An Extensible Structured Data Editor for Interlisp–D

Lisp environments are populated by a variety of complex linked data structures, particularly linguistic structures (such as programs written in Lisp and other languages). These are often designed with visual representations (e.g. the pretty–printed form of Lisp code). Through such features as read macros and print definitions, the data structures may be converted to or from a textual representation. In many cases, the only convenient way to edit the structure may be to edit the textual representation and then re–interpret it. This approach suffers from several difficulties:

- writing the structure out and reading it back in will generally create an entirely new structure; this may be a problem if the structure is shared or part of a larger structure
- the requirement that enough information be written out to reconstruct the data when it is read back may conflict with the desire that the presentation be convenient to understand and change
- if the text editor is to provide any assistance (syntax checking, semantic checking, etc.) it must be integrated with the Lisp environment, and continually translating structures to and from their textual representation
- no allowance is made for nontextual graphic presentations

SEdit provides a different approach. Data structures are edited directly, with visual presentations provided as a means of communication. Like WSIWYG text editors, the editing window contains a continuously updated presentation of the data structure, and editing operations are input in terms of this presentation. Presentation and editing rules are defined for each data type, and a kernel program uses these rules to perform the editing.

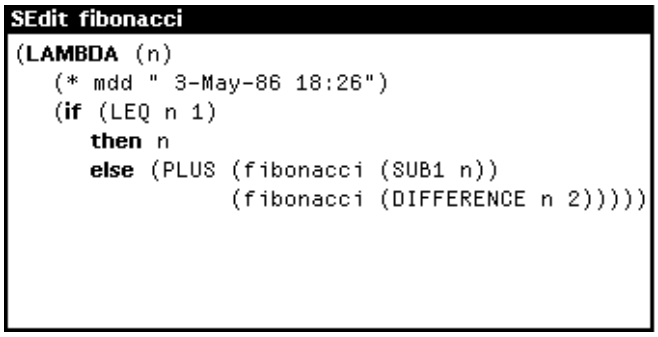
A set of such rules have been written to configure the editor as a tool for editing Interlisp programs. Interlisp code is pretty–printed in the window, and editing operations use a simple "point and type" user interface. This provides a more convenient way of editing Interlisp than previously existing tools, and furthermore will allow the simple construction of editors for other languages implemented in the Interlisp environment (Prolog, CommonLisp, 3–Lisp, Loops, etc.).

A Sample Session

Before discussing the structure and implementation of SEdit, we will try to give a feel for what it does by a brief description of some simple editing. Of course, it is a little hard to convey the flavour of as interactive a process as editing with just a few words and pictures. This discussion assumes some familiarity with the uniform editing interface which many Interlisp–D programs (and in fact most Xerox software) supports, since that was the model for SEdit's interface. The basic principles are that

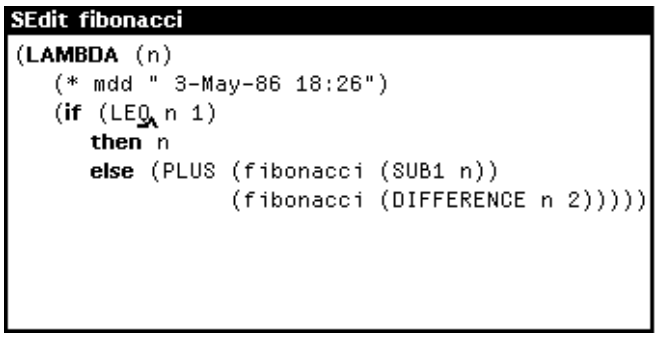
- there can be a *current insertion point*, usually indicated by some sort of caret, at which inserted material appears
- there can be a *current selection*, usually indicated by underlining or other highlighting, which indicates the material to be affected by a following command (commands may change the selected material, change some aspect of it (such as font or formatting), delete it, etc.)
- the point and selection are usually placed by positioning the mouse and pressing a button; the left or middle button places the point near the cursor and selects the indicated material, while the right button extends a previously made selection to include the indicated material
- material may be selected while holding down a modifier key; instead of becoming the current selection, an action is performed as soon as the key is released. The standard modifiers are `Copy`, which inserts a copy of the selected material at a previously chosen insertion point; `Delete`, which deletes the selected material; and `Move`, which copies the selected material and then deletes it

