUARE (DIFFERENCE
              Y1 Y2))))
```

For LAMBDA function calls, the Lisp evaluator insures that the arguments will be evaluated in sequential order (from left to right) before the function is applied.

In the preceding example this means that before the SQRT function is applied, the PLUS is evaluated. But before the PLUS is applied, the two SQUARE calls are evaluated in order. And so on.

Note: NLAMBDA functions do not have this property.

If the arguments to an NLAMBDA function are evaluated at all, they are evaluated by the function itself and not by the Lisp evaluator.

Since the function can evaluate its arguments in any order, the order of evaluation of an NLAMBDA function's arguments is impossible to predict, *a priori*.

However, if you happen to know how an NLAMBDA function evaluates its arguments, you can use embedding.

Example:

```
(SETQ FOO (SETQ BAR (PLUS A
B)))
```

Sets both FOO and BAR to the result of (PLUS A B).

COND

COND implements a conditional (IF-THEN-ELSE) control structure in Lisp. (See LispCourse #4, page 5)

COND has the form:

```
(COND (Test1 Consequent1)(Test2 Consequent2)
...)
```

COND evaluates each *TestI* in turn until the first one that evaluates to non-NIL.

It then evaluates each SExpression in the *Consequent* after this first non-NIL *Test* and returns the value of the last of these SExpressions. (Or the value of the *Test* if there are no SExpressions in the *Consequent*.)

If there is no *TestI* that evaluates to non-NIL, COND returns NIL.

Example:

```
(COND
((LITATOM SExpr)(LookupValue SExpr
Stack))
```



```
((LISTP SExpr)(LC.Eval SExpr Stack))  
(T SExpr))
```

In English:

If SExpr is a LITATOM, Lookup its value;

Else if SExpr is a list, the eval that list,

Otherwise, return SExpr.

AND, OR

AND and OR are two functions that are meant to be logical functions, but often serve to control the order of evaluation of SExpressions.

Both AND and OR take an arbitrary number of SExpressions as arguments.

AND ž evaluates the SExpressions in sequential order but stops at the first SExpression that evaluates to NIL and returns NIL. If no such SExpression is found, AND returns the value of the last SExpression.

OR ž evaluates the SExpressions in sequential order but stops at the first SExpression that evaluates to non-NIL and returns its value. If no such SExpression is found, OR returns NIL.

AND is often used to replace single clause COND expressions:

1. *(AND YesFlg (SETQQ Blat Fumble))*
is equivalent to
(COND (YesFlg (SETQQ Blat Fumble))).
2. *(AND (SETQ Baz (CAR Fu)) (PLUS 3 Baz))*
is equivalent to
(COND ((SETQ Baz (CAR Fu)) (PLUS 3 Baz))).
3. *(AND Window TextStream ID*
LC.SetID ID Window TextStream))
is equivalent to
(COND
((AND ID Window TextStream)

```
(LC.SetID ID Window
  TextStream))))
```

OR is often used to replace COND statements with a number of clauses, each having a test and no consequents:

1. *(OR*
(WINDOWP Window)
(SETQ Window (CREATEW)))
 is equivalent to
(COND
 ((WINDOWP Window))
 ((SETQ Window (CREATEW))))

2. *(OR List Array*
 (ERROR "Neither list nor array
 present"))
 is equivalent to
(COND
 (List)
 (Array)
 ((ERROR ".."))

3. *(SETQ Foo*
 (OR Arg1 Arg2 (SETQ Arg1 (SETQ Arg2
 Arg3))))
 is equivalent to
(SETQ Foo
 (COND
 (Arg1)
 (Arg2)
 ((SETQ Arg1 (SETQ Arg2
 Arg3))))

Iteration: FOR, WHILE, and UNTIL

FOR, WHILE, and UNTIL implement iterative loops in Interlisp that allow you to repeat a set of operations for each element in a sequence or list of elements.

The basic notion of an iterative loop is that there is an *iteration sequence* (e.g., a list or a sequence of integers), an *iterative variable*, and a *body of SExpressions* that (may) use the iterative variable.

For each element in the iteration sequence in turn, the iteration loop binds the iterative variable to this element and then evaluates the SExpressions in the body.

The clauses in a basic FOR, WHILE, or UNTIL statement specify the iteration sequence, the iteration variable, and the body.

See LispCourse #5, page 1 for a description of the basic FOR, WHILE and UNTIL loops.

Additional clauses, can specify what to do with the results of evaluating the body (e.g., the COLLECT statement), under what conditions to terminate the iteration before the iteration sequence is exhausted, etc.

In particular,

For conditional execution of the body of an iterative loop, see LispCourse #26, page 16 for a description of the WHEN clause in FOR/WHILE loops.

For multiple iteration variables, see LispCourse Homework #28, page 6 for a description of the AS clause in FOR/WHILE loops.

Loaded example:

```
(FOR Item IN List
  AS Index FROM 1 TO (LENGTH List) BY
  1
  WHILE (KEYDOWNP 'A)
  WHEN (NUMBERP (ELT Array Index))
  COLLECT (ADD1 (SETA Array Index
  Item)))
```

This example has 2 iterative sequences (*List & FROM 1 TO (LENGTH List) BY 1*) and, correspondingly, two iterative variables (*Item & Index*).

It has a standard body thgat uses both iteration variables -- (*ADD1 ...*)

It has an early termination condition -- *WHILE ...*

It has conditional evaluation of its body -- *WHEN ...*

And it returns the list of evaluation results from each iteration -- *COLLECT ...*

LET

LET is an implementation of the basic sequential evaluation of SExpressions. Thus as a control structure its not very interesting. Its basic use is for binding variables.

(SEE LispCourse #34, page 19)

PROG with RETURN

Like LET, PROG is used basically for binding variables.

(SEE LispCourse #34, page 23)

Basically, PROG implements a sequential evaluation of SExpressions.

However, with an embedded RETURN statement, you can terminate this sequential evaluation at any point and force PROG to return an arbitrary value.

PROG with RETURN together with COND, AND, OR, etc. can be used to built tailor-made control structures.

Example:

```
(PROG NIL
  (AND (WINDOWP Window)(RETURN
    Window))
  (AND
    (WINDOWP
      (SETQ Window
        (WindowOfText
          TextStream))
      (RETURN Window))
    (SETQ Window (CREATEW))
    (AND
      (WindowOffScreenP Window)
      (RETURN Window))
```

```

... Do lots of work with the window ...
(RETURN Window))

```

This is an example of using PROG with no binding list just to take advantage of the RETURN statements.

The example is actually equivalent to the following COND statement. But the PROG version is much easier to follow than the COND version (at least for an old FORTRAN programmer!).

```

(COND
  ((WINDOWP Window) Window)
  ((WINDOWP
    (SETQ Window
      (WindowOfText
        TextStream)))
    Window)
  ((WindowOffScreenP
    (SETQ Window
      (CREATEW)))
    Window)
  (T ... Do lots of work with the
  window ...
  Window))

```


PROG with GO

Note that PROG also supports a GO clause that can be used to construct tailor-made iterative loops.

The GO clause is like the FORTRAN GOTO.

Since GOTOs are considered BAD programming style, we won't cover GO here.

See pages 4.3 & 4.4 of the IRM for more information.

Note, however, that FOR/WHILE/UNTIL loops are actually constructed by CLISP from PROG/GO/RETURN!!!

Other Control Structures

SELECTQ

SELECTQ is a control structure that will select a sequence of SExpressions to evaluate based on the value of its first argument.

SELECTQ has the following format:

(SELECTQ *Selector Clause1 Clause2 ... DefaultSExp*)

SELECTQ is an NLAMBDA function.

Selector is an arbitrary SExpression that evaluates to an atomic value.

DefaultSExp is an arbitrary SExpression. It MUST be present. (Its a common mistake to forget it!!)

Each *ClauseI* is an SExpression of the format:

(Key *SExpr1 SExpr2 ...*)

Key is either a single atom or a list of atoms.

The *SExpri* are arbitrary SExpressions.

SELECTQ works as follows:

Selector is evaluated. And then compared against the *Key* (which is unevaluated) of each *Clause* in turn until a match is found.

A match is defined as follows:

If *Key* is an atom, then *Key* must be **EQ** to the value of *Selector*.

If *Key* is a list, then the value of *Selector* must be a **MEMEBER** of *Key*.

For the first *Clause* whose *Key* matches, each of the SExprs in the *Clause* are evaluated in turn, and the value of the **SELECTQ** is the result of the last evaluation.

If no matching clause is found, then *DefaultSExpri* is evaluated and the result is returned as the value of the **SELECTQ**.

Example:

```

1_(DEFINEQ
  (ProcessCommand
    (LAMBDA (Command)
      (SELECTQ Command
        (Create
          (CREATEW))
        (Close
          (SETQ Window
            (GetWindow))
          (CLOSEW Window)

```

```

Window)
(Move
  (SETQ Window
    (GetWindow))
  (MOVEW Window)
  Window)
((Shape Reshape)
  (SETQ Window
    (GetWindow))
  (SHAPEW Window)
  Window)
(ERROR "Unknown Command"
  Command))))

```

(ProcessCommand)

3_ (ProcessCommand 'Create)

[Prompt for a region and create a window in it.]

{WINDOW}#43,12557

4_ (ProcessCommand 'Shape)

[Prompt for a window and reshape it.]

{WINDOW}#43,12345

4_ (ProcessCommand 'Display)

Unknown Command

Display

EFS: What is the format of the COND statement equivalent to SELECTQ.

PROG1, PROG2, PROGN

PROG1, PROG2, and PROGN are three variations on the theme of sequential evaluation.

All of these functions sequential evaluate an arbitrary number of SExpressions.

They differ in the value that they return. In particular:

PROG1 returns the value the **first** SExpression.

PROG2 returns the value of the **second** SExpression.

PROGN returns the value of the **last** SExpression.

Note: These are NOT variations of the PROG, they do NOT bind any variables.

PROG1 and PROG2 are used when need to evaluated a sequence of SExpressions in a particular order, but want to return the value of the first or second SExpression rather than the last as is usually the case.

Example from LC.Apply from Homework#35:

```
( LAMBDA
  ...
  (PROG1
    (LC.Eval (CAR (LAST SExprs))
      Stack)
    (LC.Unbind Stack)))
```

The problem here is that LC.Apply has to evaluate the last SExpr BEFORE unbinding the stack, but needs to return the value of the evaluation AFTER unbinding the stack.

PROG1 solves the problem efficiently. In the Homework #35 solutions, we needed to bind an extra variable (i.e., Result) using LET in order to accomplish the same thing:

```
( LAMBDA
```

```
...
```

```

(LET (Result)
  ...
  (SETQ Result
    (LC.Eval (CAR (LAST
      SExprs)) Stack)
    (LC.Unbind Stack)
    Result))

```

PROGN is used where only a single SExpr is allowed, but you need to evaluate several SExpressions in order. In this case, you just wrap the SExpressions in a PROGN, which counts as a single SExpression.

Example:

```

(SELECTQ
  Command
  (Create (CREATEW) 'Created)
  (Close
    (SETQ Window
      (GetWindow))
    (CLOSEW Window)
    'Closed)
  (PROGN
    (SETQ Window
      (GetWindow))
    (MOVEW Window)
    'Moved))

```

The problem here is that the default clause of a SELECTQ must be a single SExpression. But we want to do three things in the default case: get the window, move it, and return the atom Moved. Solution is to use a PROGN to wrap the three function calls into a single unit.

Simple Repetition: RPT and RPTQ

RPT and RPTQ implement a simple repetition control structure.

(RPT *N SExpr*) ž Evaluates *SExpr* *N* times and returns the value of the last evaluation. Before each evaluation the free variable **RPTN** is set to the number of repetitions still to take place. RPTN can be used inside of *SExpr*.

RPT is a LAMBDA-spread function. Therefore, it is actually the value of the argument that is being repeatedly evaluated.

Example:

```
1_ (RPT 5 (QUOTE (PRINT RPTN)))
```

```
5
```

```
4
```

```
3
```

```
2
```

```
1
```

```
2_ (RPT 3 (QUOTE (CREATEW)))
```

```
[Creates 3 windows]
```

```
{WINDOW}#34,00123
```

(RPTQ *N SExpr1 SExpr2 ...*) ž Evaluates each *SExprI* *N* times and returns the value of the last evaluation. Order of evaluation is to do each *SExprI* once, then repeat. Before each evaluation the free variable **RPTN** is set to the number of repetitions still to take place. RPTN can be used inside of *SExpr*.

RPT is a NLAMBDA-nospread function.

Example:

```
3_ (SETQ A 6)
```

```
6
```

```
4_(RPTQ 6 (SETQ A (ADD1 A)) (SETQ A (SUB1
(SUB1 A))))
```

```
0
```

```
5_ A
```

```
0
```

```
6_(RPTQ 2 (PRINT "A")(PRINT "B"))
```

```
"A"
```

```
"B"
```

```
"A"
```

```
"B"
```

```
"B"
```

Mapping functions: MAP, MAPC, MAPLIST, MAPCAR

Mapping functions are a common Lisp control structure.

A mapping function iterates over some data structure (usually a list) and applies some other function to each element of the data structure in turn.

Interlisp has several built in mapping functions for lists. We will cover only MAPC and MAPCAR.

(MAPC *List WorkFn NextFn*) ž Does (*APPLY* WorkFn (CAR List)*), then does (*APPLY* WorkFn (CAR (NextFn List))*), then does (*APPLY* WorkFn (CAR (NextFn (NextFn List)))*) and so on until (*NextFn (... (NextFn List)*) returns NIL. MAPC always returns NIL.

If *NextFn* is NIL, CDR is used.

Examples:

(MAPC (OPENWINDOWS) 'CLOSEW) will CDR down the list of OPENWINDOWS and CLOSEW each window in turn.

(MAPC (OPENWINDOWS) 'CLOSEW 'CDDR)

will map down every other window in the OPENWINDOWS list (due to the use of CDDR rather than CDR) and CLOSEW each of these windows in turn.

(MAPCAR *List WorkFn NextFn*) ž Essentially the same as MAPC, but returns a list of the values returned by all the evaluations of *WorkFn*.

Examples:

(MAPC (LIST 1 2 3 4 5 6 7) 'ADD1) will CDR down the list (1 2 3 4 5 6 7) and ADD1 each item in turn, returning the result of all the ADD1s, i.e., (2 3 4 5 6 7 8).

(MAPC (LIST 1 2 3 4 5 6 7) 'ADD1 'CDDDR) will map down every third element the list (1 2 3 4 5 6 7) and ADD1 each of these elements in turn, returning the result of all the ADD1s, i.e., (2 5 8).

Note: Section 5.3 of the IRM describes several relatives of MAPC and MAPCAR.

Note: MAPC and MAPCAR are nearly identical in functionality to various FOR/COLLECT constructions. (In fact, CLISP actually implements many FOR loops as MAPC/MAPCAR type functions) Since FOR is much easier to use, I seldom use MAPC and MAPCAR and their relatives.

But, the notion of a mapping function appears other places in Lisp, where functions like FOR are not available.

For example, TEdit has a function called TEDIT.MAPPIECES that allows you to apply an arbitrary function to every "piece" of text in a TEdit text.

Recursion -- The ultimate control structure in Lisp

To be completed

References

The LAMBDA/NLAMBDA and Spread/Nospread distinctions are covered in 5.1 of the IRM. Sections 5.1.0 thru 5.1.4 are most relevant. Also look at Section 5.1.7.

Functional arguments and FUNCTION are covered in Section 5.4 of the IRM.

Most control structures (e.g., COND, AND, OR, SELECTQ, PROG, PROG1, PROGN, etc.) are covered in Chapter 4 of the IRM.

RPT, RPTQ, MAPC and MAPCAR are covered in Section 5.3 of the IRM.

LispCourse #37: Recursion; Organizing Large Programs

Recursion – The ultimate Lisp control structure

Recursion is THE control structure in Lisp.

Reasons:

Recursion is the right way to organize many procedures!

The structure of Lisp encourages you to think recursively.

The structure of Lisp makes it very convenient to write recursive functions.

To review: Recursion a control structure in which the definition of a function includes a function call to itself.

The basic idea: many problems can be solved by combining the solutions to two or more smaller problems, each of the same nature as the larger problem.

For example: to count the number of atoms in a list, you can combine by addition the number of atoms in the CAR of the list and the number of atoms in the CDR of the list.

This works fine as long as the CAR and CDR are both non-empty lists, because you can then (recursively) apply the same procedure to count their atoms.

But you must have an alternative procedure when the CAR and/or CDR is a non-list or an empty list because the count atoms in list procedure won't work in these cases.

All recursive functions have the same basic underlying structure:

```
(DEFINEQ
  (RecursiveFunction (LAMBDA (Args...)
    (COND
      ((Is Args... a terminating case?) (Process terminating case))
      (T (Call combining function with args:
          (Call RecursiveFunction using first "part" of Args...)
          (Call RecursiveFunction using second "part" of Args...)
          ...
          (Call RecursiveFunction using last "part" of Args...))))))
```

The CountAtoms procedure from LispCourse #5, page 12 implements the count-atoms-in-a-list procedure outlined in the example above and follows exactly this standard recursive function format:

```
(DEFINEQ
  (CountAtoms (LAMBDA (List)
    (COND
      ((NULL List) 0) [Terminating case]
      ((LITATOM List) 1) [Terminating case]
      (T (PLUS [Combining Function]
          (CountAtoms (CAR List)) [Recursion: 1st part of List]
          (CountAtoms (CDR List)))))) [Recursion: 2nd part of List]
```

As another example, consider the function EVAL (LispCourse #34, page 5):

```
(DEFINEQ
  (EVAL (LAMBDA (SEExpr)
    (COND
      ((LITATOM SEExpr)
        (LookupValue SEExpr) [Terminating case]
      ((NLISTP SEExpr) SEExpr) [Terminating case]
      (T (APPLY (CAR SEExpr) [Combining Function]
          (FOR Item in (CDR SEExpr)
            DO (EVAL Item)))))) [Recursive call on each part of SEExpr]
```

EFS: Trace EVAL and APPLY during the evaluation of $(\text{CountAtoms } '(A B) D E)$. You will see that recursion works in Lisp because each time CountAtoms is APPLIED to another sub-list, a new stack frame is created and *List* is rebound to the value of that sub-list.

Recursion works in Lisp because

- 1) the stack is an expandable data structure that holds the results of incomplete calculations while (recursive) sub-calculations are going on
- 2) the process of rebinding variables on the stack allows you to apply a given function to a new set of arguments while the processing of a prior call to that **same** function is still incomplete

Recursion can be *indirect*, e.g., a FunctionA calls FunctionB which in turn calls FunctionA again.

The Lisp evaluator (LispCourse #34, pages 5 & 6) is an excellent example:

EVAL recursively calls EVAL, but it also calls APPLY.

APPLY in turn calls EVAL.

So EVAL calls APPLY which calls EVAL.

For example on a list:

EVAL calls EVAL on each item in the CDR of the list, then calls APPLY using the CAR and the evaluated arguments.

APPLY in turn calls EVAL on each list in the function definition of the CAR of the original list.

Types of Recursion

Single Recursion (and tail recursion)

Singly recursive functions are recursive functions that call themselves only once during each application of the function.

Note the a call to itself may appear several times in the function definition (e.g., in several clauses of a COND statement), but only one of these should be evaluated during each call to APPLY.

Examples:

