How SEdit Works

SEdit incorporates a variety of complex algorithms and data structures. Although the code is commented, there is no logical place within it to properly describe the use of these. Therefore this document provides an overview of the most interesting and tricky parts of the program. If you want to understand exactly what the code does, this isn't a replacement for actually reading it. This is, however, a highly recommended introduction to the code; it's probably not worth trying to understand the code if you haven't read this first.

The Code

The code for SEdit currently resides in five files, named SEdit, TopLevel, SEditWindow, Linear, and IntrLsp. The first four of these comprise the SEdit kernel and its interface to the rest of Interlisp-D; the last is the definitions which configure SEdit as an editor for Interlisp code. The approximate division of labor is:

SEdit
process initialization, keyboard command loop, top-level method invocation, building and manipulating the edit tree

TopLevel
interface to Interlisp-D editing functions and file system, starting and managing SEdit processes

SEditWindow
window system interface, mouse selection and pointing, scrolling

Linear
building and manipulating the linear form, optimized screen updating

IntrLsp
methods for the standard Interlisp types

Included with the code listings is a directory of SEdit function names, identifying the file within which they occur. Within each file listing the functions are sorted into alphabetical order. SEdit currently uses the convention of preceding its function names by a pair of backslashs, to clearly distinguish them from other parts of the Interlisp system. At some point a more conventional prefix will be chosen.

Control Flow

SEdit sessions are started via the system function EDITL, which originally invoked the TTY structure editor. When SEdit is enabled, EDITL is modified to call SEdit. SEdit first attempts to find an active SEdit already editing the same function or variable. If one is found, SEdit makes sure that the edit window is expanded and not buried on the screen, and then returns. In this way it avoids starting several SEdit processes on the same structure, which could lead to major confusion. If no active SEdit is found, it will start a new one. It places a window, either by asking the user where the window should be or by using the window position of a previous SEdit (to avoid prompting the user over and over again). Once it has a window, SEdit starts a new process running the keyboard processing loop, and waits for it to signal that it has initialized.

A further complication is that EDITL may be invoked with a sequence of TTY editor commands which are to be performed. Rather than attempt to implement the large and baroque command set of the TTY editor, SEdit simply calls the TTY editor to execute the command sequence, but with the constraint that should the command sequence involve a pause for user editing (i.e. the TTY: command), the TTY editor will call SEdit back. This mechanism is used heavily by Masterscope, which usually passes a sequence of editor commands which search for particular parts of the function and then allow the user to edit them.

If EDITL is invoked without any commands, it returns as soon as the command loop signals that initialization is under way. The editing will be handled by the new process, and whoever invoked SEdit can now go on with other things. Alternatively, if a command sequence is given, SEdit will

wait until the command process signals that the user has indicated editing is complete before returning. This is quite important; if EDITL doesn't wait under these conditions programs like Masterscope may try to start dozens of SEdit procedures, without giving the user time to actually do any editing.

The command loop which is executed by the SEdit process (\\sedit) first checks to see if this is a new SEdit context or an old one. If it's new, the structure must be parsed to generate the edit tree and linear form, window parameters must be initialized, and the initial presentation must be displayed. If this is a continuation of an old edit session, any pending adjustments to the presentation are completed, and the current selection is redisplayed. Either way, \\sedit then enters a loop reading single characters and executing the appropriate action.

Keyboard Input

The interpretation of characters typed from the keyboard is determined by an Interlisp readtable, which maps characters to syntax classes. The syntax classes understood by SEdit are:

STRINGDELIM: this is the delimiter for strings (i.e. double quote)

SEPRCHAR: white space, e.g. blank and carriage return

ESCAPE: when preceded by this character, other characters should be treated as syntax OTHER (the usual escape character is %)

OTHER: a "normal character", suitable for inclusion in the pname of a litatom

(*type where when function*): a read macro. *function* is the function to be invoked when this character is typed. *type* determines what information will be passed to the function, and how its result will be interpreted; at present the only legal value is INFIX, indicating that the function will be passed the complete edit context, and is free to make any changes whatsoever. *where* and *when* control under what conditions the function will be invoked; the valid combinations are:

(ALWAYS IMMEDIATE)
the function will be invoked whenever the character is input (e.g. Delete)

(ALWAYS NONIMMEDIATE)
the function be invoked except when the character is typed as part of a string (e.g. left parenthesis)

(FIRST NONIMMEDIATE)
the function will only be invoked when this character is typed at a structure point, i.e. as an atom on its own (e.g. period)

Note that these are not exactly the conventional interpretation of readtable entries, but have been adapted to SEdit's needs.