To SEdit an Interlisp function after SEdit has been loaded, use the usual (DF *function*). This creates an edit window for the function. Unlike the older display editor (DEdit), DF will return immediately, since the editing is done in a separate process. The function body will be pretty–printed in the window:

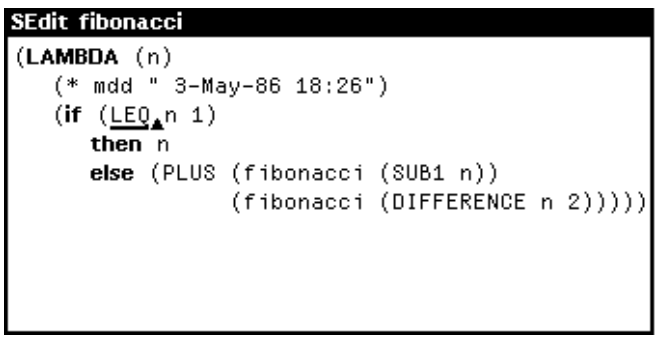


Since SEdit knows that this is Interlisp code, it uses some formatting rules specific to Lisp as well as its general rules for formatting lists, atoms, and strings. Formatting rules specify visual presentations as the positioning of text strings (in any font) and arbitrary bitmaps.

SEdit allows operations both on complete Lisp objects (such as lists and atoms) and on parts of objects, such as the individual characters in an atom's name. To do this conveniently, it uses the convention that the left mouse button is used to point to parts of structures, while the middle mouse button always points to whole Lisp structures. Thus, left–clicking the Q in LEQ selects that character, but middle–clicking there will select the whole atom (this convention matches TEdit's character/word selection convention). Furthermore, the same convention applies to insertion; a left click between the E and Q will allow inserting more characters, but a middle click will allow inserting a new structure (after the LEQ, since the mouse was closer to the end of the atom than the beginning). SEdit indicates the type of insertion expected by changing the caret, to '^' for a substructure and to '▲' for a structure. Thus, left–clicking the right side of the Q in the above window produces:



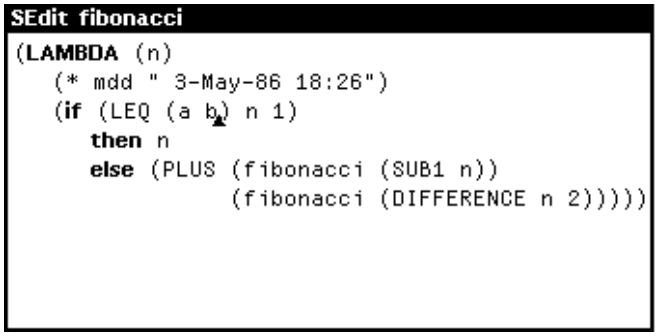
and any characters now typed would be appended to the atom LEQ. On the other hand, middle–clicking in the same place produces:



and if characters were inserted they would form a new atom.

Enclosing structures are selected by multiple clicks. In the previous example, if the middle button were clicked twice, the list (LEQ ...) would be selected; if it were clicked thrice, the (if ...) would be selected, etc. Sequences of structures or substructures are selected by extending the current selection with the right mouse button. If the atom LEQ has been selected, right–clicking the 1 would select the sequence of three structures LEQ, n, and 1 (this is *not* the same as selecting the list which contains these structures). Similarly, the onacc in the middle of fibonacci could be selected by left–clicking at one end of the sequence and right–clicking at the other end.