```
(DEFINEQ
  (MEMBER (Thing List)
    (* * Is Thing EQUAL to any item in List?)
    (COND
      ((NULL List) NIL)
      ((EQUAL Thing (CAR List)) T)
      (T (MEMBER Thing (CDR List))))))
```

```
(DEFINEQ
  (LENGTH (List)
    (* * Return the number of items in List)
    (COND
      ((NULL List) 0)
      (T (ADD1 (LENGTH (CDR List))))))
```

```
(DEFINEQ
  (REVERSE (List)
    (* * Make a copy of List with the items in reverse
    order)
    (COND
      ((NULL List) NIL)
      (T (APPEND (REVERSE (CDR List))
        (CAR List))))))
```

Note that these three function are all singly recursive, but differ in what they do to the result returned by the recursive function call.

Each call to MEMBER just returns the value of the recursive function call (or NIL if List is NULL).

Each call to LENGTH returns 1 plus the result returned by the recursive function call.

Each call to REVERSE returns a computation based on the value returned by the recursive function call and a computation based on the value of the main argument.

Because of these differences, an all-knowing Lisp evaluator would have to maintain different amount of state about each of these recursive functions during evaluation. In particular,

For MEMBER, the evaluator could throw out all information about a function call (i.e., its stack frame) once it made its recursive call.

This is because there is no information about the state of the computation to be maintained once the recursive call is made. The value of the highest level call is the value of the lowest-level function call, with no modifications.

For LENGTH and REVERSE, the evaluator needs to maintain information about each recursive call until the recursion is complete.

This is because when the recursive call is made, the computation at the current level is incomplete. In LENGTH, for example, the recursive computation is being done in the middle of an ADD1 evaluation. Information about the status of ADD1 needs to be maintained until the recursive call to LENGTH has returned a value.

MEMBER is a *tail recursive* function, LENGTH and REVERSE are not.

A *tail recursive* function is recursive function whose value depends only on the value returned by a recursive call or some value that is computed directly without a recursive call.

Tail recursive functions are important because a good Lisp evaluator can eliminate the unnecessary stack frames during the computation, making these computations very efficient. (See comparison of iteration and recursion below).

Recursive functions that are not tail recursive need to maintain their intermediate state on the stack, and therefore can be relatively expensive to compute.

Double (or more) Recursion

Doubly recursive functions are functions that during each application call themselves two (or more) times.

CountAtoms (above) is a perfect example: for each invocation (where List is a LISTP) it recurses once on its CAR *and* once on its CDR.

As a second example, the following function is like MEMBER but rather than just looking at the top-level elements in a list, it descends into each sub-list looking for a sub-item that might be EQUAL to its first argument:

```
(DEFINEQ
  (MEMBER*
    (LAMBDA (Thing List)
      (COND
        ((NULL List) NIL)
        ((NLISTP List) (EQUAL Thing
                               List))
        (T (OR
            (MEMBER* Thing (CAR
                          List))
            (MEMBER* Thing (CDR
                          List))))))))))
```

In general, doubly recursive functions cannot be tail recursive because the evaluator always needs to maintain state (i.e., the result of) about the result of the first recursive call while evaluating the second recursive call.

The exception to this rule is doubly recursive functions that are evaluated for side-effect only and do not return a useful value. In this case, computation can be done on the returned value of a recursive call, so no state has to be maintained.

Double recursion is often called tree recursion because it is used to traverse tree structures. See Homework for examples.

Iteration versus Recursion

Iteration and recursion are in many ways similar control structures.

In fact, any iterative procedure can be *automatically* converted into an equivalent recursive procedure.

For example:

```
(LET ((Sum 0))
    (FOR Item IN List DO (SETQ Sum (PLUS Sum Item)))
    Sum)
```

can be converted to

```
(DEFINEQ
  (Sum (LAMBDA (List)
        (COND
          ((NULL List) 0)
          (T (PLUS (CAR List) (Sum (CDR List)))))))
```

In contrast, not every recursive procedure can be converted into an equivalent iterative procedure:

CountAtoms, for example, has no iterative equivalent.

Tail recursive procedures are important, however, because they can always be rewritten in an equivalent iterative form.

For example, MEMBER can be written as:


```
(FOR Item IN List WHEN (EQUAL Thing Item) DO
  (RETURN T))
```

It is interesting that the Sum function just above is NOT tail recursive, but *is* equivalent to an iterative procedure. The reason is that the Sum function can be rewritten as the following equivalent tail recursive procedure, which in turn can be rewritten as an iterative procedure:

```
(DEFINEQ
  (Sum2 (LAMBDA (List Total)
    (COND
      ((NULL List) Total)
      (T (Sum2 (CDR List) (PLUS (CAR List)
        Total)))))))
```

The ability to rewrite recursive procedures as iterative procedure is important because iterative computations are in general much more efficient than recursive computations.

This is because recursive function calls require the evaluator to maintain a set of stack frames containing information about partially completed computations. This stack grows as the recursion gets deeper.

In contrast, iterative procedures require only a fixed set of state variables to be maintained (e.g., the iterative variable). The number of these state variables does not increase with the number of iterations.

It is important to not that while recursive *procedures* cannot always be rewritten as iterative *procedures*, it is often the case that a given *problem* can be solved using a recursive procedure *or* an iterative procedure that is not strictly equivalent to the recursive procedure.

For example:

The REVERSE function above cannot be written iteratively because it is not tail recursive.

But we can write two slightly different procdures that reverse the order of a list:

```
(DEFINEQ (ReverseTR (LAMBDA (List ResultSoFar)
  (COND
    ((NULL List) ResultSoFar)
    (T (ReverseTR
        (CDR List)
        (CONS (CAR List) ResultSoFar)))))))
```

```
(DEFINEQ (ReverseIt (LAMBDA (List)
  (LET (Result)
    (FOR Item IN List
      DO (SETQ Result (CONS Item
        Result))))
    Result))))
```

Ultimately, the choice between iteration and recursion is one of programming style and programming ease.

Some problems are best thought about recursively. In this case, it easiest and clearest to write recursive functions.

Other problem are best thought of iteratively. In this case, iterative functions are probably best.

The only exception is when efficiency considerations are important. In this case, iterative solutions, if possible, are probably called for.

Organizing Large Programs by Object and Operation

Many large programming projects have the structure that there are a number of different types of objects in the world and a number of operations that can be applied to any of these types of objects.

For example, in arithmetic programming the objects are integers, real numbers, complex numbers, etc. There are also a few standard operations, addition, subtraction, multiplication and division.

In a text editor, the objects are characters, words, lines, and paragraphs. The operations might be insert, delete, replace, transpose, etc.

It is helpful to illustrate the structure of these programming projects using a table of objects and operations.

For example:

		Objects		
		Integer	Real	Complex
Operations	Addition	AddInts	AddReals	...
	Subtraction	SubInts	...	
	Multiplication	...		
	Division			

Each cell of this table defines the need for a function to carry out the designated operation on the designated type of object.

For example, the cell in the first row and first column of the table above indicates the need for a function to add integers.

Similarly, the cell in the first row, second column defines the need for a function that adds real numbers.

And so on.

When organizing the program to handle the given objects and operations, you have several choices:

- 1) You can organize your functions by the rows in the table, i.e., by operation.
- 2) You can organize your functions by the columns in the table, i.e., by object type.
- 3) You can organize your functions using the whole table.

Organization by Operation

If you organize by operation, you would write a single operation for each operation. This function would then determine the type of its arguments and then call the appropriate function to do the work.

For example:

The addition function might look like:

```
(DEFINEQ
  (ADD
    (LAMBDA (N1 N2)
      (COND
        ((AND (FIXP N1)(FIXP N2))
         (IntegerAdd N1 N2))
        ((AND (FLOATP N1)(FLOATP
                N2))
         (RealAdd N1 N2))
        ((AND (ComplexP N1)(ComplexP
                N2))
         (ComplexAdd N1 N2))
        (T (ERROR "Unknown argument
                  types or argument types different"
                  (LIST N1 N2)))))))
```

The subtraction, multiplication, and division functions would have similar structures.

Once the generic operations functions were written, you could write all further functions in terms of the generic operators.

For example, the following function would add the items in a list of integers or a list of reals or a list of complex numbers:

```
(DEFINEQ (Sum (LAMBDA (List)
  ((NULL List) 0)
  (T (ADD (CAR List)(Sum (CDR List)))))))
```

Organization by Type of Object

Alternatively, you could organize your program by object type.

In this case, you might create a RECORD or DATATYPE for each object type containing each a function for each operation to be carried out on that object type.

For example, the following RECORD would be used for defining the arithmetic object types:

```
(RECORD ArithObjectType (PredicateFn AddFn SubFn MultFn
  DivFn))
```

Integers would then be defined by:

```
(create ArithObjectType
  PredicateFn _ (FUNCTION FIXP)
  AddFn _ (FUNCTION IntegerAdd)
  SubFn _ (FUNCTION IntegerSubtraction)
  MultFn _ (FUNCTION IntegerMultiplication)
  DivFn _ (FUNCTION IntegerDivision))
```

You would then maintain a list of all of the arithmetic object types in the system.

Once this list was created, you could write a generic ADD function as follows:

```
(DEFINEQ (ADD (LAMBDA (N1 N2)
  (LET (TypeRecord)
    (FOR Type IN ListOfSystemTypes
      WHEN (AND (APPLY* (fetch PredicateFn of
        Type) N1)
        (APPLY* (fetch PredicateFn of
          Type) N2))
      DO (RETURN Type)))
    (COND
      (TypeRecord
        (APPLY* (fetch AddFn of TypeRecord) N1
          N2))
      (T
        (ERROR "Unknown argument types or
          argument types different" (LIST N1 N2))
```

Organization using the table: Data-directed programming

Alternatively, you could organize your program by operation and object type.

In this case, you would create a table of functions indexed by object type and operations.

For example:

```
(SETQ Table
  ((Integer Add IntegerAdd)
   (Integer Subtract IntegerSubtract)
   ...
   (Real Add RealAdd)
   ...
   (Complex Divide ComplexDivide)))
```

You could then create a generic function called *Operate* that carried out a given operation on a given set of arguments.

Operate would look like:

```
(DEFINEQ (Operate (LAMBDA (Operation N1 N2)
  (LET (TypeN1 TypeN2 Function)
    (* * Get the type of the arguments)
    (SETQ TypeN1 (GetType N1))
    (SETQ TypeN2 (GetType N2))
    (COND
      ((OR
        (NEQ TypeN1 TypeN2)
        (NULL TypeN1)
        (NULL TypeN2))
       (ERROR ...))
      (* * Lookup the function in the table)
      (SETQ Function (FOR Item IN Table
        WHEN (AND (EQ (CAR
          Item) TypeN1)
```

```

(EQ (CADR
Item)
Operation))
DO (RETURN (CADDR
Item))))
(* * Apply the function)
(COND
(Function (APPLY* Function N1 N2))
(T (ERROR ...))))))

```

The Sum function could then be written as:

```

(DEFINEQ (Sum (LAMBDA (List)
((NULL List) 0)
(T (Operate 'Add (CAR List)(Sum (CDR List)))))))

```

Organization by Object Instance: Object-oriented programming

All of the preceding organizations looked at the type of an object in order to determine what functions to use on that object.

An alternative method would be to allow each data object to carry around with it the functions that are necessary to operate on that object.

In this case, each data object would be a RECORD or DATATYPE of the following form:

```

(RECORD ArithObject (TypeName Value AddFn SubFn MultFn DivFn))

```

You could then write an ADD function as follows:

```

(DEFINEQ (ADD (LAMBDA (N1 N2)
(COND
((NEQ (fetch (ArithObject TypeName) of N1)
(fetch (ArithObject TypeName) of N2))
(ERROR ...)))

```

```
(APPLY* (fetch (ArithObject AddFn) of N1)
        (fetch (ArithObject Value) of N1)
        (fetch (ArithObject Value) of N2))))
```

You would still need one function indexed by object type, the function that creates a new object of that type. This would create a new object with its own attached functions. The attached functions could be supplied specially for each individual object or could be the same for all objects of a given object type.

For example, the following might be a function that creates the integer object:

```
(DEFINEQ (MakeInt (LAMBDA (Value AddFn SubFn MultFn
                        DivFn)
            (create ArithObject
                    TypeName _ 'Integer
                    Value _ Value
                    AddFn _ (OR AddFn
                                (FUNCTION DefaultIntegerAddFn))
                    SubFn _ (OR SubFn
                                (FUNCTION DefaultIntegerSubFn))
                    MultFn _ (OR MultFn
                                (FUNCTION
                                 DefaultIntegerMultFn))
                    DivFn _ (OR DivFn
                                (FUNCTION
                                 DefaultIntegerDivFn))))))
```

The integer of value 5, with default functions would then be created using the function call: *(MakeInt 5)*.

Similarly, an integer of value 7 with a special AddFn would be created using: *(MakeInt 7 (FUNCTION MyAddFn))*

Much of the Interlisp system is programmed in this style.

For example, each window in the system is a DATATYPE of the following form:

```
(DATATYPE WINDOW (... CLOSEFN SHRINKFN MOVEFN ...))
```

Each window in the system has attached to it the functions that specify how to close it, how to shrink it, how to move it, etc.

When the window is created, default functions are placed in these fields in the WINDOW datatype unless otherwise specified by the user.

The function CLOSEW (SHRINKW, MOVEW etc) in Interlisp is implemented as follows:

```
(DEFINEQ (CLOSEW (LAMBDA (Window)
              (APPLY* (fetch (WINDOW CLOSEFN) of
                        Window) Window))))
```

At any time, the user can change the behavior of a window by changing its CLOSEFN, MOVEFN, SHRINKFN or whatever.

Choosing from among these organizations

Choosing among these organizations is largely a matter of programming style and the type of problem you are dealing with.

If the program has a fixed number of objects, but may increase in the number of operations, then an operation-based organization may be best since each added operation means adding a single new function.

If the program has a fixed number of operations but an increasing number of object types, then an object type-based organization might be best since each added object type would involve only a new object-type record or datatype.

If there is a variability in both object types and operations, table-based data-directed programming might be best.

Finally, if there are many specialized individual objects that need to be treated differently from the rest of the objects of their basic type (as is the case for windows), then an object-oriented organization would be the most efficient.

References

Recursion

Winston & Horn, Chapter 4

Touretzky, Chapter 8

Program Organization

Abelson and Sussman, Section 2.3

LispCourse #38: Files; Streams; Input/Output

Files

Files From the Programmer's Point of View

A file is just a data structure that is "outside of Lisp", e.g., on the local disk, on a file server, on the screen, etc.

Because a file is not part of any Lisp virtual memory, it can (generally) be shared with other Lisp virtual memories or with other programming environments, etc.

Since files are not part of Lisp, there are special protocols for creating and accessing files that are somewhat different from the way we access standard Lisp data structures.

What kind of data structure a file represents is largely determined by the protocols we use for accessing that file.

At the lowest level, all files are just variable-length one-dimensional vectors of bytes (i.e., 8-bit packets or integers between 0 & 255).

However, with the proper choice of input/output statements in Lisp, a file can be made to look like any arbitrary data structure, e.g., a list of SExpressions, a TEdit text, a Sketch, etc.

When dealing with files in a Lisp program, you frequently shift between viewing the file as simply an unstructured vector of bytes and viewing the file as some higher-level data structure.

File Names

Every file has a file name. The file name basically tells Lisp where to find the file in the outside world.

The syntax and use of file names was discussed in LispCourse #15, pages 1 to 15.

The following are Interlisp functions that allow the programmer to construct and decompose file names:

(FILENAMEFIELD *FileName* *FieldName*) ÿ returns the part of *FileName* that is specified by *FieldName*. Allowable *FieldNames* are HOST, DIRECTORY, NAME, EXTENSION, and VERSION. (See LispCourse #15 for the semantics of these field names. HOST is the same as Device).

Example:

```
1_ (FILENAMEFIELD '{dsk}<halasz>lisp>init.lisp 'NAME)
  init
2_ (FILENAMEFIELD '{dsk}<halasz>lisp>init.lisp
   'DIRECTORY)
  halasz>lisp
```

(UNPACKFILENAME *FileName*) ÿ returns *FileName* in prop list format, where the props are the allowable field names listed under FILENAMEFIELD.

Example:

```
3_ (UNPACKFILENAME '{dsk}<halasz>lisp>init.lisp;37)
  (HOST dsk DIRECTORY halasz>lisp NAME init EXTENSION lisp
   VERSION 37)
```

(PACKFILENAME *FieldName1* *FieldContents1* ... *FieldNameN* *FieldContentsN*) ÿ returns a *FileName* constructed using the information in the *FieldName/FieldContents* pairs. The *FieldNames* should be chosen from among the field names listed under FILENAMEFIELD plus the atom BODY.

If a *FieldName* is specified twice, the first is used.

If any *FieldName* is BODY, then the corresponding *FieldContents* is unpacked (using UNPACKFILENAME) and the resulting *FieldName/FieldContents* pairs are used in place of the BODY/*FieldContents* pair.

Also, if *FieldName1* is a list, it is assumed to be a prop list of the format returned by UNPACKFILENAME, in which case PACKFILENAME is APPLIED to this list (ignoring the remaining arguments!).

Example:

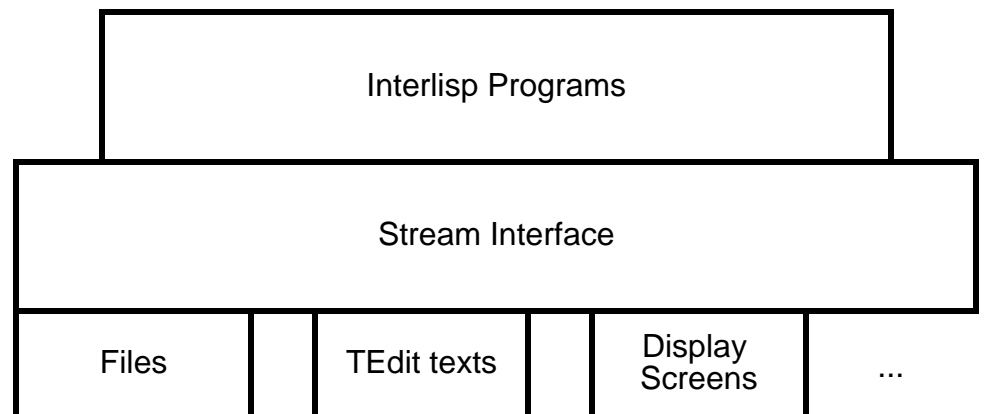
```
4_ (PACKFILENAME '(HOST dsk DIRECTORY halasz>lisp
NAME init EXTENSION lisp VERSION 37))
{dsk}<halasz>lisp>init.lisp;37
5_ (PACKFILENAME 'HOST '{DSK} 'NAME 'FileX)
{DSK}FileX
```

Streams

In the old days, Interlisp dealt with files directly, i.e., anytime you needed to refer to a file you used its file name.

This convention has been replaced by the concept of *streams* in Interlisp-D.

Streams are a uniform interface between Interlisp and devices or data structures that exist in the "outside world" including files, TEdit texts, display screens, etc.



In this new scheme, Interlisp programs deal with files through the stream interface. For example, you generally refer to files using the stream that represents the file rather than by the name of the file.

A *stream* is a data type. Each stream instance represents an active interaction with an "external" device or data structure, e.g. a file.

Each instance stores with it all kinds of information about the status of the interaction and the status of the external device or data structure.

For example, a stream for a file stores information about the file's name, where you are in the file, how long the file is, etc.

Low-level Access to Files Using Streams

Opening and Closing

Before a file can be accessed, it must be *opened*.

Opening a file, creates a stream for that file and caches in the stream instance all the necessary information. It also opens all the necessary communication pathways if the file is stored on a remote device such as a file server.

Opening a file also notifies the remote device to prevent access to the file by other users, if that's appropriate.

To open a file use:

(OPENSTREAM *FileName Access Recognition*) *ž* opens a file and creates a stream for it. OPENSTREAM returns the stream it creates. This stream should be used to refer to the file.

FileName is a standard Interlisp file name.

Access is one of INPUT, OUTPUT, APPEND, or BOTH.

If *Access* is INPUT, subsequent accesses to this file is limited to reading **from** the file

If *Access* is OUTPUT, subsequent access is limited to writing **to** the file. Moreover, if the file already exists it is erased.

If *Access* is APPEND, subsequent access is limited to writing **to** the file, but the current contents of the file (if any) are not erased.

If *Access* is BOTH, both reading and writing will be allowed.

Recognition is one of OLD, NEW, or OLD/NEW.

If *Recognition* is OLD, OPENSTREAM will look for an already existing file of the given file name.

If *Recognition* is NEW, OPENSTREAM will create a new version of the file with the given file name.

If *Recognition* is OLD/NEW, OPENSTREAM will look for an existing file first, but will create a new file if the old one doesn't exist.

Example:

```
6_ (SETQ FileX (OPENSTREAM '{DSK}Halasz.Lisp
'INPUT 'OLD)
{STREAM}#54,23123
```

(OPENP *FileNameOrStream Access*) ž returns the name of the file specified by *FileNameOrStream*, if that file is open with the access mode specified by *Access*. Returns NIL otherwise.

FileNameOrStream can be a file name or a stream or NIL. *Access* is one of the atoms described under OPENSTREAM or NIL.

If *Access* is NIL, then OPENP will return the file name if the file is open with any access mode.

If *FileNameOrStream* is NIL, then OPENP will just return a list of all open files.

Example:

```
7_ (SETQ FileX (OPENSTREAM '{DSK}Halasz.Lisp
'INPUT 'OLD)
{STREAM}#54,23123
8_ (OPENP Filex)
{DSK}Halasz.Lisp;23
9_ (OPENP '{DSK}Halasz.Lisp)
{DSK}Halasz.Lisp;23
10_ (OPENP Filex 'OUTPUT)
NIL
```

Once you are finished accessing a file, you should *close* the file.

Closing the file uncaches the information stored in the stream, deletes the stream, closes the communication pathways, and frees the file to be used by other users.

To close a file use:

(CLOSEF? *Stream*) ž closes the file specified by *Stream*, if that file is open. Returns the name of the file that it closed.

Stream should be a stream on an open file.

Example:

```
11_ (CLOSEF? Filex)
{DSK}Halasz.Lisp;23
12_ (CLOSEF? Filex)
NIL
```

(CLOSEALL) ž closes all currently open files. Returns a list of the names of the files that were closed.

File Pointers

Every open file has a *file pointer* associated with it.

The file pointer indicates the position in the file at which the next read or write will take place.

After each access to the file, the file pointer is updated to indicate the next position in the file (i.e., the next place to start reading or the next place to write).

Positions are measured in bytes, where the first byte in the file is position 0. So the file pointer varies between 0 and the one less than the length of the file.

When a file is first opened, the file pointer is set to 0 except if the access mode is append in which case the file pointer is set to the position of the end of the file.

At any point you can read the value of the file pointer to figure out where you are in the file.

For files that are on devices that support random access (e.g., local disk, IFS file servers, floppies but NOT NS file servers), you can set the file pointer to any arbitrary location in the file. The next read or write will then take place at this location.

The following functions read the file pointer:

(GETFILEPTR *Stream*) ý Returns the current file pointer for the open file represented by *Stream*, which should be a stream datatype.

(GETEOFPTR *Stream*) ý Returns the file pointer value for the location at the end of the open file represented by *Stream* (i.e., the number of bytes in the file).

The following functions can change the file pointer:

(RANDACCESSP *Stream*) ý Returns the file name of *Stream*, if that file is random accessible. NIL otherwise. *Stream* should represent an open file. If RANDACCESSP returns the file name, then SETFILEPTR can be used on the file.

(SETFILEPTR *Stream Position*) ž Sets the file pointer for the file represented by *Stream* to be *Position*. *Position* is any positive integer.

(SETFILEPTR *Stream* -1) is a special case meaning to set the file pointer to the end of the file.

SETFILEPTR results in an error if the file is not RANDACCESSP.

(FILEPOS *Pattern Stream Start End Skip*) ž analogous to STRPOS (See LispCourse #28, page 18). Searches through the file referenced by *Stream* looking for any sequences of characters that matches the characters in string *Pattern*. If a match is found, FILEPOS sets the file pointer to the file position where the match starts and returns this position as its value. If no match is found, FILEPOS returns NIL and the file pointer is unchanged

If *Start* is specified, the search begins at file position *Start*, otherwise search starts at the current file pointer.

If *End* is specified, the search terminates at file position *End* (if no match has been found), otherwise search ends at the end of the file.

If *SkipChar* is specified, any instance of *SkipChar* in the *Pattern* string will match any character in the file. (*SkipChar* is the wildcard character).

Reading and Writing

The following two functions are used to read/write a byte from/to an open file:

(BIN *Stream*) ž Reads the next byte, i.e., the byte right after the file pointer and returns it. The file pointer is updated one byte. The byte returned is a number between 0 and 255.

(BOUT *Stream Byte*) ž Write *Byte* at the next position in the file, i.e., the position right after the file pointer. The file pointer is updated one byte. *Byte* should be a number between 0 and 255.

Example

The following is an implementation of a COPYBYTES function that copies the specified bytes from a source file to a destination file:

```
(DEFINEQ (COPYBYTES
  (LAMBDA (SourceFile DestFile Start End)
    (LET (StopLoc
          (SourceStream (OPENSTREAM SourceFile
                           'INPUT 'OLD))
          (DestStream (OPENSTREAM DestFile 'OUTPUT
                                   'NEW)))
      (** Set up correct start and stop pointers)
      (AND (NOT Start)(SETQ Start 0))
      (AND (NOT End)(SETQ End (GETEOFPTR
                               SourceStream))))))
```

```

(COND
  ((NOT (GREATERP Start End)) (ERROR "Error
    Msg"))
  (SETQ StopLoc (MIN (GETEOFPTR SourceStream)End))
  (* * Move to start location)
(COND
  ((RANDACCESSP SourceStream)
   (SETFILEPTR SourceStream Start))
  (T (FOR Index FROM 0 TO (SUB1 Start)
      DO (BIN SourceStream))))
(* * Copy bytes until stop location)
(FOR Index FROM Start TO StopLoc
  DO (BOUT DestStream (BIN SourceStream)))
(* * Close files and return)
(LIST (CLOSEF? SourceStream) (CLOSEF?
  DestStream))))))

```

Higher-level File Input and Output

For most applications, accessing files a byte at a time is unnecessarily detailed.

Interlisp provides a number of higher-level input/output routines that can read and write Lisp objects (i.e., SExpressions) rather than bytes.

These higher-level routines are, of course, simply functions that eventually call BIN and BOUT to access the file.

They just provide a more convenient interface to files for most applications.

Higher-level Input Functions

The standard input function is READ:

(READ *Stream*) ÿ reads and returns the next SExpression (litolom, number, list, string, etc.) from the file referenced by *Stream*. *Stream* must reference an open file.

READ will start at the current file position and skip over white space characters (i.e., space, tab, carriage return) until the first non-space character. It will then read in the SExpression that begins at this character.

The interpretation of SExpressions is done using the same rules as in the Lisp Exec \tilde{z} lists are bounded by parentheses, strings are bounded by double quotes, atoms are bounded by white space or by the start/end of a list or string, numbers are atoms containing only digits, etc.

At the end of the READ, the file pointer for the file is set just after the last character of the SExpression.

(SKREAD *Stream*) \tilde{z} moves the file pointer for the file referenced by *Stream* ahead as if a READ had been done, but does not actually read anything. SKREAD returns NIL.

Used for skipping over things you don't actually want to read.

Some more specialized input functions are:

(READC *Stream*) \tilde{z} reads and returns the next character from the file referenced by *Stream*. The file pointer is moved ahead by 1.

READC reads all characters alike, ignoring the special effects of characters like double quotes and parentheses that would affect READ.

READC is like BIN, except it returns a character (i.e., a single character atom) rather than a byte (i.e., a number).

(RATOM *Stream*) \tilde{z} reads and returns one atom from the file referenced by *Stream*. *Stream* must reference an open file. Works like READ, except that parentheses and double quotes are considered to be single character atoms.

Example: If the next SExpression on the file is (FOO), then RATOM would return the atom %(. But if the next SExpression were ABC, then RATOM would return the atom ABC.

(RSTRING *Stream*) ž reads characters from the file referenced by *Stream* up to but not including the next white space character, parentheses, or double quote. Returns the characters read as a string.. *Stream* must reference an open file. Sets the file pointer to right after the last character read.

Example: If the next character on the file on the file is a space, then RSTRING would return the null string "". But if the next characters were ABC, then RSTRING would return the string "ABC".

Finally, it is some times desirable to read a whole file in a single shot using the following function:

(READFILE *StreamOrFileName*) ž reads SExpressions from the file referenced by *StreamOrFileName* up to but not including the first occurrence of the atom STOP, or until the end of the file is reached if no STOP is encountered. Returns a list of the SExpressions read.

StreamOrFileName will be opened if necessary.

READFILE uses READ to do its reading.

Higher-level Output Functions

Interlisp has a number of functions for printing SExpressions onto files.

Note that these functions just print the print name of each lisp object on the file.

This is no problem for atoms, lists, numbers, and strings.

But, arrays and datatype have print names which are particularly non-informative. In order to print these kinds of objects, you generally have to decompose them and print their parts separately using your own functions.

When writing out SExpressions on a file, you have a to make a decision as to how the SExpressions should be written:

- in a way that can be read back into Lisp using READ
- in a way that is more human-readable but cannot be read back in by READ.

Example:

When you print a string you can print the double quotes or not.

If you print the double quotes, then READ can recognize the characters as a string when it is reading the file later.

If you don't print the double quotes, then your output may look nicer, but READ will not be able to recognize the characters as a string when it tries to later read the file.

Interlisp provides higher-level output functions for both of these modes.

General Printing

The basic output functions are:

(PRIN1 *SExpression Stream*) ž prints SExpression on the open file referenced by *Stream*. Printing is done in a way that does not necessarily make it possible to READ the SExpression, e.g., the double quotes are omitted from strings and spaces inside of atoms are not escaped.

Examples:

The atom Foo% Bar is printed as Foo Bar

The string "ABC" is printed as ABC

The list (A B C) is printed as (A B C)

(PRIN2 *SExpression Stream*) ž prints SExpression on the open file referenced by *Stream*. Printing is done in a way that makes it possible to READ the SExpression, e.g., the double quotes are printed for strings and spaces inside of atoms are escaped.

Examples:

The atom Foo% Bar is printed as Foo% Bar

The string "ABC" is printed as "ABC"

The list (A B C) is printed as (A B C)

(TERPRI *Stream*) ž prints a carriage return on the open file referenced by *Stream*.

(SPACES *N Stream*) ž prints *N* spaces on the open file referenced by *Stream*.

(PRINT *SExpression Stream*) ž equivalent to (PRIN2 *SExpression Stream*) followed by a (TERPRI *Stream*).

Printing Numbers

The printing of numbers can be more precisely controlled using the following function:

(PRINTNUM *Format Number Stream*) ž prints *Number* on the open file referenced by *Stream* using the format specified by *Format*.

Format is a list structure with one of the following formats:

(FIX *Width Base Pad0Flg LeftFlushFlg*)

Indicates that *Number* is to be printed as a integer.

Width is the number of character spaces to reserve for the integer.

Base is the base in which the integer is to be printed. (Defaults to base 10.)

If *LeftFlushFlg* is NIL, then the number is right justified in the *Width* spaces.

Otherwise, it is left justified and the unused spaces to the right are filled with blanks.

If *Pad0Flg* is T, then any part of the *Width* spaces that are to the left of the number are filled with zeros.

Examples: (Note: the | characters are for exposition only!):

(PRINTNUM '(FIX 4) 100) prints | 100|.

(PRINTNUM '(FIX 4 NIL T NIL) 100) prints |0100|.

(PRINTNUM '(FIX 4 NIL NIL T) 100) prints |100|.

(FLOAT *Width DecWidth ExpWidth Pad0Flg*)

Indicates that *Number* is to be printed as a real number.

Width is the number of character spaces to reserve for the number.

DecWidth is the number of digits to appear to the right of the decimal point.

ExpWidth is non-NIL, specifies that the number should be printed in exponent format (i.e., scientific notation) with *ExpWidth* character spaces used for the exponent part.

PadChar if non-NIL specifies that the leading spaces to the left of the number are to be filled with zeros.

Examples: (Note: the | characters are for exposition only!):

(PRINTNUM '(FLOAT 4 2) 4.23) prints |4.23|.

(PRINTNUM '(FLOAT 5 2) 4.23) prints | 4.23|.

(PRINTNUM '(FLOAT 5 2 NIL 22) 4.23) prints |04.23|.

(PRINTNUM '(FLOAT 5 1 NIL 22) 4.23) prints |004.2|.

PRINTOUT

Finally, any reasonably complex output will involve lots of PRIN1s, TERPRIs, PRINTNUMs, etc. in very complex combinations. The PRINTOUT CLISP statement provides a simpler (sometimes) interface to all of this.

PRINTOUT is very complex and we will cover on a small bit of it here. See section 6.5 of the IRM for more details.

PRINTOUT has the format:

(PRINTOUT *Stream Command1 Command2 ...*)

Stream references an open file.

Each *CommandI* is either one of the printing commands discussed below or an arbitrary Lisp SExpression.

PRINTOUT iterates through the *CommandIs*. If it encounters a command, it carries out that command. Otherwise, it prints the SExpression using PRIN1.

Note that numbers are PRINTOUT commands. Therefore the only way to print numbers is as the result of a command!

Some of the commands are:

.SP *N* ž print *N* spaces

T ž print a carriage return

.SKIP *N* ž skip *N* lines, i.e., print *N* returns

.PAGE ž print a form feed character

.FONT *FontSpec* ž change the font for printing to the file. *FontSpec* is a font list like (TIMESROMAN 12 BOLD).

.P2 *SExpr* ž PRIN2s *SExpr*

.FR *ŷN SExpr* ž PRIN1s *SExpr* right-flushed in the next *N* character positions.

.CENTER *ŷN SExpr* ž PRIN1s *SExpr* centered in the next *N* character positions.

.IFormat *Number* žprints *Number* as an integer using *Format* as a format spec. *Format* is like the FIX format in

PRINTNUM, except that rather than parts of a list, you use atoms separated by periods.

Example: `.I5.NIL.T` is the same as `(FIX 5 NIL T)` in PRINTNUM.

.FFormat Number prints *Number* as a real number using *Format* as a format spec. *Format* is like the FLOAT format in PRINTNUM, except that rather than parts of a list, you use atoms separated by periods.

Example: `.F5.2` is the same as `(FLOAT 5 2)` in PRINTNUM.

SExpr evaluates *SExpr* for side-effect only. Nothing is printed except as a result of the evaluation. Used to tailor your own PRINTOUT commands.

`# 'XXX` will print nothing.

`# (PRINI 'XXX Stream)` will print XXX on the file.

Examples:

```
(PRINTOUT Stream "This is Line 1" T "This is Line 2" T)
```

prints

This is Line 1

This is Line 2

```
(PRINTOUT Stream .I5 445 .SP 6 .F3.2 0.3456 T)
```

prints

| 445 .34|

Miscellaneous Considerations

The File T

There is one special input file in the system known as T. T is both an input and an output file. On the input end, it is the keyboard. On the output end, it is the Lisp Exec window.

You can use T as the Stream argument to any of the Lisp input/output functions discussed above.

Example:

(PRINT "Hello" T) prints Hello in the Exec window.

(READ T) reads the next SExpression the user types in.

When reading from T, the functions READ, RATOM, READC, etc. all wait for the user to type something before returning.

Line Buffering When Reading Keyboard Input

The file T and all other input that comes from the keyboard is subject to line buffering, i.e., the user type-in is held in a buffer until a carriage return is typed.

Until the return is typed, the user input is not available to be read by a program.

Once the return is typed, the user type-in can be read by a program.

Line buffering can be turned off using the function CONTROL:

(CONTROL T) ž turns the line-buffering off.

(CONTROL NIL) ž turns the line buffering on.

With line-buffering off, input can be read by the program immediately after it is typed in.

Example: If you want to do something immediately after the user types the next character, you should turn line buffering off and wait for the character press using READC or BIN as follows:

```
(DEFINEQ (WaitForChar
  (LAMBDA NIL
    (* * Turn off buffering)
    (CONTROL T)
    (PROG1
      (READC T)(* * wait for a character press)
```

(CONTROL NIL) (* * Turn on buffering
again))))))

Default Input/Output File

Almost all of the Lisp input/output functions will accept NIL as their *Stream* argument. In this case they will use the current default input or output file.

Initially, the default input and output is set to T, i.e., the default input file is the keyboard and the default output file in the Lisp Exec window.

The following functions can be used to change the default files:

(INPUT *Stream*) ž makes the open file referenced by *Stream* be the default input file. Returns the old default input file. If *Stream* is NIL, just returns the default input file without changing it.

(OUTPUT *Stream*) ž makes the open file referenced by *Stream* be the default output file. Returns the old default output file. If *Stream* is NIL, just returns the default output file without changing it.

References

Input/Output is covered in Chapter 6 and Section 18.16 of the IRM.

Beware, however, this documentation is somewhat out of date. In particular, streams are not covered. Instead, files are said to be referenced by full file name, i.e., in the old manner.

LispCourse #39: Solutions to Homework #38

Solution functions are attached.

Notes on Problem C

Problem C requires a *breadth-first search* of the family tree. In particular, starting at the node for the given person, you want to check all one-away relatives (e.g., parents and children). If none of these match, you want to check all two-away relatives (i.e., all one-away relatives of the one-away relatives). And so on until you find an N-away relative that matches or you are out of relatives.

This contrasts with a *depth-first search*, where you would search all of the person's mothers relatives first, then his or her father's relatives, then his or her children's relative (where children are considered in some arbitrary order). For each of the mother's relatives, you would first search that person's mother's relatives, then their father's and so on. Basically, you search one branch of the family tree until it ends, then search the next branch and so on.

In a *breadth-first search*, at each step you have a set of N-away relatives, none of whom match the thing you are searching for. For each of these N-away relatives, you want to compute their 1-away relatives (i.e., the N+1-away relatives). While you are computing the N+1-away relatives for each of the N-away relatives, you need to maintain lots of information – in particular, you need to keep a list of the N-away relatives that still have to be worked on AND you need to maintain a list of the N+1-away relatives already discovered (who will be "expanded" during the next round if no match is found among the N+1 relatives).

The best way to maintain this information is in a global queue (see Homework #32 & LispCourse #33).

Just keep repeating the following until the queue becomes empty:

Take a person off of the head of the queue.

If he matches what you're looking for, you're done – exit.

If there's no match, expand the person into a set of his 1-away relatives and add all these relatives to the end of the queue.

Note that with this scheme, you don't need to keep explicit track of what level (i.e., K-away or L-away, or whatnot) you are working on. Because you always add the N+1-away to the end of the queue but take the next person to "expand" from the front of the queue, you are certain to process all the N-away relatives before you start on the N+1 away relatives. The moment you find a match, you just stop and you can be sure that you've found the closest relative that matches.

However, if there are more than one matching relative at the Nth level, you will find only one and then stop. To find all closest relatives in case of a tie requires some small variations on this scheme.

LispCourse #40: Using the Display: Bitmaps, DisplayStreams, & Windows

Introduction

Interaction with the display from Interlisp-D programs involves 3 basic types of Interlisp objects – *bitmaps*, *display streams*, and *windows*.

A *bitmap* is a two-dimensional array of bits (1s and 0s) in your computer's memory.

You deal with bitmaps by setting specific bits to 1 or 0, or by copying rectangular arrays of bits around from bitmap to bitmap.

The Interlisp-D display screen is just a special bitmap (called the SCREENBITMAP) that is 808 bits high and 1024 bits wide that is displayed on the screen with every 1-bit is black and every 0-bit is white (or vice versa if you change the VIDEOCOLOR parameter as per page 19.7 in the IRM).

Display streams represent an interface that allows you to deal with bit maps at a level higher than bits, i.e., in terms of characters, fonts, lines, circles, regions, etc.

You can call functions like DRAWCURVE on a display stream. The result will be a curve drawn on the destination bitmap of that display stream.

A display stream is a datatype that represents some destination bitmap. Stored in this datatype are fields that contain things like the current font to be used for writing characters to the destination bitmap, the X-Y position of the "cursor" in the bitmap, left and right margins for writing and drawing on the bitmap, etc.

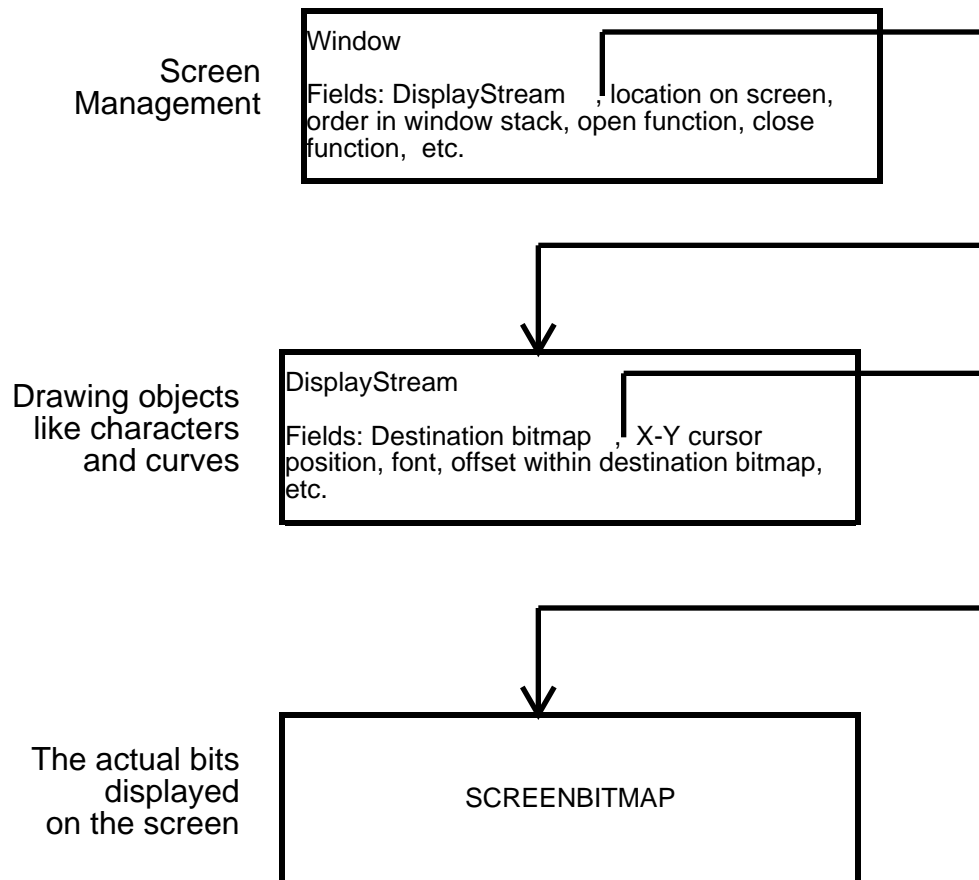
Windows are a way of managing what is being displayed on the most important bitmap of all, the Interlisp-D display screen.

A window is a datatype that represents the window object that can be displayed on the Interlisp screen.

Stored in the fields of the window datatype are all kinds of functions that specify what is to be done when various operations are carried on the window, e.g., when the window is opened, closed, shrunk, or moved.

Also stored in each window datatype is a display stream through which characters, lines, curves, etc. are drawn on the window's bitmap (i.e., the display stream's destination bitmap) which is always SCREENBITMAP.

In summary, the data structures underlying each window you see on the screen are as follows:



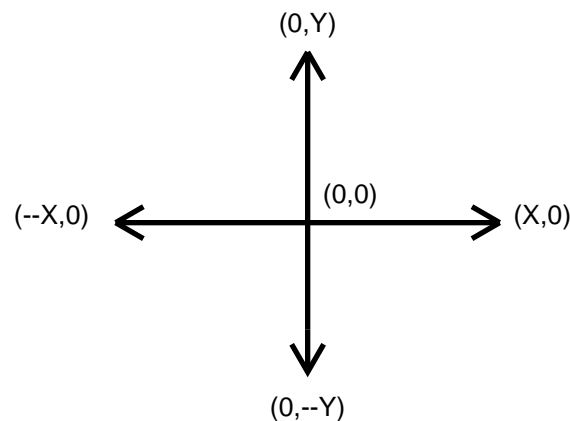
Coordinate Systems, Positions, & Regions

Coordinate Systems for Specifying Locations in Bitmaps, et al.

Each bitmap, display stream, and window has its own coordinate system used to specify locations within the object.

When dealing with the display, these coordinate systems are always measured in bits (or screen units or the area that it takes to display 1 bit on the screen or 1/72nd of an inch).

For all three objects, the coordinate system is a standard Cartesian system in the standard orientation.



Bitmaps have a finite size. Thus, for bitmaps the origin of the coordinate system is placed at the lower-left corner of the bitmap and only the upper-right quadrant (positive X and Y) is used to specify locations in the bitmap.

Display streams and windows are considered to look onto an infinite plane. Thus the origin is arbitrarily placed (see below) and the entire coordinate system is used.

The coordinate system for a window is the same as the coordinate system for its underlying display stream. The coordinate system for a display stream is mapped onto the coordinate system for its destination bitmap using X and Y translation parameters as discussed below.

Positions and Regions

Positions and regions are data structures that represent X-Y locations and rectangles, respectively, in an arbitrary coordinate system.

A *POSITION* is a record with two fields, XCOORD and YCOORD. Most functions that take an X-Y location as an argument require a POSITION record.

To create a POSITION record:

(create POSITION XCOORD _ X YCOORD _ Y)

A *REGION* is a record with four fields: LEFT, BOTTOM, WIDTH, and HEIGHT specifying the lower-left corner and extent of a rectangular region in some coordinate space.

To create a REGION record:

(CREATEREGION *Left Bottom Width Height*)

There are several functions available to manipulate positions and regions, including the following:

(INSIDEP *Region Position*) ž returns T if *Position* is inside *Region*.

Examples:

1_ (INSIDEP (CREATEREGION 100 100 10 10)(create POSITION XCOORD _ 150 YCOORD _ 100))

NIL

2_ (INSIDEP (CREATEREGION 100 100 100 10)(create POSITION XCOORD _ 150 YCOORD _ 100))

T

(INTERSECTREGIONS *Region1 Region2 ... RegionN*) ž returns the region that is the intersections of *Region1*, *Region2*, ..., and *RegionN*. *NIL*, if there is no intersection.

(UNIONREGIONS *Region1 Region2 ... RegionN*) ž returns the region that is the union of *Region1*, *Region2*, ..., and *RegionN*. The union is the smallest (rectangular) region that contains all of the given regions.

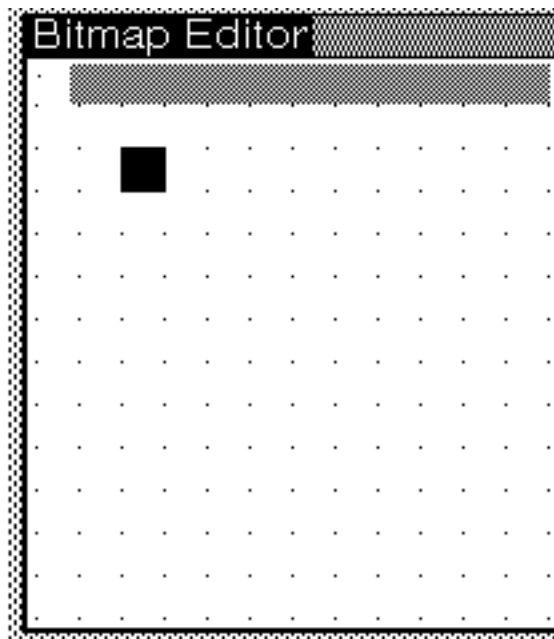
BITMAPS

Introduction

A bitmap is datatype that represents an N by M array of bits in memory.

The bits in a bitmap are identified using a positive integer coordinate system whose origin (0,0) is the lower-left corner of the bitmap.

For example, (10,2) represents the bit that is 10 to the left of and 2 up from the bit in the lower-left corner of the bitmap.



Creating Bitmaps

To create a bitmap use the following function:

(BITMAPCREATE *Width Height*) ž creates and returns a bitmap *Height* bits high and *Width* bits wide.

BITBLT

The major operation on bitmaps is the moving of bits from one bit map to another using the BITBLT function:

(BITBLT *SourceBitMap SourceLeft SourceBottom DestBitMap DestLeft DestBottom Width Height SourceType Operation Texture*) ž copies some bits in *SourceBitMap* and combines them with some bits in *DestBitMap*, resulting in a change to these bits in *DestBitMap*.

The bits copied from *SourceBitMap* are those in the region defined by *SourceLeft*, *SourceBottom*, *Width*, & *Height*.

The bits effected in the *DestBitMap* are those in the region defined by *DestLeft*, *DestBottom*, *Width*, & *Height*.

If either of these regions overflows the edges of its bitmap, then *Width* and/or *Height* are decreased until both regions fit into their bitmaps.

The way in which the bits are copied from the *SourceBitMap* is determined by *SourceType* and *Texture* as follows:

If *SourceType* is INPUT, then the bits are copied directly from the region in *SourceBitMap*.

If *SourceType* is INVERT, then the bits are copied from the region in *SourceBitMap*, but each bit is inverted (i.e., 1s become 0s and vice versa).

If *SourceType* is TEXTURE, then *SourceBitMap*, *SourceLeft*, and *SourceBottom* are ignored and the bits to be copied are taken from the bitmap specified by *Texture*.

If the *Texture* bitmap is smaller than *Width* by *Height*, then it is repeated as many times as necessary to make a rectangle of bits that is of size *Width* by *Height*.

Note: the global variables **WHITESHAE**, **BLACKSHAE** and **GRAYSHAE** are small bitmaps for white, black, and gray, respectively.

SourceType defaults to INPUT.

The way in which the copied bits are combined with the bits already in *DestBitMap* is determined by *Operation* as follows:

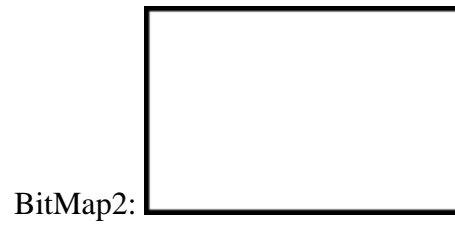
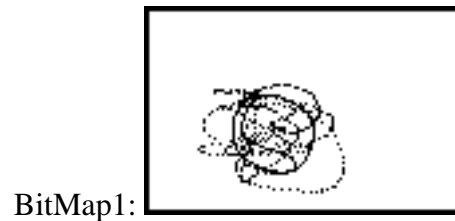
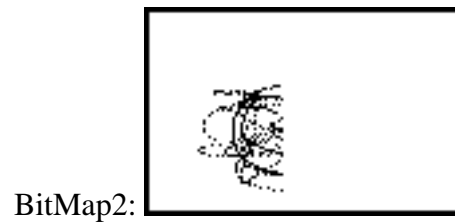
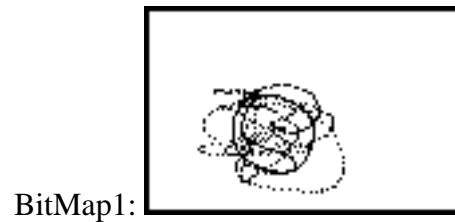
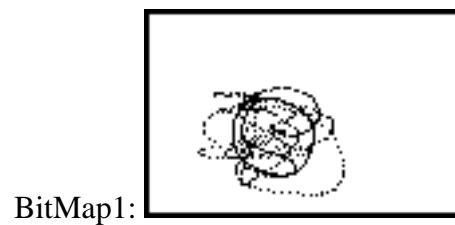
If *Operation* is REPLACE, the bits in *DestBitMap* are replaced by the copied bits.

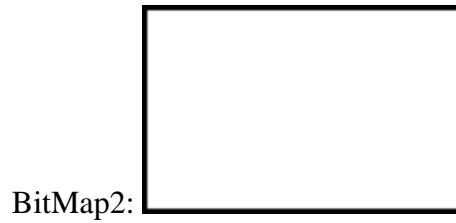
If *Operation* is PAINT, the bits in *DestBitMap* are logically ORed with the copied bits.

If *Operation* is INVERT, the bits in *DestBitMap* are logically XORed with the copied bits.

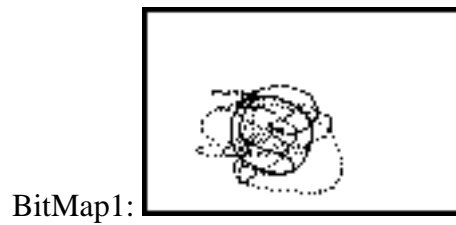
If *Operation* is ERASE, the bits in *DestBitMap* are logically ANDed with the inversion of the copied bits.

Operation defaults to REPLACE.

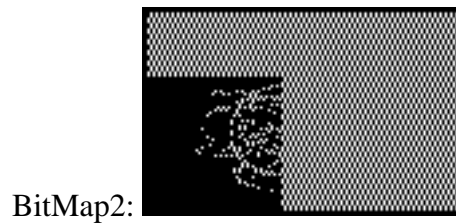
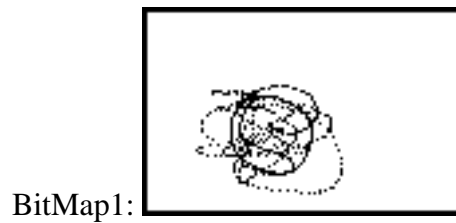
Examples:**START****(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 'INPUT 'REPLACE)****(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 'INPUT 'ERASE)**



**(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 'INVERT
'PAINT)**

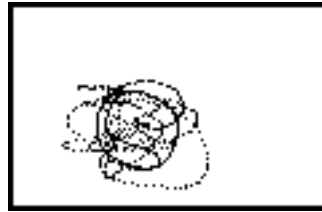


**(BITBLT BitMap1 0 0 BitMap2 0 0 125 175
'TEXTURE 'PAINT GRAYSHADE)**

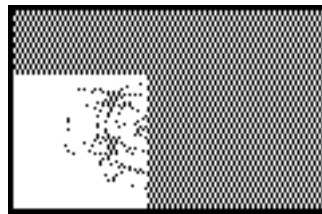


**(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 'INVERT
'ERASE)**

BitMap1:



BitMap2:



Miscellaneous Bitmap Functions

The following are other miscellaneous functions that operate on bitmaps:

(BITMAPBIT *BitMap* *X* *Y* *NewValue*) ž If *NewValue* is either 0 or 1, then sets the value of the (*X*,*Y*)th bit in *BitMap* to have value *NewValue* and returns its old value (either 0 or 1). If *NewValue* is NIL, just returns the value of the (*X*,*Y*)th bit in *BitMap*. If *NewValue* is anything else, then its an error.

(BITMAPCOPY *BitMap*) ž returns a new bitmap that is an exact copy of *BitMap*.

(BITMAPHEIGHT *BitMap*) ž returns the height in bits of *BitMap*.

(BITMAPWIDTH *BitMap*) ž returns the width in bits of *BitMap*.

The following functions return pointers to the "special" bitmaps:

(SCREENBITMAP) ž returns the screen bitmap.

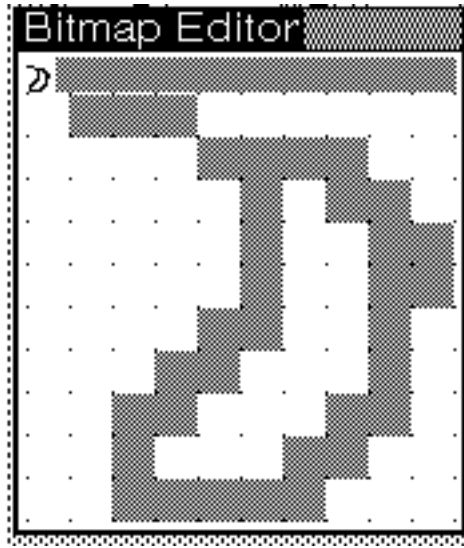
(CURSORBITMAP) ž returns the bitmap that is the cursor.

Hand Editing Bitmaps

(EDITBM *BitMap*) ž calls an editor to edit *BitMap*. If *BitMap* is NIL, will create a bitmap after asking for the height and width of the desired bitmap.

The bitmap editor runs in a window which you will be asked to place.

The window is shown below.



In the bar just below the title bar flush to the left side of the window, the bitmap being edited is displayed at normal resolution.

To the right of this is a gray area Ÿ clicking the MIDDLE mouse button in this gray area will bring up a menu of commands, including "OK" which will allow you to exit the editor.

The gridded area represents a portion of the bitmap blown up Ÿ each square corresponds to one bit.

Clicking the RIGHT mouse button in a square makes the corresponding bit black.

Clicking the MIDDLE mouse button in a square makes the corresponding bit white.

To change the portion of the bitmap being displayed in this blown-up area, LEFT or MIDDLE click in the normal resolution bitmap above and choose the "Move" option (i.e., the only option) from the menu that appears.

Notes

Most of the above functions can take either a display stream or a window in place of a bitmap argument. In this case, the operation is carried out on the bitmap associated with the given display stream or window.

Thus, BITBLT et al. can be used to manipulate the contents of a window. (See below).

Display Streams

Display streams are a special type of Stream (see LispCourse #38, page 3) designed to provide an interface to bitmaps (most importantly the display screen bitmap).

Display streams allow you to access bitmaps in higher-level terms than just BITBLT and BITMAPBIT. In particular, display streams are designed to deal with objects like characters and geometric objects (lines, curves, circles, etc.).

Display Stream Properties (& creating a display stream)

Basically, a display stream has a destination bitmap and set of properties that specify how higher-level objects like characters and lines should be drawn on this bitmap.

The most important properties of a display stream are the following:

Destination ž the bitmap that this display stream refers to. Initializes to (SCREENBITMAP).

XOffset, YOffset ž the origin of the display stream's coordinate system expressed in terms of the destination bitmap's coordinate system. Initializes to 0 and 0, so that the display stream origin is at the lower-left corner of the destination bitmap.

ClippingRegion ž a region *in the display stream's coordinate system* that limits where anything can be written or drawn to the display stream. If a display stream has a clipping region, then commands that request characters and/or lines to be drawn outside of this clipping region will simply be ignored.

XPosition, YPosition ž the X and Y coordinates of the "current" position of the display stream. The "current" position is an invisible cursor that determines where things will be drawn unless otherwise specified. (Initially 0 and 0, i.e., at the display stream's origin.)

Texture ž the background pattern used in the display stream (see Texture description under BITBLT above). Initializes to WHITESHADE.

Font ž a font descriptor that specifies the font to be used for printing characters on this display stream. Initializes to font descriptor for Gacha 10. (See LispCourse #21 for a discussion of fonts.)

Operation ž the BITBLT operation (see BITBLT above) used by default when printing or drawing on the destination bitmap. Must be one of REPLACE, PAINT, INVERT, or ERASE as above. Initializes to REPLACE.

For other properties of a display stream, see section 19.9.1 of the IRM..

To create a display stream, use the DSPCREATE function:

(DSPCREATE *Destination*) ž returns a display stream object using *Destination* as its destination bitmap. If *Destination* is NIL, then (SCREENBITMAP) is used. All of the properties of the display stream are initialized as described above.

The properties of a display stream can be manipulated using a set of functions that work as follows:

Each property is manipulated by a selector/mutator whose name is "DSP" followed by the property name in all caps.

For example: the Font property is accessed using the function **DSPFONT**.

Each of these functions takes two arguments: the new value of the property and the display stream.

If the new value is NIL, then the function just returns the current value of the property.

If the new value is a value, then the function returns the old value and sets the property to the new value.

Example:

(DSPFONT NIL DS) returns the Font property of display stream DS.

(DSPFONT (FONTCREATE 'TIMESROMAN 16) DS) changes the Font property of DS to be TimesRoman 16 and returns the old font of DS.

(DSPTEXTURE GRAYSHADE DS) changes the background texture of DS to be gray and returns the old background texture.

Moving the Current Position in the Display Stream

When printing or drawing to a display stream, the default is to print/draw at the current position, i.e., at (XPosition, YPosition) in the display stream.

DSPXPOSITION and DSPYPOSITION can be used to independently change the X and Y coordinates of the current position.

The following functions can also be used to change the current position:

(MOVETO X Y DisplayStream) ž moves *DisplayStream*'s current position to point (X,Y).'

(RELMOVETO DeltaX DeltaY DisplayStream) ž moves *DisplayStream*'s current position to a point *DeltaX* units to the right and *DeltaY* units up from its previous position.

Printing and Drawing on Display Streams

To print characters on a display stream, use the standard printing routines discussed in LispCourse #38 (starting on page 12).

For example: (PRIN1 "ABC" DS) will print ABC on display stream DS. It will do the printing starting at the DS's current point and then will move the current point to the end of the last character printed.

The font used for printing the characters is determined by DS's font property.

To draw straight lines on a display stream use the following functions:

(DRAWTO *X Y Width Operation DisplayStream*) ž draws a line on *DisplayStream* from the current point to location (X,Y). The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*'s operation property). At the end, *DisplayStream*'s current point is set to (X,Y).

(RELDRAWTO *DeltaX DeltaY Width Operation DisplayStream*) ž draws a line on *DisplayStream* from the current point to the location *DeltaX* to the right and *DeltaY* up. The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*'s operation property). At the end, *DisplayStream*'s current point is set to the end of the line.

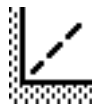
(DRAWBETWEEN *Position1 Position2 Width Operation DisplayStream*) ž draws a line on *DisplayStream* from *Position1* to *Position2*. The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*'s operation property). At the end, *DisplayStream*'s current point is set to *Position2*.

To draw curved lines on a display stream use the following functions.

These functions all take a Brush argument. The Brush argument is a two item list containing the *shape* and the *width* of the "brush" that will be used to draw the curve. *Shape* is one of: ROUND, SQUARE, VERTICAL, DIAGONAL. *Width* is the thickness of the line to vbe drawn in bits.

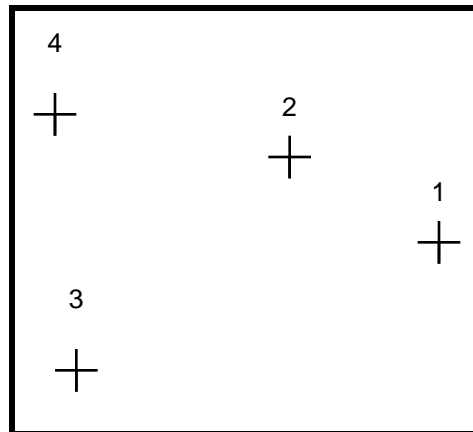
These functions also take a Dashing argument that determines how to dash the line. If Dashing is NIL, no dashing will be done. Otherwise, Dashing is a list containing an even number of positive integers. The first integer specifies how long (in bits) the brush should be "on", the second integer then specifies how long the brush should be "off", the third integer specifies how long the brush should be "on" again, etc.

