Harvard University
Applied Math 110

An Introduction To ED110

In order to make LISP a reasonably easy-to use language, there has to be some capacity for editing the deffinitions of functions. One method has already been presented to you; the user can pretty-print his functions out to a file, TECO that file to make the necessary corrections, and then LOAD that file to reconstruct the functions in L110. This method is convenient in one sense: you have already learned how to deal with the quirks of TECO to some extent, so you are not forced to learn a whole new language to edit your functions. But it is not so pleasent in several other ways. Ffirst, it is expensive from an efficiency point of view, since there is all the overhead of the translation of list structure to text in the Pretty-Print, followed by the translation back to list structure in the LOAD. Second, it is inconvenient for little changes that you might want to make and unmake rapidly, such as the insertion and deletion of calls on BREAK.

There is another way, and this is what ED110 is about. However, any curiosity or exploration on your part of our ED110 is totally optional; this is just for those who want to play a little, or who are dissatisfied with the TECO procedure. The principle is, why not change the list structure of the functions directly, with such destructive modifying fnctions as RPLACA and RPLACD, and edit the functions that way?

For example, if we wanted to change the function foo from a NLAMBDA to a LAMBDA, we could execute the following command:

```
->(RPLACA(GETD 'FOO) 'LAMBDA)
```

This technique works fine if we are trying to make a modification to the beginning of the function, but gets harder and harder the further "inside" the structure that the modification gets. So, we need some help, some other functions to make use of RPLACA and RPLACD on the list structure of function definitions easy. That is exactly what ED110 is.

To use ED110 you utter to the Harvard Shell:

```
@RUn <program> ED110<cr>
```

and you will find yourself with L110 with the edit functions pre-loaded. Alternatively, you can (LOAD 'EDIT T) from within L110 to obtain the same effects.

Note: This paper is only an introduction to ED110 and only describes a few of the functions that are available. If it seems to you that doing something with these is not convenient as it ought to be, you are probably right, and should go on to look at the more complete "A Users Guide toe ED110 Lisp".

Initilialization

To prepare the editor to edit a function (it only does one at a time), you use the function EF for Edit Function. It takes one argument that should evaluate to an atom, which is the name of the function which you want edited. For example: ->(EF 'FOO) will get the editor ready to edit the function binding of FOO.

Type-Out

To see where you are at any point in an editing process, you should yse the commands (TA) for Type-All, which takes no arguments, of (Z) for Zee where you are, which doesn't take any argument either. A TA will pretty print a little piece ot the function, specifically, the smallest sublist which contains the pointer (the pretty printing goes to POPORT, just like PP, so make sure POPORT is NIL). Yes, Virginia, sort of like TECO, ED110 has a pointer! To help you keep track of it, the above ffunctions type-out commands indicate its position with a little "*".

Now remember, ED110 is editing the list structure, not the text. One difference is the it can't deal with the individual characters within an atom. The pointer can't be located between the characters of "SETQ", for example, and therefore you can't change those characters one by one. Since "SETQ" is represented in the list structure as one pointer, the atom SETQ acts like one unit to ED110. This kind of unit is called a LEXEME. So, the ED110 pointer is located between two lexemes, rather than between two characters.

Transportation

To move the pointer over lexeme, there is the C command, for Climb. It takes one numeric argument, which can either be positive or negative. It moves the pointer over that many lexemes, either forward or backwards. For example: ->(TA)

```
(LAMBDA (X Y Z)(COND ((NULL * X) Y(T Z)))
->(C 1)
1
->(TA)
(LAMBDA (X Y Z)(COND ((NULL X *) Y)(T Z)))
->(C 1)
1
->(TA)
(LAMBDA (X Y Z)(COND ((NULL X) * Y)(T Z))) and so forth.
```

In case you want to move the pointer a fair bit, you should use the S command, for Search. It takes one argument, which should evaluate to an atom, and scans the buffer starting at the present position and going to the right, and puts the pointer after the next occurance of the atom, if there is one. An optional second argument, an integer, allows you to search for the n'th occurance all at once. For example,

```
->(TA)
(LAMBDA (X Y Z)( * COND ((NULL X) Y)(T Z)))
->(S 'Y)
T
->(TA)
(LAMBDA (X Y Z)(COND ((NULL X) Y * )(T Z)))
->(S 'X -2)
T
->(TA) (LAMBDA (X * Y Z)(COND ((NULL X) Y)(T Z)))
```

Of course, the only value of being able to move the pointer around is to be able to do something once it is there.

Insertion

To insert an atom, or nummber, or list, you use the I command, for Insert. It takes one argument, and inserts it at the present position of the pointer, and leaves the pointer after the insertion, like TECO. For example, ->(TA)

```
(LAMBDA (X Y Z)( * COND ((NULL X) Y)(T Z)))
->(I 'ZORT)
T
->(TA) (LAMBDA (x ZORT * Z )(COND ((NULL X) Y)(T Z)))
```

Deletions

To delete an element of a list, you use the D command, and what it stands for is pretty clear. It deletes the element of the list which immediately follows the present position of the pointer, whatever that element is (NOT just the next lexeme!). If the pointer is before a ")", at the the end of a list, it does nothing. For example: ->(TA)

```
(LAMBDA (X ZORT * Y Z)(COND ((NULL X) Y)(T Z)))
->(D)
T
->(TA) (LAMBDA (X ZORT * Z)(COND ((NULL X Y)(T Z)))
->(C 2)
2
->(TA)
(LAMBDA (X ZORT) * (COND ((NULL X)(T Z)))
->(D)
(LAMBDA (X ZORT Z) * )
```

Let that last example be a warning to you: delete can delete big chunks all at once, and if they are gone, they are gone for good.

Well, that ought to be enough to get you started in ED110. The complete write-up is in A USER'S GUIDE TO ED110 will tell you about lots of otherr nifty functions, of course. The rest of this will be some disjointed, helpful(?) comments.

Remember that ED110 only edits the list structure of the function. Don't expect it to correct any printed out versions that you may have around. When you are done editing with ED110, you should save by pretty-printing to a file, using update (UNIX L110 only) or PPF defined in the ED110 package (see description).

Commands to ED110 and regular LISP functions can be interspersed at will, by the way. So, if you are debugging a function, you can EF it, stick in a (BREAK), and then try it. Then, you can mmake other changes, and try it again. You only have to EF once.

Although "(" and ")" are lexemes, they can't be inserted or deleted as such. Indeed, what would it mean to delete one of a matching pair of parens? The editor does have commands to deal with parens, but look them up in the GUIDE

Lyle Ramshaw (now at Stanford) wrote ED110 when taking AM110 as a Sophmore at Harvard. This revised introduction is from the Ramshaw original, edited by Forrest Howard.