Some characters activate SEdit commands. For instance, typing a left parenthesis inserts an empty list at the current insertion point and positions the point inside it. Typing a right parenthesis positions the point just after the list immediately enclosing the current point. Typing a blank when inserting within an atom splits the atom at the point (unless the point is at one end of the atom) and switches to structure insertion. This allows Lisp list structures to be typed in as usual. For instance if we continue the above example by typing "(a b " the window will now appear like this:



(Note that the matching right parenthesis appears as soon as the left parenthesis is typed; this is an immediate consequence of having the window always be a pretty–printed presentation of the underlying list structure. Double quotes work the same way for strings.)

Modified selections allow the easy rearrangement of existing structures. For instance, to switch the order of the two arguments to PLUS, simply

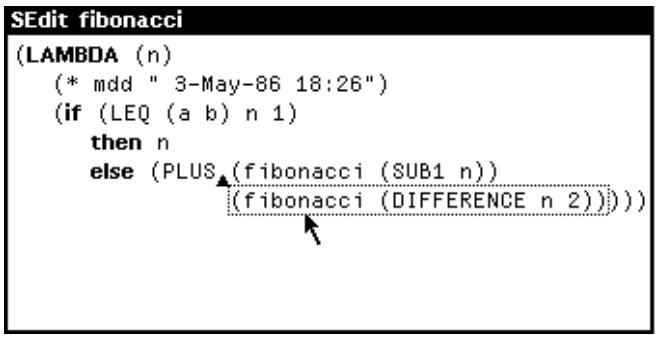
a) middle–click after PLUS (sets the insertion point)

b) hold down the Move key (we are going to move part of the structure to the current insertion point)

c) middle–click on the second fibonacci twice (selects the whole function call)

d) release the Move key (to indicate the selection is completed)

Just before the Move key is released, the window looks like this:



The selection is highlighted by a dotted outline to indicate that this is a Move selection (Copy selections have a dotted underline, pending–replace selections have a solid outline, and Delete selections are inverted). As soon as the change is made, the window is updated to show the resulting structure (SEdit determines what changes to the presentation result from the editing, and repaints as little of the presentation as possible).

An Overview of the Control Structure

The operations demonstrated above are actually performed by the cooperation of three separate pieces of code: the SEdit kernel, the SEdit user interface, and a set of methods defining the datatypes to be edited. By factoring the control this way, SEdit can be customized to edit new datatypes or provide different commands without having to modify or understand the complexities of the kernel (this is fortunate, since the kernel is quite complicated). The kernel invokes the datatype methods at appropriate times, manages the global sequencing of operations, and handles the optimization and caching necessary to update the screen efficiently. The datatype methods fall into three classes:

- methods for effecting and responding to changes in the structure being edited
- methods for positioning the point and selection appropriately
- methods for generating the visual presentation of the structure

The user interface translates mouse and keyboard events into appropriate calls on the SEdit kernel operations. It is not as modular as I would like; the keyboard event interpreter is table driven and readily extensible, but the mouse event interpreter is not extensible at present.

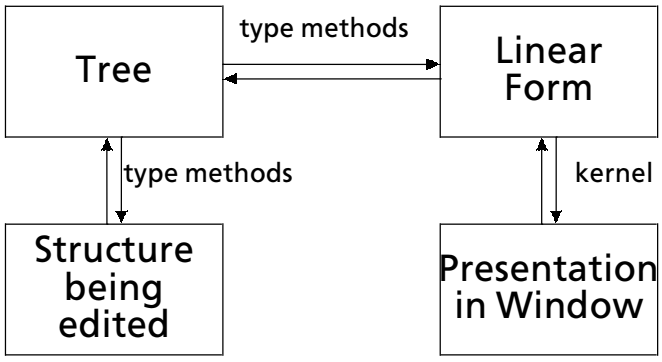
Each instance of SEdit has a *context*, which includes the structure being edited and the window in which the editing is being done. Editing operations are performed by calling kernel procedures and passing the context. The context is monitor locked, so operations may be invoked from several processes. Normally, each context has an associated keyboard process which reads keyboard commands and performs them. No editing state is maintained by the keyboard process; it can be deleted and recreated as convenient (but SEdit ensures that there is never more than one keyboard process per context).