Example: A Dashing of (5 2) specifies that the line should have 5 bits on then 2 bits off, then 5 bits on, etc.

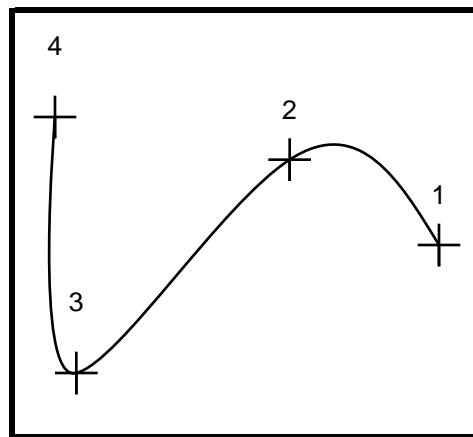


(DRAWCURVE *Knots ClosedFlg Brush Dashing DisplayStream*) ž
Knots is an ordered list of POSITIONs. DRAWCURVE draws a spline curve on *DisplayStream* that is fit to these POSITIONs. If *ClosedFlg* is NIL, the spline will be an open curve; otherwise it will be a closed curve. *Brush* and *Dashing* are as described above. The current position is left unchanged.

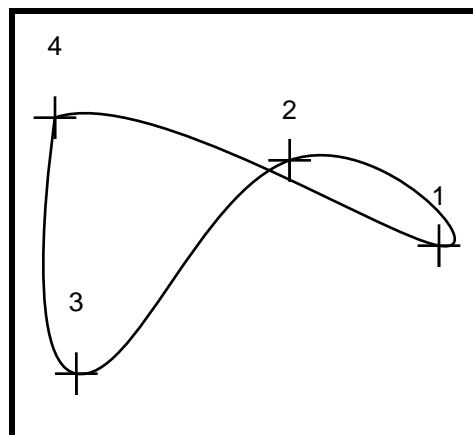
The knots:



An open curve using drawn using these knots:



A closed curve using drawn using these knots:



(DRAWCIRCLE *X Y Radius Brush Dashing DisplayStream*) ž Draws a circle on *DisplayStream* centered at (X,Y) and having radius *Radius*. The current position is left at (X,Y) .

(DRAWELLIPSE *X Y MinorRadius MajorRadius Orientation Brush Dashing DisplayStream*) ž Draws an ellipse on *DisplayStream* centered at (X,Y) and having a minor radius *MinorRadius* and a major radius *MajorRadius*. The orientation of the *MajorRadius* is determined by *Orientation*, which is in degrees from upright in the counterclockwise direction. The current position is left at (X,Y) .

Windows & the Window Package

The window package provides two basic services to Interlisp-D programs:

- 1) It manages the "space" on the Interlisp-D display screen, allowing multiple programs using multiple display streams to all access the screen simultaneously without interfering with each other.
- 2) It provides an interface that dispatches the user's mouse "actions" to the Interlisp-D programs that these actions were intended to effect.

We will discuss only the first function in this section. Mouse action dispatching will be covered in the section on the Mouse below.

Space Management: The Window Stack

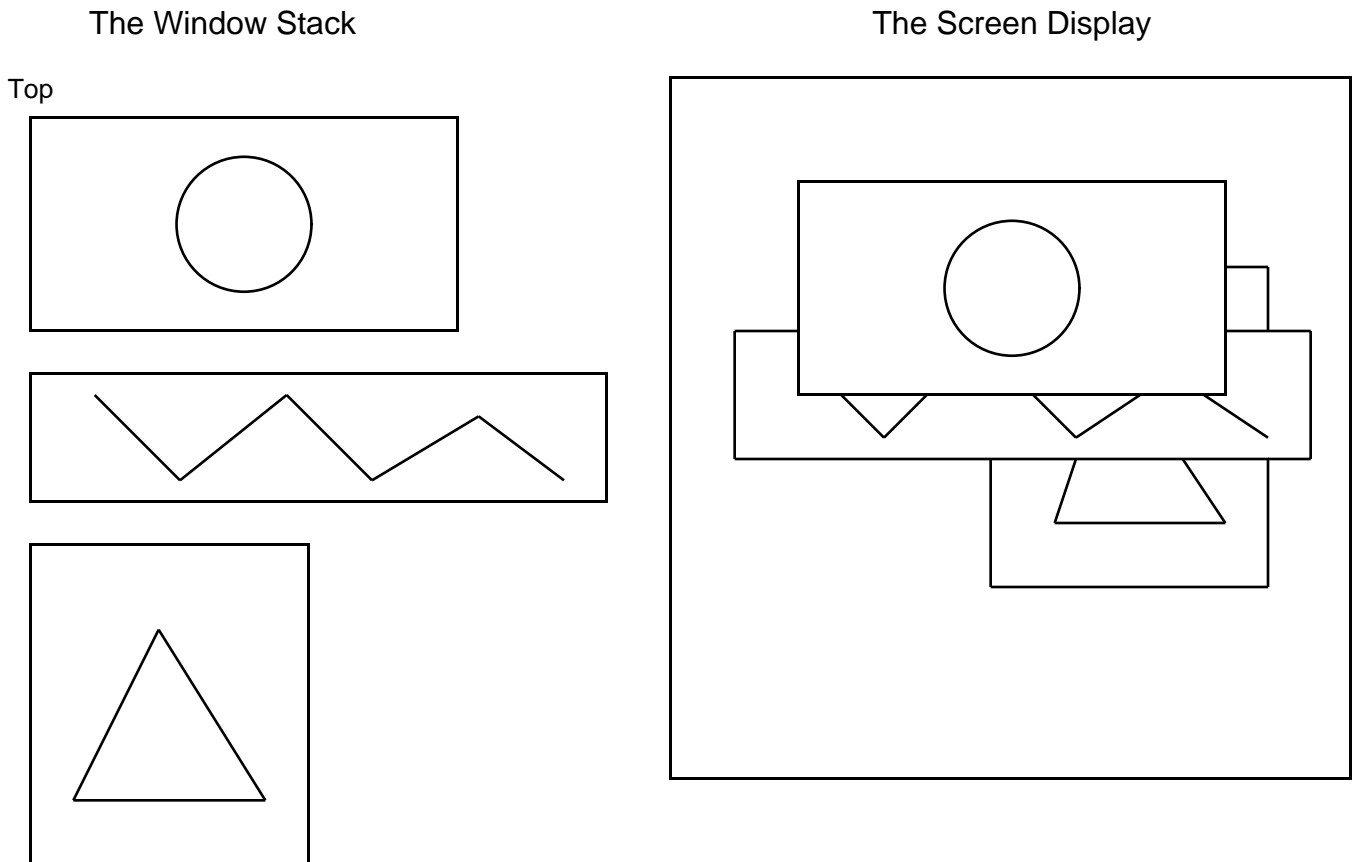
Space on the Interlisp-D screen is managed using an occlusion stack of open windows.

Each open window represents a rectangular region of the screen in which a display stream (or part of a display stream) is to be displayed.

All of the open windows are placed on a stack (i.e., an ordered list), ordered by "depth".

If the display regions of two or more windows intersect, then the actual screen display in the intersection region will reflect the contents of the window nearest to the top of the stack (i.e., earlier in the ordered list).

The windows that are lower (deeper, later) in the stack will have the part of them corresponding to the intersection region "occluded" by the windows higher in the stack.



The window at the top of the stack is always visible in its entirety on the screen.

Windows later in the stack may also be entirely visible (if they do not intersect with any other open windows), but this cannot be guaranteed.

Any operation on a window brings it to the top of the stack. Thus all operations are carried out on windows that are entirely visible on the screen.

The function call (**OPENWINDOWS**) returns the window stack as an ordered list.

Windows

A window is just a data structure that contains all of the information necessary for the window package to display the window on the screen.

Open and Closed Windows

Windows can be *open* or *closed*.

A *closed* window is just a data structure and is not part of the window stack. It therefore cannot be displayed on the screen.

An *open* window is on the window stack and is therefore displayed on the screen providing that it is not completely occluded by other open windows on the stack.

Whenever you operate on a window (e.g., draw in the window), the window is opened (and brought to the top of the stack).

(OPENW *Window*) opens the closed window *Window*, placing it at the top of the window stack.

(CLOSEW *Window*) closes the open window *Window*.

(OPENWP *Window*) returns *Window* if it is an open window; NIL otherwise.

Creating Windows

To create a window use:

(CREATEW *Region Title Border NoOpenFlg*) ž creates and returns a new window.

The created window will be displayed (if opened) in screen region *Region*. If *Region* is NIL, the the user will be asked to specify a region on the screen.

If *Title* is non-NIL, a title bar will be created at the top of the window and *Title* will be printed left-flush in this title bar.

If *Border* is a number it is the number of bits to use as a border around the edge of the window; otherwise the window will have a border width of 4.

If *NoOpenFlg* is NIL, the created window will be opened and placed at the top of the window stack. Otherwise, the created window will not be opened.

(WINDOWP *Window*) ž returns *Window* if it is a window, NIL otherwise.

Windows and Display Streams

When a window is created, a corresponding display stream is created and stored in the window data structure.

The function: **(WINDOWPROP *Window* 'DSP)** will retrieve the display stream associated with *Window*. (See explanation of WINDOWPROP below.)

This destination of a window's display stream is always (SCREENBITMAP). Thus, a program should never alter the Destination property of a window's display stream.

The window package automatically takes care of setting the XOffset, YOffset and ClippingRegion properties for the window's display stream as the window is scrolled or moved about the screen.

A program should never alter the XOffset, YOffset, and ClippingRegion properties of a window's display stream.

Otherwise, all properties and operations applicable to display streams are also applicable to windows.

For example, to change the font used to print in a window, you use DSPFONT. It is not necessary to specify the window's display stream directly since all the DSPxxx functions know how to coerce a window into its corresponding display stream.

(DSPFONT BigFont Window) has the same effect as
(DSPFONT BigFont (WINDOWPROP Window 'DSP))

Printing and Drawing in Windows

Printing and drawing in a windows is identical to printing and drawing in display streams – use the print functions and the *DSPxxx* functions described above.

Just use the window as the argument in place of a display stream.

Doing Things to Windows

The following functions carry out various operations on windows.

(MOVEW *Window* *Position*) – moves *Window* so that its lower-left corner is located at POSITION *Position* on the display screen. If *Position* is NIL, then the user is asked for to specify a position.

If *Window* is closed, it will not be opened unless *Position* is NIL.

(SHAPEW *Window* *NewRegion*) – reshapes and moves *Window* so that it is displayed in screen region *NewRegion*. If *NewRegion* is NIL, the user is asked to specify a new region. Opens *Window* if it is closed.

(TOTOPW *Window*) – places *Window* at the top of the window stack and adjusts the display accordingly. *Window* is opened if it was closed.

(BURYW *Window*) – places *Window* at the bottom of the window stack and adjusts the display accordingly.

(CLEARW *Window*) – clears the window, filling it with the Texture of its underlying display stream. Also sets the current position to the upper-left corner of the window.

(REDISPLAYW *Window*) – causes *Window* to be redisplayed. Used if, for example, the window's display gets messed up. For this to work correctly the window must have a REPAINTFN property (see below).

(SHRINKW *Window*) – closes *Window* and replaces it on the screen with a smaller icon window. The icon window is actually another window that is associated with the window being shrunk.

Unless the window's properties specify otherwise (see below), the icon window is just a black bar containing *Window*'s title or the date and time if *Window* has no title.

You cannot shrink an icon window.

SHRINKW is a no-op if *Window* is closed.

(EXPANDW *Window*) ž expands or "unshrinks" *Window* if it was previously "shrunk". If *Window* is an icon window, expands the window that the icon window represents. If *Window* is neither shrunk nor an icon window, EXPANDW is a no-op.

Tailoring Windows to Special Applications: Window Properties

Every window has a large set of properties that determine how it looks and, more importantly, how it behaves in various circumstances.

You needn't set any of these properties, they will all default to something reasonable if you just want an ordinary window.

But if you want a fancy window, you need to set one or more of these properties to get the looks or behavior you are interested in.

Most of the properties are intended to have functions or lists of functions as values. These functions are called (with the window as an argument) whenever some operation is done on the window (e.g., the window is closed) or some event occurs in the window's environment (e.g., the user clicks a mouse button in the window).

In addition to these functional properties, each window has a few properties that are not functions, e.g., the window's title, and a few that are read-only (i.e., that the window package sets and that the user can query but not change).

Looks Properties

TITLE ž a title to be printed left-flush in the title bar at the top of the window. If TITLE is NIL, the window will have no title bar.

If **TITLE** is changed from **NIL** to a value, then a title bar is created resulting in a slight enlargement of the window. Defaults to **NIL**.

BORDER ž an integer indicating the width of the border around the edge of the window. The border will have (at most) 2 bits of white around the inside with the remaining border width being black; subject to the constraint that the white bits never make up more than half the border width. Defaults to 4.

WINDOWTITLESHADE ž a texture (see texture under **BITBLT** above) that is used as the background in the title bar to the right of the end of the title. Where the title is printed, the background is black. Defaults to value of global variable **WINDOWTITLESHADE**, which defaults to **BLACKSHADE**.

Read-only Properties

DSP ž the window's display stream.

HEIGHT, WIDTH ž the height and width in bits of the interior of the window (i.e., not including the border and title bar). The interior part of the window is the user accessible portion of the window Ÿ you cannot draw or print directly on the title bar or border.

REGION ž a **REGION** that describes (in the **SCREENBITMAP** coordinate system) the region occupied by the entire window (title bar and all) on the screen.

Processes Property

PROCESS ž the process that is associated with this window. This is the process that will become the **TTY** process when **GIVE,TTY.PROCESS** is called using this window as an argument.

By default (see **WINDOWENTRYFN**) this is the process that becomes the **TTY** process when you button down inside the window.

Functions to Be Invoked by Operations on the Window

CLOSEFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just before the window is closed. If any function is the atom DON'T or returns the atom DON'T, then the window will not be closed.

Warning: You cannot call CLOSER inside of a CLOSEFN function otherwise you will get infinite recursion.

OPENFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just after the window is opened. If the atom DON'T appears anywhere on the OPENFN list, then the window will not be opened and the OPENFNs will not be called.

TOTOPFN ž a single function that will be called (with the window as the only argument) whenever the window is brought to the top of the window stack.

MOVEFN ž a single function or a list of functions that will be called (with the window and the new position of the lower-left corner of the window as arguments), in order, just **before** the window is moved. If any function is the atom DON'T or returns the atom DON'T, then the window will not be moved. If any function returns a POSITION record, then the window will be moved to that position.

AFTERMOVEFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just **after** the window is moved.

REPAINTFN ž a single function or a list of functions that will be called (with the window and the region of the window to be repainted as arguments), in order, whenever the window is redisplayed (i.e., during scrolling or when REDISPLAYW is called). The function should redraw the contents of the specified region in window.

Warning: You cannot call CLEARW inside of a REPAINTFN. Use DSPFILL instead.

Functions to Be Invoked by Mouse Events in the Window

BUTTONEVENTFN ž a single function that will be called (with the window as the only argument) whenever there is a change in state of the mouse buttons (a mouse button goes up or down) while the cursor is inside the window and the window is associated with the TTY process.

While the BUTTONEVENTFN is being processed, further mouse events do not reinvoked the BUTTONEVENTFN.

As a general convention a BUTTONEVENTFN is called when a mouse button goes down but does not do its work until the mouse button goes up. This convention is accomplished by writing the "correct" kind of BUTTONEVENTFNS - i.e., functions that wait until the mouse button is up before continuing their work.

RIGHTBUTTONFN ž a single function that will be called (with the window as the only argument) instead of the BUTTONEVENTFN whenever only the RIGHT mouse button goes down in the window.

If RIGHT mouse clicks are to be treated the same as other mouse clicks, then just make the BUTTONEVENTFN and the RIGHTBUTTONFN be the same function.

Defaults to DOWINDOWCOM, which brings up a menu of the standard window operations.

WINDOWENTRYFN ž a single function that will be called (with the window as the only argument) whenever a button goes down in the window and the process associated with the window is not the TTY process.

Default is to call GIVE.TTY.PROCESS on the window and then call the window's BUTTONEVENTFN.

CURSORMOVEDFN, **CURSROUTFN**, **CURSROUTFN** ž a single function that will be called (with the window as the only argument) whenever the cursor moves into (out of, about inside of) the window.

Properties that Support Icons and Shrinking

SHRINKFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just before the window is shrunk. If any function is the atom DON'T or returns the atom DON'T or if any of the CLOSEFNs is DON'T or returns DON'T, then the window will not be shrunk.

ICONFN ž a single function that will be called (with the window and the previous icon, if any), just before the window is shrunk. The ICONFN should return a bitmap or a window that will be used as the basis for making the icon window.

EXPANDFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just after the window is expanded from the shrunk state. If any function is the atom DON'T, then the window will not be shrunk and the rest of the EXPANDFNs will not be called.

Manipulating a Window's Properties: WINDOWPROP, et al.

The function WINDOWPROP is a selector/mutator that can be used to manipulate properties of a window:

(WINDOWPROP *Window Property NewValue*) ž sets the *Property* property of *Window* to be *NewValue*. Returns the old value of the specified property.

WINDOWPROP is a LAMBDA/No-spread function. If *NewValue* is omitted from the function call, then

WINDOWPROP will simply return the old value of the specified property. (Note: Omitting *NewValue* is not the same as specifying NIL as the *NewValue*. The former will not change the specified property, the later will set the property's value to NIL.)

Important note: WINDOWPROP can be used to add any arbitrarily named property to a window Ÿ the *Property* argument need not be one of the properties supported by the window package (and described above). Thus WINDOWPROP can be used to cache any sort of information on the window that may be useful to the program. See the scrolling window example below.

The functions WINDOWADDPROP and WINDOWDELPROP can be used to add and remove items for properties whose values are lists (e.g., lists of functions as for the CLOSEFN property):

(WINDOWADDPROP *Window Property ValueToBeAdded*) ž adds *ValueToBeAdded* to the list that is the value of the *Property* property of *Window*. If *ValueToBeAdded* is already on that list it is not added again. If the current value of the specified property is not already a list, it is made into a list before *ValueToBeAdded* is added. *ValueToBeAdded* is always placed at the end of the list. Returns the old value of the specified property.

(WINDOWDELPROP *Window Property ValueToBeRemoved*) ž removes *ValueToBeRemoved* from the list that is the value of the *Property* property of *Window*. If *ValueToBeRemoved* was actually on that list, WINDOWDELPROP returns the old value of the specified property. Otherwise, WINDOWDELPROP returns NIL.

Making Scrollable Windows

The window package provides some support for making scrolling windows.

In particular, the window package provides and manages the scroll bars that pop up whenever the mouse rolls through the left edge of the window.

The window package also supports a default scheme for doing the actual scrolling of the window contents Ÿ although the default scheme requires some amount of programming setup.

The SCROLLFN Property

Every window has a SCROLLFN property.

If the SCROLLFN property is NIL (the default), then the window will not be scrollable.

If the SCROLLFN has a non-NIL value, then the window will be scrollable as follows:

Whenever the cursor moves from inside the window to outside the window across the window's right border, the window package (in particular, the scroll handler) will bring up a scroll bar.

If the user presses a mouse button while the scroll bar is up, then the window package will call the function that is the value of the SCROLLFN property. If the user continues to hold down the mouse button while in the scroll bar, the SCROLLFN will be called repeatedly every few milliseconds until the button is released.

The SCROLLFN will be called with the following 4 arguments: 1) the window, 2) the distance to scroll horizontally, 3) the distance to scroll vertically, and 4) a flag that indicates whether the button is still being held.

The distances to scroll depend on which mouse button was pressed as follows:

LEFT ž the distance in screen units (bits) from the cursor to the top (left if horizontal scroll) of the window.

RIGHT ž negative of the distance in screen units (bits) from the cursor to the top (left if horizontal scroll) of the window.

MIDDLE ž a **FLOATP** that describes the ratio of the distance between the cursor and the top (or left) edge of the window to the length of the entire scroll bar. (In other words, proportion of the way down (or across) the scroll bar that the cursor is.)

The **SCROLLFN** should take care of changing the contents of the window as appropriate for the distances specified by the user's mouse press.

Every window has a **SCROLLFN** property.

SCROLLBYREPAINTFN

The window package provides a default **SCROLLFN**, called **SCROLLBYREPAINTFN**, that can be attached to the **SCROLLFN** property of any window that has a non-NIL **REPAINTFN** property.

The idea behind **SCROLLBYREPAINTFN** is as follows:

A window is seen as showing at any given time a small part of a much larger display. The coordinate system of this display is the coordinate system of the window's display stream.

Each window has a property called **EXTENT** that describes the region in the display stream's coordinate system that the larger display occupies. Setting the **EXTENT** property of the window is the responsibility of the program that uses **SCROLLBYREPAINTFN**.

The ClippingRegion property of the window's display stream determines what region of the display stream (and hence what region of the EXTENT or larger display) is being currently shown in the window.

When a window scrolls, the ClippingRegion of the display stream changes Ź another part of the larger display is now being shown in the window.

When a window's REPAINTFN gets called it is passed a window and a region in the window's display stream. The REPAINTFN is responsible for drawing in the window that region of the display stream.

Thus, SCROLLBYREPAINTFN works as follows:

SCROLLBYREPAINTFN calls the window's REPAINTFN to draw in the window the contents of the new ClippingRegion whenever a scroll is requested (i.e., whenever the window's SCROLLFN is called).

SCROLLBYREPAINTFN takes care of translating the scroll distances (see the SCROLLFN description above) into a new ClippingRegion and then calling the window's REPAINTFN with the new ClippingRegion as an argument.

The programmer need only write a REPAINTFN that can display an arbitrary region of the larger display underlying the window. The programmer must also specify the EXTENT of the underlying display since SCROLLBYREPAINTFN needs to this to calculate the new ClippingRegion after each scroll.

Example

The goal is to develop a scrollable window that displays a list of strings, one string per line.

