

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.

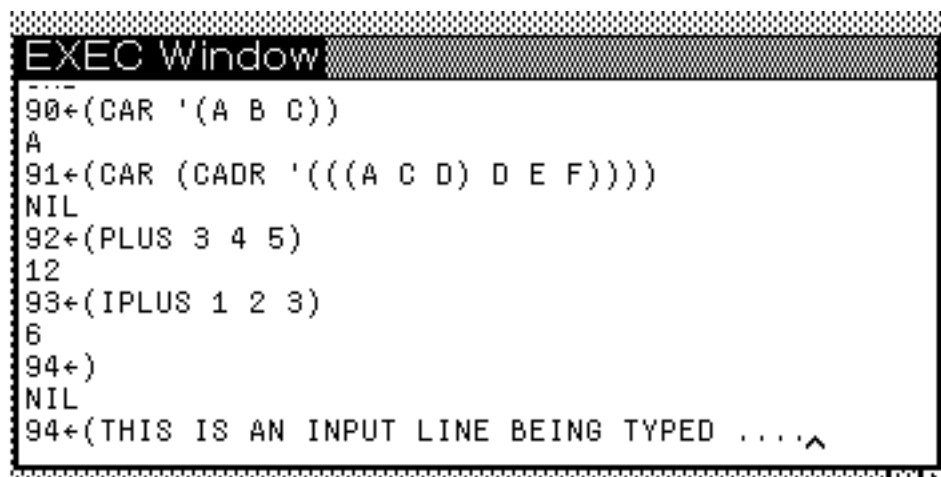
The TTYIN type-in editor

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:

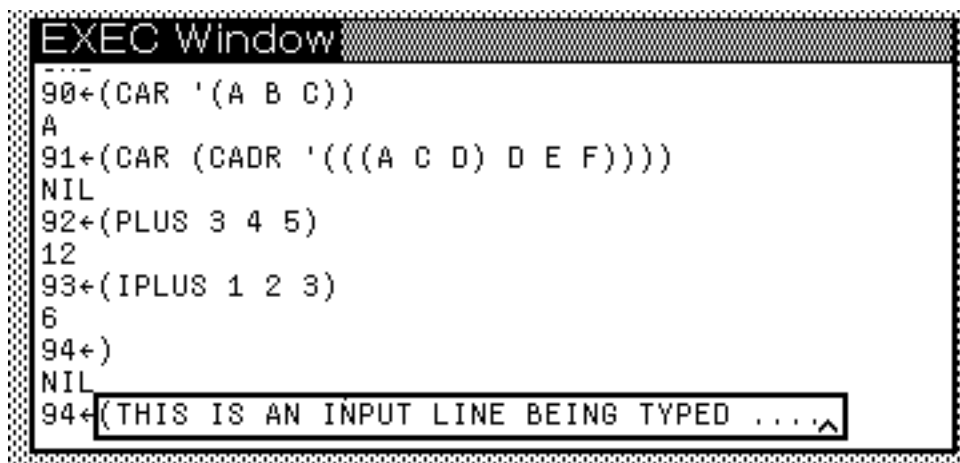


```

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 screenshot shows a window titled "EXEC Window" containing several lines of Lisp code. The code is as follows:
 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 line "35←(PLUS (PLUS (PLUS 2 3)" is partially visible. Two arrows are drawn on the screen: one points down to the closing parenthesis ']' of the third line (33←(EDITBM IT]), and the other points down to the opening parenthesis '(' of the last line (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 "[".

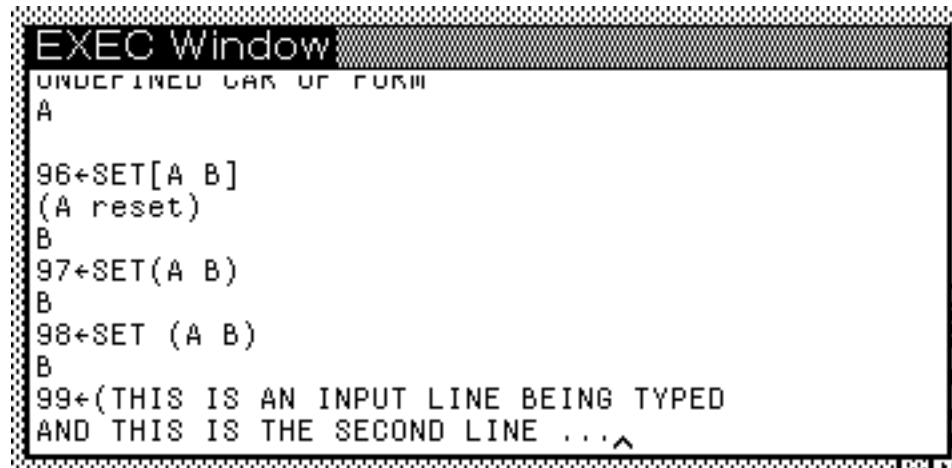
Examples:

(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 Ț 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 Ț 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 Ț 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 notmal 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 thru 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!