SEdit is installed by calling the function `(EDITMODE 'SEDIT)`, whereupon it becomes the system display editor and will be used whenever the user asks to display edit a structure (through edit commands, Masterscope searches, inspectors, browsers, or whatever). The context remains active until the edit window is closed (although the keyboard process disappears while the window is shrunk).

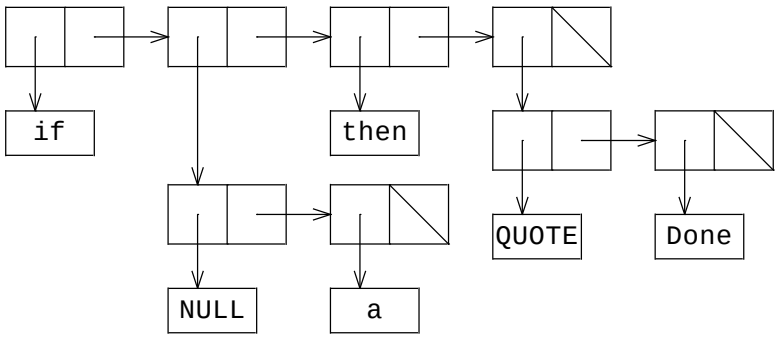
An Overview of the Data Structures

The SEdit user is aware of two data structures: the structure being edited, and the window in which it is presented. Internally, SEdit maintains two additional data structures: the *tree*, which is a representation of the structure being edited, and the *linear form*, which is a representation of the visual presentation appearing in the window. The tree provides a common representation for the underlying structures. This allows SEdit to implement a small number of simple editing operations uniformly described as tree mutations, while the type–specific methods map these into the actual changes required. It also caches various bits of information computed by SEdit. The linear form is a sequence of strings, atoms, and bitmaps to be displayed, intermixed with horizontal and vertical positioning information.

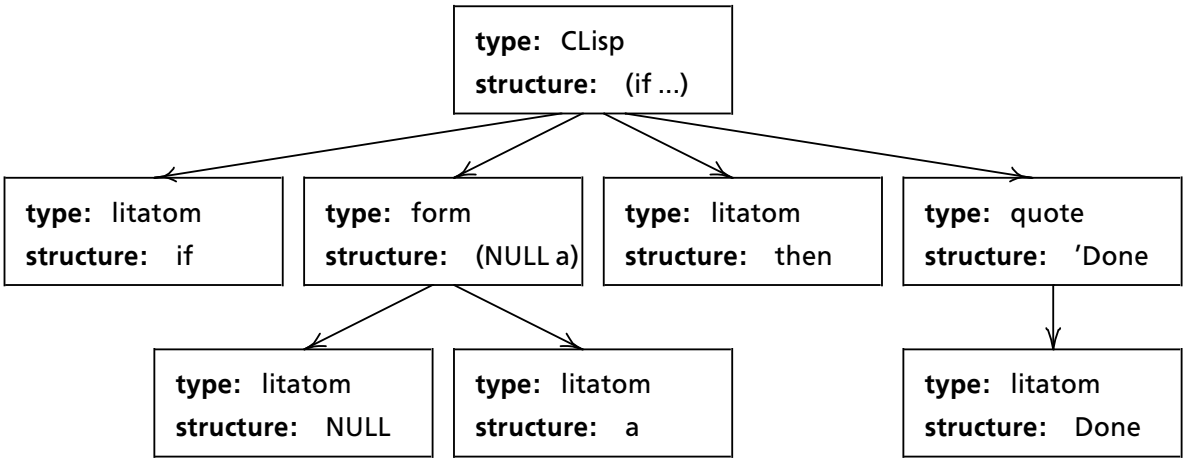
These four data structures are all representations of the same data. The constraints between them are maintained as follows:



As an example, consider editing the list structure `(if (NULL a) then (QUOTE Done))`. The underlying structure is composed of cons cells and atoms:

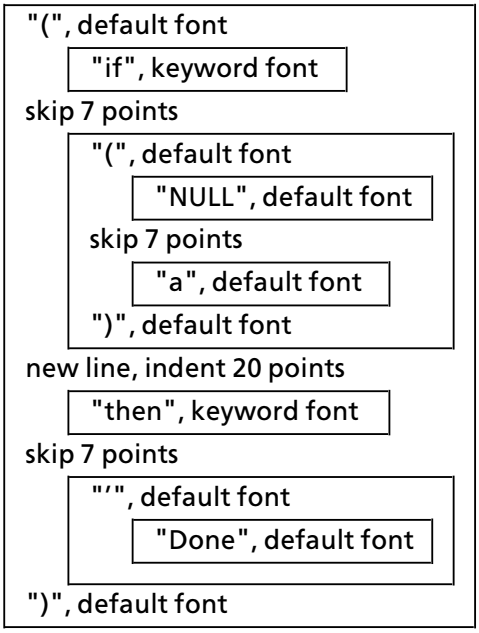


SEdit's tree will look like this:

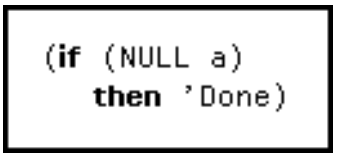


Note that each node in the tree has a type, indicating the type of the structure, and a field containing the structure itself. Note also that the types need not correspond directly with Interlisp's type structure: in this example, many cons cells are grouped together as one node in the tree, and may be marked as type `list`, `CLisp` (a class of special form), `form`, `quote`, `binding`, etc. depending on the components of the structure and its position in the surrounding structure. Thus, the list `(QUOTE Done)` is classified as type `quote` above (and has one subnode), but would have been type `binding` (with two subnodes) had it appeared in a list as the second element of a form identified as a `LET` or `PROG`.

The linear form will be something like:



Note that although the linear form is a sequence (hence its name), it reflects the structure of the tree from which it was generated. Finally, the window will show some or all of the presentation described by the linear form:



SEdit has two other important data structures. The context was mentioned before; it includes the data structures mentioned above, information such as the current selection and current insertion point, and all the other little bits of state SEdit needs to keep track of what is going on. The *environment* contains all of the customization information which controls SEdit's behaviour: the tables which drive the generation of the tree from the original data structure, the table for interpreting keyboard events, parameters such as default spacing and fonts, etc. At present, there is just one environment defined – the one which configures SEdit specifically for editing Interlisp code, but by creating new environments the user could have SEdit contexts for editing Prolog, CommonLoops, etc., all coexisting.

(Actually, there *is* a second environment defined, which tricks SEdit into pretty–printing Interlisp code into a TEdit document (to produce the listings later in this document), but it hardly qualifies as an *edit* environment.)

Documentation

The rest of the documentation for SEdit is in five sections:

- a more detailed account of how to use SEdit to edit Interlisp code
- a description of how SEdit is configured for editing a new type
- a detailed description of SEdit's internals
- an annotated listing of the code which implements the basic Interlisp types
- an annotated listing of SEdit's kernel

The first of these is independent of the others, but the second should be read before attempting to understand any of the following sections. If you are planning to extend SEdit to edit your own language (or change the way it currently edits one), you should also look at the third of these — a few good examples are sure to convey what the specifications missed.

Other Comments

The formatting and incremental reformatting algorithms, the incremental window update algorithms, and the various data structures were all developed by the author, so there are no references for them. Although there are several extensible prettyprinters for Lisp environments (notably Waters' GPRINT), some do not allow a sufficiently general class of presentations, most only deal with formatting lines of fixed width characters, and none deal with the (crucial) problem of incremental reformatting.

At present, SEdit is usable but far from polished. There is still room for improvement in the screen updating, and many rather desirable commands are still missing (notably *undo* and *redo*). Extra formatting rules for CommonLoops are a high priority, as is completing the interface to other Interlisp tools (e.g. the file package). Finally, the author hates the name SEdit, and promises three free features of their choice to the first person who comes up with one he likes better.