```
(MakeStringWindow
  (LAMBDA (StringList)
    (* fgh: "28-Jun-85 12:11")
    (* * Create a scrollable window to display the strings in
      StringList, one string per line.)

    (LET ((SW (CREATEW NIL "String Window")))
      (* * SCROLLFN will be the default -- SCROLLBYREPAINTFN)

      (WINDOWPROP SW (QUOTE SCROLLFN)
        (FUNCTION SCROLLBYREPAINTFN))

      (* * REPAINTFN will be StringWindowRepaintFn)

      (WINDOWPROP SW (QUOTE REPAINTFN)
        (FUNCTION StringWindowRepaintFn))

      (* * EXTENT of underlying display for scrolling is a region whose
        lower-left corner is at 0,0 and whose width is the width of the
        window and whose height is the number of strings times the height
        of each line as determined by the height of the window's font.)

      (WINDOWPROP SW (QUOTE EXTENT)
        (CREATEREGION 0 0 (fetch (REGION WIDTH)
          of (WINDOWPROP SW (QUOTE REGION)))
          (TIMES (LENGTH StringList)
            (FONTPROP (DSPFONT NIL SW)
              (QUOTE HEIGHT))))))

      (* * Cache the string list on the window so we can use it
        in the REPAINTFN)

      (WINDOWPROP SW (QUOTE StringList)
        StringList)

      (* * Display the first window full -- will be region out of EXTENT
        starting from 0,0 with height and width determined by the
        height and width of the window.)

      (REDISPLAYW SW)

      (* * Return the window)

      SW)))

(StringWindowRepaintFn
  (LAMBDA (Window Region)
    (* fgh: "28-Jun-85 12:21")
```

```

(* * Repaint a string window. StringList to print is cached on
the StringList property of the window. Region argument is used to
determine which strings to print in the window. Starting from 0,0
for each LineHeight increment print 1 string. Thus, string 1 goes
at 0,0. String 2 goes at 0, LineHeight. String 3 goes at 0,
(2*LineHeight) etc. LineHeight is just the height of the font
used in the window.)

(LET ((StringList (WINDOWPROP Window (QUOTE StringList)))
      (LineHeight (FONTPROP (DSPFONT NIL Window)
                            (QUOTE HEIGHT)))
      (RegionHeight (fetch (REGION HEIGHT) of Region))
      (RegionBottom (fetch (REGION BOTTOM) of Region))
      FirstLine LastLine)

  (* * Determine the first {i.e., top most} line to print)

  (SETQ FirstLine (ADD1 (FIX (QUOTIENT (PLUS RegionBottom RegionHeight)
                                       LineHeight))))

  (* * Determine the last {i.e., bottom most} line to print)

  (SETQ LastLine (ADD1 (FIX (QUOTIENT RegionBottom LineHeight))))

  (* * Move to the left edge and the bottom of the first {top most}
line in the window. The Times/Quotient construction is to
position on an exact line base, i.e., on Y position that is
an exact multiple of LineHeight.)

  (MOVETO (fetch (REGION LEFT) of Region)
          (TIMES (QUOTIENT (PLUS RegionBottom RegionHeight)
                          LineHeight)
                LineHeight)
          Window)

  (* * For the top most to the bottom most line in the Region, print
the corresponding string then move down a line. If the calculated
string number is not positive, then we have scrolled below the
end of the EXTENT. In this case don't print any string.)

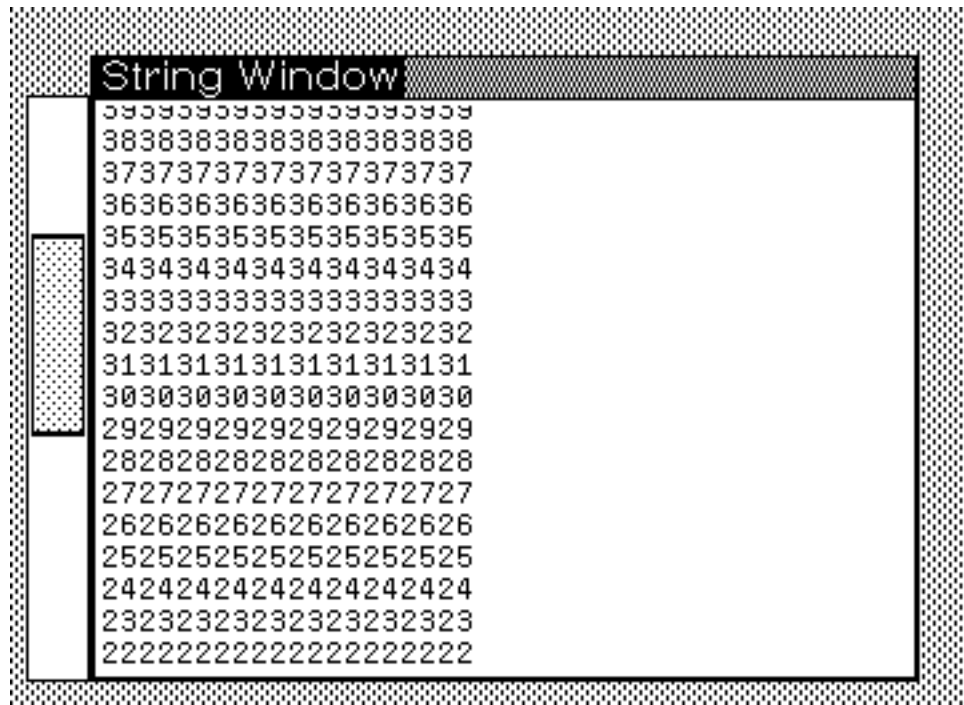
  (for Line from FirstLine to LastLine by -1
    do (AND (GREATERP Line 0)
            (PRIN1 (CAR (NTH StringList Line))
                  Window)
            (DSPXPOSITION 0 Window)
            (RELMOVETO 0 (MINUS LineHeight)
                       Window))))

```

32_(MakeStringWindow

(FOR X FROM 1 to 50 COLLECT (CONCAT X X X X X X X X X X)))

{WINDOW}#45,5643



Menus

The Menu package is very closely related to the Window package.

Basically, a menu is a window (or part of a window) in which pressing a mouse button in certain specified regions (i.e., menu items) causes certain specified events to happen.

In Interlisp-D, a menu is implemented as window with a special `BUTTONEVENFN` that uses a menu data structure to determine what action to take depending on what menu item the cursor is over when the button was pressed.

Bringing a menu up on the display requires two steps: 1) creating the menu data structure, and 2) displaying the menu in the window.

Creating the menu data structure

A *MENU* is a system datatype. Cached on the data type is all the information the menu package needs in order to display the menu in a window and in order to carry out actions when the mouse is clicked within the menu.

The MENU datatype has the following fields:

Fields that describe the menu's behavior

ITEMS – a list of the items to be included in the menu. If an item is an NLISTP, then it is displayed in the menu. If an item is a list, then the CAR of that list is displayed in the menu.

The display consists of the print name of the item or its CAR unless it is a bitmap, in which case the bitmap is displayed.

Under most circumstances, each item will be a list of three elements: the item *label*, the item *action*, and the item *message*. This list is interpreted by the default `WHENSELECTEDFN` as described below.

If a `WHENSELECTEDFN` other than the default is used, then each item can consist of whatever that `WHENSELECTEDFN` requires.

WHENSELECTEDFN ž a function that gets called whenever an item is selected in the menu by a mouse click. The function will get called with three arguments: 1) the item selected, 2) the menu, and 3) the mouse key used (`LEFT`, `MIDDLE`, or `RIGHT`).

The `WHENSELECTEDFN` defaults to the function `DEFAULTWHENSELECTEDFN` which operates as follows:

If the `CADR` of the item is non-`NIL`, then that `CADR` is evaluated and returned as the value of the function.

Otherwise, the item itself is returned.

WHENHELDFN ž a function that gets called whenever the user holds the mouse button down inside an item for more than `MENUHELWAIT` milliseconds (defaults to 1200). The function will get called with three arguments: 1) the item selected, 2) the menu, and 3) the mouse key used (`LEFT`, `MIDDLE`, or `RIGHT`).

The `WHENHELDFN` defaults to the function `DEFAULTMENUHELDFN` which operates as follows:

If the `CADDR` of the item is non-`NIL`, then that `CADDR` is printed in the prompt window.

Otherwise, *This item will be selected when the button is released* will be printed in the prompt window.

WHENUNHELDFN ž a function that gets called whenever the `WHENHELDFN` has been called and the user lets up on the mouse button or moves the cursor out of the item. The function will get called with three arguments: 1) the item selected, 2) the menu, and 3) the mouse key used (`LEFT`, `MIDDLE`, or `RIGHT`).

The default WHENUNHELDFN is CLRPRMPT, which clears the prompt window.

Fields that describe the menu's looks

TITLE ž the title to appear in the title bar of the menu. If NIL, then the menu will have no title bar.

MENUFONT ž the font in which the items will be printed. Defaults to Helvetica 10.

MENUROWS *or* **MENUCOLUMNS** ž the number of rows or columns the menu is to have. If both of these are NIL, the menu will have one column.

ITEMHEIGHT, ITEMWIDTH ž the height (width) of each item in the menu. If ITEMHEIGHT is not specified, then the height will be the maximum height of any item in the menu (as determined by the sizes of any bitmaps and the height of the MENUFONT). If ITEMWIDTH is not specified, the maximum width of any item will be used.

CENTERFLG ž if non-NIL, the menu items are printed centered in their areas. Otherwise, they are left-justified.

MENUBORDERSIZE ž the width in bits of the black border around each item. Defaults to 0.

MENUOUTLINESIZE ž the width in bits of the black border that outlines the entire menu. Defaults to the maximum of 1 and MENUBORDERSIZE.

Fields that describe the menu's positioning

MENUPOSITION ž the default position of the menu in screen coordinates (for pop-up menus) or window coordinates (for permanent menus) [See below for op-up versus permamnent menus].

If NIL, then the menu is placed at the cursor.

The point of the menu to be placed at MENUPOSITION is determined by MENUOFFSET.

MENUOFFSET is the point inside the menu that will be placed over MENUPOSITION. Defaults to 0,0, i.e., the lower-left corner.

To create a menu data structure use the standard CREATE statement from the Record Package:

Example:

```
(SETQ MenuX
  (create MENU ITEMS _ '(Yes No) CENTERFLG _ T
    TITLE _ "Yes or No??"))
```

To change a menu data structure use the standard replace statement from the Record Package:

Example:

```
(replace (MENU TITLE) of MenuX with "Well??")
```

Bringing up a menu on the screen: Pop-up and Fixed Menus

There are two ways to use a menu: *pop-up* and *fixed*.

Pop-up menus are used in a program to get some information from a user. A program using a pop-up menu brings the menu up on the screen and then waits for the user to select an item. When the item is selected, the menu is removed from the screen and the menu's WHENSELECTEDFN is called. The WHENSELECTEDFN will carry out some action, return a value to the program, or both.

Fixed menus remain on the screen "permanently". Whenever the user clicks a mouse button in one of the menu's items, the menu's WHENSELECTEDFN is called to carry out some action. Since fixed menus are not part of the ongoing processing of a program, the value returned by the WHENSELECTEDFN is ignored.

The MENU data structure is identical for pop-up and fixed menus. The difference is in the function used to bring the menu up on the screen.

The function for displaying a pop-up menu is:

(MENU *Menu Position*) ž displays *Menu* at *Position* (in the screen coordinate system) and then waits for the user to press and release a mouse button.

Pressing a mouse button has the following effects:

If the cursor is an item in *Menu*, that item is inverted on the screen.

If the user holds the mouse button down inside the item for MENUHELDWAIT milliseconds, then *Menu*'s WHENHELDFN is called.

If the user lets up on all mouse buttons while the cursor is still in the item, then *Menu*'s WHENSELECTEDFN is called.

If the user lets up on all mouse buttons while the cursor is outside of *Menu*, then no action is taken.

MENU returns the value returned by the call to the WHENSELECTEDFN, unless the user releases the mouse button while the cursor is outside of *Menu* in which case MENU returns NIL.

If the *Position* argument is NIL, then the MENUPOSITION field of *Menu* is used. If the MENUPOSITION field is also NIL, then the current cursor position is used.

Example:

```
(SELECTQ (MENU (create MENU ITEMS _'(Yes No)))
  (Yes (PRINT "The answer is Yup."))
  (No (PRINT "The answer is Nope."))
  (PRINT "No answer given."))
```

The function for displaying a fixed menus is:

(ADDMENU *Menu Window Position*) ž displays *Menu* in *Window* at *Position* (in the window's coordinate system) and returns immediately. The CURSORINFN and BUTTONEVENTFN of *Window* are replaced by the function MENUBUTTONFN so that when the user presses a mouse button inside an item in *Menu* (in *Window*) the following events take place:

The item is inverted.

If the user holds the mouse button down inside the item for MENUHELDWAIT milliseconds, then *Menu*'s WHENHELDFN is called.

If the user lets up on all mouse buttons while the cursor is still in the item, then *Menu*'s WHENSELECTEDFN is called.

If the user lets up on all mouse buttons while the cursor is outside of *Menu*, then no action is taken.

If no *Position* argument is given, then the MENUPOSITION field of *Menu* is used. If the MENUPOSITION field is also NIL, then the current cursor position is used.

If no *Window* argument is given, then a window is created just big enough to hold the menu and placed at *Position* (in the screen coordinate system).

ADDMENU always returns the window the menu was placed in.

Note: you can add more than one menu to a single window. But you cannot add a single menu to more than one window!

Since fixed menus are no automatically removed, the following function must be used to remove a fixed menu from the display:

(DELETEMENU *Menu CloseFlg Window*) ž deletes *Menu* from *Window*. If *CloseFlg* is non-NIL and there are no other menus in *Window*, then CLOSEW is called on *Window*.

If *Window* is not given, (OPENWINDOWS) is searched for the window containing *Menu*. If no such window is found, DELETEMENU is a no-op.

Hierarchical Menus

You can create hierarchical menus by creating menu items that have subitems.

A menu item with subitems is displayed with a small gray arrowhead at the right edge of the item.

If the user, drags the cursor from inside the item to outside the item across the right edge, then a submenu containing the subitems will be brought up.

Selecting one of these subitem from this submenu is functionally equivalent to selecting an item from the main menu.

Note that the subitems may themselves have subitems, making a fully hierarchical menu.

The MENU datatype has a field called SUBITEMFN. The value of the field should be a predicate that determines whether an item has subitems or not. The SUBITEMFN is called with the menu and the item as arguments. It should return either NIL to indicate that the item has no subitems or a list of subitems from which to use as the ITEMS field while constructing the subitem menu.

The default SUBITEMFN is the function DEFAULTSUBITEMFN which checks to see if the item is a list of 4 elements with the 4th element being a list whose CAR is SUBITEMS. If it is, it returns the CDR of this 4th element. Otherwise, it returns NIL.

Example of a menu with subitems and the default SUBITEMFN:

```
(create MENU
  ITEMS _
  '((Yes 'Yes "Answer Yes"
```

```
(SUBITEMS
  ("Yes w/ Check" 'Yes1 "Answer
  Yes, but check first.")
  ("Yes w/o Check" 'Yes2 "Answer
  Yes, without checking first.)))
(No 'No "Answer No."))
```



Menu Example

The following set of functions implements a fixed menu that sits left-flushed along the bottom of the screen. The menu has some common commands that you usually invoke from the Exec window.

Note: The function `CM.MakeCommandMenu` creates a fixed menu while the function `CM.TEdit` uses a pop-up menu.

```
(CM.MakeCommandMenu
  (LAMBDA NIL
    (ADDMENU
      (CREATE MENU
        ITEMS _
          '((TEdit (CM.TEdit) "Opens up a
          new TEdit window.")
          (FileBrowse (CM.FB) "Opens a File
          Browser window.")
          (Lafite (CM.Lafite) "Starts up Lafite
          mail program."))
        TITLE _ "Common Commands"
        MENUFONT _ '(HELVETICA 14 BOLD)
        MENUROWS _ 1
        CENTERFLG _ T
        ITEMHEIGHT _ 50
```



```

                                MENUBORDERSIZE _ 2)
                                NIL
                                (create POSITION XCOORD _ 0 YCOORD _ 0))))
(CM.TEdit
  (LAMBDA NIL
    (** Ask user if New or Old file to TEdit, then open a Tedit)
    (SELECTQ (MENU (create MENU ITEMS _ '(New Old)
                    MENUFONT _
                    '(HELVETICA 14)
                    ITEMHEIGHT _ 30
                    CENTERFLG _ T))
              (New (TEDIT))
              (Old (TEDIT
                    (PROGN (PRIN1 "Enter file name: ")(READ))))
              NIL)))
(CM.FB
  (LAMBDA NIL
    (FILEBROWSER)))
(CM.Lafite
  (LAMBDA NIL
    (LAFITE)))

```

Using the Mouse

Writing BUTTONEVENTFNs, CURSORxxFNs, and WHENSELECTEDFNs

In general, programs that depend on mouse actions are best written using the Window and/or Menu packages.

The BUTTONEVENTFNs and CURSORxxFNs for windows and the WHENSELECTEDFN for menus allow you to carry out arbitrary actions whenever the user buttons down inside a window or menu item or even whenever the user moves a the cursor within a window.

For example, the GRAPHER program for editing node-link graphs is built almost entirely on BUTTONEVENTFNs that fire whenever the user buttons down inside the GRAPH window. The BUTTONEVENTFNs determine what button was pressed and where in the window it was pressed relative to the graph being displayed. These use this information to decide what action to carry out.

There is an important factor to pay attention to when writing BUTTONEVENTFNs, CURSORxxxFNs, and WHENSELECTEDFNs.

In particular, these functions are evaluated as part of the Mouse process (i.e., the process that tracks the cursor and interprets mouse button presses). While these functions are being evaluated, the Mouse process can't be doing anything else, i.e., it can't be carrying out its normal job of tracking the cursor and interpreting button presses.

If a BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN is going to be long running, it is good practice to spin this evaluation off of the mouse process, so that the user can go off and do other things with the mouse while the function is being evaluated.

To do this, you can use the function call **(SPAWN.MOUSE)** in the BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN.

As described in LispCourse #14 (page 16), this will make the current mouse process (the one that's going to do the long running evaluation) into a process called **OLDMOUSE** and then start up a new mouse process.

The BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN will thus run under the **OLDMOUSE** process and not interfere with ongoing mouse operations.

Also, there is a convention in Interlisp-D that a events don't occur until the user releases a mouse button. For example, menu items are selected when the user releases the mouse button inside the item, rather than when she presses the mouse button.

When writing BUTTONEVENTFNs for windows, the programmer is responsible for adhering to this convention. When the BUTTONEVENTFN is called, it is good practice to wait until the user lets up on the mouse button before carrying out any actions. If the user lets up on the mouse button outside the window, then it is considered correct return without carrying out the action.

So, at the beginning of a BUTTONEVENTFN it is common practice to have a **(UNTILMOUSESTATE UP)** and **(OR (INSIDEP (WINDOWPROP Window 'REGION) (CURSORPOSITION Window)) (RETURN NIL))**. [See below for discussion of UNTILMOUSESTATE and CURSORPOSITION.

Getting Information about the Mouse and the Cursor

Many programs require information about the state of the mouse buttons or the cursor. The following functions return this information:

Position of the Cursor

(CURSORPOSITION *NewPosition Window*) ž Reads and then returns the location of the cursor in the coordinate system of *Window*. If *NewPosition* is specified, sets the cursor to be at the specified position in the window's coordinate system.

(LASTMOUSEX *Window*), **(LASTMOUSEY *Window*)** ž Returns the X (Y) location of the cursor in the coordinate system of *Window* as of the last time the cursor position was read. Does not actually read the current position.

LASTMOUSEX, LASTMOUSEY ž global variables that contain the X (Y) position of the cursor in the screen coordinate system as of the last time the cursor position was read.

(GETMOUSESTATE) ž Reads the current mouse state including the cursor location and sets the global variables LASTMOUSEX and LASTMOUSEY.

CURSORPOSITION calls GETMOUSESTATE before returning the cursor position. The functions LASTMOUSEX and LASTMOUSEY do not.

State of the Mouse Buttons

(MOUSESTATE *ButtonSpec*) ž Reads the mouse button state and returns T if that state matches *ButtonSpec*, NIL otherwise. *ButtonSpec* is description of the mouse buttons having one of the following forms:

LEFT, MIDDLE, RIGHT ý indicating the corresponding mouse button is down.

UP ý indicating all mouse buttons are released.

(ONLY *Button*) ž indicating that mouse button *Button* (i.e., one of LEFT, MIDDLE, RIGHT) is the ONLY button down.

(AND *ButtonSpecs*), (OR *ButtonSpecs*), (NOT *ButtonSpec*) ý indicating the logical combinations of other *ButtonSpecs*.

Note: MOUSESTATE is a macro which is like an NLAMBDA function in that its arguments are not quoted.

Examples:

```
(MOUSESTATE LEFT)
```

```
(MOUSESTATE (ONLY LEFT))
```

```
(MOUSESTATE (OR LEFT MIDDLE))
```

```
(MOUSESTATE (AND (NOT UP)(NOT (ONLY LEFT))))
```

(LASTMOUSESTATE *ButtonSpec*) ž Like MOUSESTATE but does not first read the current mouse state. Thus it returns the mouse state as of the time it was last read. Good for looking at exactly what caused MOUSESTATE to return T. (Does not first call GETMOUSESTATE as does MOUSESTATE.)

(UNTILMOUSESTATE *ButtonSpec Interval*) ž waits until (MOUSESTATE *ButtonSpec*) returns T or until *Interval* milliseconds have elapsed.

If (MOUSESTATE *ButtonSpec*) returns T, then UNTILMOUSESTATE returns T. If *Interval* milliseconds elapses without (MOUSESTATE *ButtonSpec*) returning T, then UNTILMOUSESTATE returns NIL.

If *Interval* is NIL, then UNTILMOUSESTATE will wait indefinitely.

Example of Using the Mouse

The following two functions implement a trap window ž if the user rolls the cursor into the window, he cannot roll it out again. Every time the cursor nears the edge of the window, the window jumps to be centered around the cursor location.

Thus wherever the cursor moves, the window will follow.

(Being a nice guy, there is an escape.)

```
(EX.CreateWindow
  (LAMBDA NIL (* fgh: "29-Jun-85 16:26")
    (LET ((Window (CREATEW (CREATEREGION 100 100 300 150))))
      (* * Add special CURSORINFN to Window)
      (WINDOWPROP Window (QUOTE CURSORINFN)
        (FUNCTION EX.CursorInFn))
      (* * Add a warning title)
```

```

(WINDOWPROP Window (QUOTE TITLE)
  "Caution: This window is trap"))))

(EX.CursorInFn
  (LAMBDA (Window)
    (* fgh: "29-Jun-85 16:28")

    (* * The user has moved inside of the window, don't let him out)

    (LET ((EdgeWidth 10)
          (Region (DSPCLIPPINGREGION NIL Window))
          (HalfWidth (QUOTIENT (fetch (REGION WIDTH) of (WINDOWPROP
                                                                    Window
                                                                    (QUOTE REGION)))
                                2))
          (HalfHeight (QUOTIENT (fetch (REGION HEIGHT)
                                       of (WINDOWPROP Window (QUOTE REGION)))
                                2))
          (Position (CURSORPOSITION NIL Window))
          NewRegion)

      (* * Compute a region just inside of the border of the window)

      (SETQ NewRegion (CREATEREGION (PLUS EdgeWidth (fetch (REGION LEFT)
                                                            of Region))
                                    (PLUS EdgeWidth (fetch (REGION BOTTOM)
                                                            of Region))
                                    (DIFFERENCE (fetch (REGION WIDTH)
                                                        of Region)
                                                EdgeWidth)
                                    (DIFFERENCE (fetch (REGION HEIGHT)
                                                        of Region)
                                                EdgeWidth)))

      (* * Forever, if the cursor moves out of the inner region move the
      window to center on the cursor position)

      (while T
        do (COND
            ((INSIDEP NewRegion (CURSORPOSITION NIL Window Position))
             (* Allow user to escape if
             chords the left and right mouse buttons)
             (AND (LASTMOUSESTATE (AND RIGHT LEFT))
                  (RETURN NIL)))
            (T (MOVEW Window (DIFFERENCE LASTMOUSEX HalfWidth)
                            (DIFFERENCE LASTMOUSEY HalfHeight))))
            (* Allow other processes to run)

            (BLOCK))))))

```

References

Chapter 19 of the IRM! But beware that this material is fairly old and out-of-date. Look in the Interlisp release messages for updates.

LispCourse #41: The Compiler and MASTERSCOPE

The Interlisp Compiler

What is the Interlisp Compiler?

An important goal of any program is that it run FAST. *Ceteris paribus*, the faster it runs the more work it can do.

Running fast, usually means doing as little work as possible while the program is running.

There are two ways to accomplish this:

- 1) Minimize the amount of work to do
- 2) Do some work ahead of time so there is less work to do while the program is running.

Recall from LispCourse #34 (and Homework #34) that the Lisp evaluator (i.e., EVAL/APPLY) does lots and lots of work whenever it evaluates a function call.

Because of this, evaluating a function call is relatively slow.

And because programs are made up of evaluating function calls, (interpreted) Lisp programs tend to be relatively slow.

Moreover, in the evaluator outlined in LispCourse #34 (& 35), all of the work is done each and every time the function is called, at the time function is called.

Much of this work is in fact redundant and need only be done once, e.g., when the function is called for the first time.

Thus the Lisp evaluator in LispCourse #34 ignores both of the speed-up techniques described above.

The goal of the *compiler* is to make these two speed-up techniques available in the Interlisp evaluation process.

The compiler is a program that takes a Lisp function definition in source code form (i.e., in the form that you write it in) and does as much of the "evaluation work" as it can.

It then rewrites the function definition in a new form (i.e., compiled code form) that captures all of the work it has done.

When the Lisp evaluation process encounters a function definition in compiled code form, it can take advantage of the work the compiler has already done and therefore APPLYing the function is much faster than if the function were in source code form.

Since the compiler is run once, before the function is ever evaluated, it does two things:

- 1) it minimizes work by getting rid of the redundancy in multiple evaluations
- 2) it moves some of the cost of function call evaluation from the time of evaluation to some earlier time (i.e., to the time when compilation is done).

The result is that compiled functions can be evaluated much, much faster than the equivalent interpreted (i.e. source code) functions. This is an obvious advantage.

The disadvantage of the compiler scheme is that you have to take the time to compile your functions before you run them.

This can be a big disadvantage when you are debugging and testing your program.

In these cases, you make frequent changes to your function definitions.

If you have to recompile your functions each time you make a change, the compile time can easily outweigh the time wasted by the slower evaluation of interpreted functions.

Other disadvantages of compiled code are the following:

- 1) Compiled code can be read only by real Lisp wizards. To the rest of us, it looks like gibberish.

- 2) Compiled code to some extent brings back the data/program distinction into Lisp.

In source code form, function definitions are just list structures that can easily be treated as data by another function.

In compiled code form, function definitions are special objects that are not easily accessible using the standard Interlisp data manipulation mechanisms.

The bottom-line is that compiled code is good for helping finished programs run fast while interpreted code is good for testing and debugging, and when you need to blur the distinction between data and program.

An important feature of Interlisp (in fact of most Lisps) is that compiled code and interpreted code can be freely intermixed without any special considerations on the part of the programmer.

An example of what the compiler does

The compiler is a very complex program that does lots and lots of fancy things to speed up the evaluation of Lisp code.

Compiler research and the problem of compiled code optimization are important research areas in computer science.

None of this will be covered here (since I don't know anything about it!).

However, the following is an example of what the compiler does:

Consider the following abstract function definition:

```
(LAMBDA (A B) (LET (D E)(LET (F G) (LIST A B D E F G))))
```

In the *interpreter* outlined in LispCourse #34, evaluation of the LIST function call would involve calling the **LookUpValue** function 6 times to look up the values of the atoms A, B, ...

Each time, the **LookUpValue** function would search up the stack looking for a binding of the given variable. For A & B, it would find the binding on the third stack frame from the bottom. D & E would be found on the second stack frame, etc.

All of this stack searching would take lots of time.

The compiler, however, can do some of the interpretation work ahead of time. For example:

The compiler can predict (based on the structure of the Lisp source code) that any reference to *A* within the second embedded LET will refer to the *A* bound in the third stack frame from the bottom.

Furthermore, since *A* is the first item in the parameter list, the compiler can figure out that *A* will be the first bound variable in its stack frame.

The compiler knows about the format of the stack. It can therefore generate code that directly fetches the value of the first bound variable in the third stack frame without any search of the stack.

Thus, the compiler would replace all the references to the value of *A* within the second embedded LET with compiled code that just looks up the value of the first bound variable in the third stack frame.

Then, when the function is later evaluated, the expensive stack lookup operation for the value of *A* would be skipped.

The compiler could do the analogous thing for all of the bound variables within this function definition.

If the function were compiled, the evaluation of the (LIST ...) statement would involve no stack lookup operations, resulting in a much faster evaluation.

Note that the compiler *cannot* do the same thing for free variable references. This is because the stack frame binding referred to by a free variable is determined at *run time* by what functions and bindings are currently on the stack.

The compiler has no way of predicting what the stack will look like at run time, and can therefore not replace the stack lookup by a direct reference.

A good compiler, however, could also do some optimizations in this function.

In particular, the two LETs could be collapsed into a single LET, eliminating the need to create an additional stack frame.

This is possible because the compiler can tell that the embedded LET does not rebind any of the variables used in the outer LET.

Using the compiler

Compiler Questions

All of the functions that invoke the compiler start by asking the user the following series of questions. Each question should be answered "Yes" (or "Y") or "No" (or "N"), followed by a Carriage Return.

Listing? ž Asks whether you want a detailed listing of the compiled code being generated. Always answer this question with "No".

REDFINE? ž Asks whether the compiled code should replace the source code as the function definition for the functions being compiled. In general, this question should be answered "Yes".

Occasionally, you may just want to create a file of compiled code without altering the definitions in the current virtual memory. In this case, answer this question with "No".

SAVE EXPRS? ž Asks whether to save the original source code whenever a function is redefined using its compiled code.

If "No", then the original code is lost when the function is redefined with the compiled code.

If "Yes", then the original source code is placed on the property list of the atom that is the function's name using the property EXPR.

The editor, the compiler, the file package, etc. all know about the EXPR property and handle in appropriately.

For example, if you call DEdit on a function whose definition is compiled code, DEdit will instead edit the source code stored in the EXPR property (if there is any).

If the function definition stored in EXPR is changed during the DEdit, then DEdit automatically redefines the function to be the *new* source code and saves the old compiled definition on the property list under the property CODE.

In general, answer this question "Yes". because you will often want to edit the source code and recompile the function.

If you answer "No", you will have to read the source code from a file (if you even bothered to save it) when you want to change the function.

OUTPUT FILE? † Asks whether to write the compiled code to a file that can be LOADED at a later time or in a new sysout, etc.

"No" means no file will be created.

"Yes" will cause the compiler to prompt you for a file name.

Anything else will be interpreted as a file name, in which case "Yes" will be assumed and that file will be used.

Note: as a shorthand you can answer the **LISTING?** question can be answered using the following:

S ž use the same answers to all questions as given for the last compile.

F ž just compile to a file without redefining the functions in the virtual memory.

ST ž answer REDFINE? and SAVE EXPRS? with "Yes" but ask about the output file.

STF ž answer REDFINE? with "Yes" and SAVE EXPRS? with "No" and ask about the output file.

Functions that invoke the compiler

The following function invoke the compiler:

(COMPILE *Functions*) ž Compile the current source code definitions for each of the functions in the list *Functions*. If *Functions* is an atom, (LIST *Functions*) is used.

The current source code definition is either the function definition or the source code stored under EXPR on the property list.

(TCOMPL *Files*) ž Used to compile source code files created by MAKEFILE. *Files* is a list of source code files to be compiled one-by-one in order. If *Files* is atomic, (LIST *Files*) will be used.

Compiling a MAKEFILE file involves compiling all of the functions on that file, writing the compiled code to a new file of the same name (but with the extension .DCOM), and then copying all of the non-function items (e.g., VARS, RECORDS, etc) from the source file to the new compiled file.

The resulting .DCOM file is a LOADable replacement of the original MAKEFILE source file, except that the function definitions contain compiled rather than source code.

TCOMPL returns a list of DCOM file produced.

Note: Since TCOMPL automatically produces a file, it does not ask the OUTPUT FILE? question.

(RECOMPILE *File*) ž Used to recompile a single source code file *File* after one or more of its functions have been edited using DEdit.

RECOMPILE works like TCOMPL, except that it does not compile all functions on *File*. Instead the following scheme is used:

If the function definition in the virtual memory is an EXPR (i.e., is not compiled code), then RECOMPILE compiles that definition and writes it to the output DCOM file. [As indicated above, functions are redefined to be their EXPR version (source code) whenever they are edited using DEdit.]

If the function definition in the virtual memory is NOT an EXPR, then RECOMPILE simply copies the previous compiled definition from the previous version of the DCOM corresponding to *File*.

RECOMPILE is considerably faster than TCOMPL when only one or a few function definitions have been changed because it doesn't recompile functions that haven't changed.

Example

```
32_(DEFINEQ (AAA (LAMBDA (A B C) (PLUS A B C))))
(AAA)
33_(DEFINEQ (BBB (LAMBDA (A B C) (LIST A B C))))
(BBB)
34_(SETQ EXAMPLECOMS '((FNS AAA BBB)(VARS (XYZ 44))))
((FNS AAA BBB) (VARS (XYZ 44)))
35_(MAKEFILE 'EXAMPLE)
```

```

{PHYLUM}<HALASZ>EXAMPLE.;1
36_(TCOMPL 'Example]
listing? no
redefine? yes
save exprs? yes
(dwimifying AAA)
(AAA (A B C))
(AAA redefined)
(dwimifying BBB)
(BBB (A B C))
(BBB redefined)
({PHYLUM}<HALASZ>EXAMPLE.DCOM;1)
37_DF[AAA]
prop unsaved
AAA
38_(MAKEFILE 'EXAMPLE)
{PHYLUM}<HALASZ>EXAMPLE.;2
39_(RECOMPILE 'EXAMPLE]
listing? N
redefine? Y
save exprs? Y
(dwimifying AAA)
(AAA (A B C))
(AAA redefined)
BBB,
{PHYLUM}<HALASZ>EXAMPLE.DCOM;2

```

Special Considerations when Writing Code to be Compiled

Interlisp takes ever effort to make compiled and interpreted code totally interchangeable. However, there are some ways in which this simply cannot be done. The following are some special considerations involved in writing code that will be compiled.

All of these considerations are optional. They are simply ways of taking advantage of feature available in compiled but not interpreted code.

GLOBALVARS

As described above, the compiler writes special code to handle many of the variable references more efficiently than the standard stack search mechanism.

As we discussed in LispCourse #34, free variable reference in interpreted code is always done through a stack search unless you use the `GETTOPVAL/SETTOPVAL` functions. In the later case, the value cell of the atom is used directly without any stack search.

You have a little more control over this process in code produced by the compiler. In particular, you can declare any variable to be a **GLOBALVAR**.

Declaring a variable to be a **GLOBALVAR** tells the compiler that whenever that variable is used freely in a function, code should be generated to directly access the value of the atom, skipping the stack search. Declaring a variable to be a **GLOBALVAR** is essentially telling the compiler to generate code to do a `SETTOPVAL` or `GETTOPVAL` whenever the variable is used freely.

If you don't declare a variable to be a **GLOBALVAR**, then the compiler will generate code to do the normal stack search when it encounters that variable used freely.

There are several ways to declare a variable as a **GLOBALVAR**:

1) Put a clause in the `COMS` list that contains the functions that you want to use that variable as a **GLOBALVAR**. The clause should be of the form (**GLOBALVARS** *Var1 Var2 ...*).

When any of the functions on the file are compiled, free variable references for any variable in a **GLOBALVARS** clause will be compiled as global variables.

2) Put a property GLOBALVAR with value T on the property list of the atom. Anytime the compiler runs across this atom used as a free variable, it will compile it as a global variable.

3) Add the atom to the global list GLOBALVARS. Anytime the compiler runs across this atom used as a free variable, it will compile it as a global variable.

Macros

A macro is a Lisp form that is evaluated at compile time to produce a Lisp form that is in turn compiled.

The evaluation of a macro to produce the form to be compiled is called *expanding* the macro.

For example: `(LIST (CAR '(PLUS DIFFERENCE)) A B)` might be a macro that when expanded returns the Lisp form `(PLUS 33 C)` given that the value of A is 33 and B is C when the macro is expanded.

Note that the form that gets entered into the compiled function is `(PLUS 33 C)`. When this compiled function later gets evaluated, the variables A and B have no effect whatsoever, only the variable C (which didn't appear in the macro definition at all) is relevant to the evaluation.

Contrast the concept of a macro with the following Lisp construction:

```
(EVAL (LIST (CAR '(PLUS DIFFERENCE)) A B))
```

When this form is evaluated, the inner LIST function returns `(PLUS 5 6)` given that the value of A is 5 and B is 6. This form is then evaluated by EVAL.

Note that if this form were compiled, it would be compiled exactly as is. The value of A and B would not enter into the compiling process but would appear as variables in the compiled definition (as they are in the source definition). The value of A and B are

then used at evaluation time, i.e., when the compiled form is being evaluated.

Compiler macros are an important part of most Lisp dialects. Unfortunately, in Interlisp they are relatively poorly implemented and very clumsy to use. Therefore, macros are not used with high frequency in Interlisp.

Using Macros

Macros are used very much like functions & in fact many of the Interlisp "functions" we have talked about are in fact implemented as macros.

Example: If TestList is a macro name, then (TestList A B C) would be a form that calls that macro.

The name TestList cues the interpreter or the compiler that the TestList macro should be expanded using A B C as arguments to the expansion. (Expansion is described below.)

The form that actually gets evaluated or compiled is the form that results from this expansion.

The original form (TestList A B C) is basically ignored.

The compiler and the interpreter treat macros slightly differently:

Whenever the compiler encounters a form, it first checks to see if the CAR of the form is an atom with a macro definition. If so, it expands the macro (as described below) and compiles the result of this expansion instead of the original form.

If there is no macro definition, then the compiler looks to see if the CAR has a function definition.

In contrast, the interpreter looks first for a function definition and only if one is not found does it look for a

macro definition for the CAR of a form. If it needs and finds a macro definition, it expands the macro (as described below) and evaluates the result of this expansion in place of the original form.

Defining macros and macro expansion

To define a macro, you need to put a macro definition onto the property list of the macro's name under the property MACRO (or DMACRO).

The definition should be an Lisp form with one of the following formats:

(List SExpression) ž this is called a substitution macro. When this macro is expanded, each time an atom appearing in *List* appears in *SExpression*, the corresponding argument from the macro call (e.g., the A B C in *(TestList A B C)*) is substituted in its place. The result is *SExpression* with these substitutions.

Example:

If *(PUTPROP 'ADD2 'MACRO '((X) (PLUS X 2)))*
Then *(ADD2 (CAR Z))* would expand to *(PLUS (CAR Z) 2)*

(LitAtom SExpression) ž When this macro is expanded, *LitAtom* is bound to the CDR of the calling form. *SExpression* is then evaluated. The result of the expansion is the result of this evaluation.

This format allows you to compute the SExpression to be compiled (or evaluated).

Example:

If *(PUTPROP 'LIST 'MACRO '(Args (LET ((ConsList (CONS NIL NIL))) (FOR Item IN (REVERSE Args)*

```

DO (SETQ ConsList
    (CONS
      (CONS (QUOTE CONS)
            (CONS Item
                ConsList))
      NIL)))
(CAR ConsList)

```

Then: *(LIST 1 2 3)* would expand to *(CONS 1 (CONS 2 (CONS 3 NIL)))*.

(LAMBDA ParamList FunctionDefinition) ž When this macro is expanded, it simply returns the function call form generated by using the LAMBDA expression as the CAR of the form and the arguments to the macro call as the CDR of the form.

Example:

```

If (PUTPROP 'ABS 'MACRO '(LAMBDA
  (X) (COND ((GREATERP X 0) X)(T
  (MINUS X))))

```

```

Then (ABS (CAR (LIST 1 3))) would
expand to ((LAMBDA (X) (COND
  ((GREATERP X 0) X)(T (MINUS X))))
  (CAR (LIST 1 3)))

```

The purpose of this is to avoid the expense of a function call in the compiled code, but to make the source code look and function just like a function call.

The macro expansion takes care of turning the (ABS ..) form into an equivalent form that does not require a function call when compiled.

The bottom line on Interlisp macros

In several years of Interlisp programming, I have written only two macros.

On the other hand, there are others, especially those who were brought up in other Lisp dialects, who use macros quite a bit.

MASTERSCOPE

MASTERSCOPE is a very handy Interlisp package that comes loaded with every standard Interlisp sysout.

MASTERSCOPE will analyze functions for you, store these analyses in a database, and then allow you to ask questions about the analyzed functions and their relationships.

The following will be a very brief description of some of the commands available in MASTERSCOPE. The whole package is fairly complex and will not be covered in detail here (but see Chapter 13 of the IRM).

MASTERSCOPE has "English-like" commands implemented as a single NLAMBDA-NoSpread function whose name is ".". Thus all MASTERSCOPE commands consist of a period followed by a command or "question".

The following are some of the useful commands:

(. **ANALYZE ALL ON *File***) ž analyzes all of the functions on the source file *File*. This is usually the first command in a MASTERSCOPE analysis of a set of functions.

Example: (. ANALYZE ALL ON HOMEWORK34)

(. **SHOW PATHS FROM *Function***) ž opens a window that shows a graph of all the (analyzed) functions that are called directly or indirectly from *Function*.

Example: (. SHOW PATHS FROM LC.Eval)

(. WHO USES AS A RECORD *RecordName*) ž returns a list of functions that use the record *RecordName*.

(. EDIT WHERE ANY CALLS *Function*) or **(. EDIT WHERE ANY USES *Variable*)** or **(. EDIT WHERE ANY USES AS A RECORD *RecordName*)** ž invokes DEdit on every function that directly calls the function *Function* (or uses the variable *Variable*). The exact form in each function that does the calling (or using) is selected and centered in the DEdit window when DEdit starts up.

The **(. EDIT WHERE ..)** command can be followed by a ž and one or more commands from the TTY Editor. In this case, these commands will be carried out on the functions instead of invoking DEdit on the functions.

Example: The following will change all variables named *Stack* to *MyStack* within the HOMEWORK34 functions.

(. EDIT WHERE ANY USES Stack ž (R Stack MyStack))

MASTERSCOPE actually has many much more specific questions you can ask (e.g., free variable usage, who fetches from a record, etc.). Consult the IRM for details.

There are some strategies, however, that make MASTERSCOPE work better.

For example: If you have many functions each with a bound variable that has the same "semantics" in each of the functions, then call that bound variable the same thing in each function. Naming it the same in each function will not effect the operation of the functions because its locally bound, but it will make it easy to change all of the functions at once if you want to change the semantics of the variable.

In NoteCards. many (maybe 100 in total) functions have a bound variable called NoteCardID. It was initially called just ID, but at one point a LinkID was added to many of these functions and we decided to rename all ID variables to NoteCardID. Luckily we had named them all alike so that this could be accomplished in a single MASTERSCOPE command.

The bottom line: learn to use the basics of MASTERSCOPE. Its one of the handiest tools around for writing large programs in Interlisp.

References

The Interlisp Compiler is the subject of Chapter 12 of the IRM.

Macros are the subject of Section 5.5 of the IRM.

MASTERSCOPE is the subject of Chapter 13 of the IRM.