The action produced by a character depends on both the syntax class of the character and the type of the current point (string, atom, or structure). For instance, a SEPRCHAR typed to a string point will insert that character in the string, typed to an atom point will split the atom, and typed to a structure point will simply be ignored. An OTHER character typed to a string or atom point will be inserted, but typed to a structure point will cause a new node, representing the single character atom, to be created and inserted.

Mouse Actions

In the Interlisp-D window system the effects of mouse movement and buttons are determined by a special mouse process, which invokes functions attached to the windows. The effect of this is that activities like selecting and pointing are actually run under the mouse process, rather than under the SEdit process. This has several ramifications. First, the mouse process must be able to easily access the state of the editing process, so that it has the necessary information available to carry out the actions. For this and other reasons the entire state of the SEdit process is encapsulated in a single data structure, called an `EditContext`, which is then attached as a property of the window.

Second, access to the editor's data must be controlled, so that multiple processes don't interfere with each other. This is achieved through a monitor lock on the `EditContext` (called the `ContextLock`).

The Linear Form

The data structure used to encode the linear form must satisfy several constraints. First, as its name suggests, it is treated as a sequence of items. The window is repeatedly being redrawn from the linear form, and usually the segment which is redrawn cuts across the tree structure. This requires being able to start iterating through the linear form from any point, and continue until some other arbitrary point, perferrably without having to maintain a stack. On the other hand, it is generated and modified hierarchically; the kernel must be able to quickly extract and replace the linear form for any node in the tree, without affecting the nodes above or (in some cases) below it. To achieve this, the linear form for each node is stored as a list pointed to by the node, which ends with a pointer back to the node. This list may contain linear items and subnodes of the node, indicating that their linear forms are to be inserted at that point. Finally, each node contains a pointer (called the `LinearThread`) which points to the position in its supernode's linear form at which it appears. This threading allows any part of the tree structured linear form to be easily and efficiently traversed.

One complication to this scheme is that because of the reference counting garbage collector in Interlisp-D, we don't actually make the tail of the linear form point back to the node directly. This would introduce a circular chain of pointers, preventing these structures from ever being garbage collected. Instead, we use a data structure called a `WeakLink`, which contains one non-reference counted pointer field. Of course, the use of such pointers can make debugging hazardous; unfortunately, they are pretty much unavoidable in memory management scheme such as Interlisp-D's, and SEdit uses them in a number of places.

Most of the items which can appear in the linear form are relatively simple, but `LineStart`s are the exception. `LineStart`s are inserted by the linearization procedures to indicate that a new line is to be started, but are used by the kernel a lot of extra information. First, each `LineStart` is linked to the `LineStart` before and after it, allowing efficient access to sequential lines of the linear form. Second, each `LineStart` records the maximum ascent and descent of the items which appear on that line, since these determine the amount of vertical space required to determine the line. Since lines can contain arbitrary combinations of text in different fonts and sizes, as well as bitmaps of arbitrary size, each line's ascent and descent must be separately computed, and may change with any change to the nodes appearing on that line. Third, each `LineStart` has a pointer to the node in whose linear form it appears; although this could be determined by scanning along the linear form, it's used often enough that we cache it. Fourth, each `LineStart` stores the y coordinate of that line's baseline. This is a function of that y coordinate of the preceding line, the line ascents and descents, and the separation between the lines.

Thus, the algorithm for updating the window from the linear form is something like this:

```
(* pointer is the current position in the linear form)
while pointer is not NIL do
    if pointer is a list, then
        let item be the first element of the list
        if item is a number (* horizontal space), then
            increment current x by item
        elseif item is a LineStart, then
            set x to the indentation of item
            set y to the baseline of item
        elseif item is a bitmap or string item, then
            paint this item and increment x by its width
        else item is a subnode
            set pointer to the linear form of item
    else pointer is a WeakLink
        set pointer to the CDR of the linear thread of
            the node pointed to by this WeakLink
```

Incremental Relinearization

When the node structure is changed, SEdit is faced with the task of updating the screen as efficiently as possible. This is actually treated as two problems: first, the new linear form must be computed as efficiently as possible, and second, given the changes that occurred in the linear form, we wish to make the corresponding adjustments to the window, with a minimum of repainting.

The first problem is dealt with using a number of tricks. First, as changes are made to the tree no attempt is made to update the linear form immediately; all that is done is to keep a list of those nodes which have been changed and thus need their linear forms recomputed (this is the purpose of \\note.change). This list is kept sorted by depth, and duplicates are discarded Once SEdit decides that it's time to update the window, it finds a minimal set of nodes to relinearize, such that all of the changed nodes are contained directly or indirectly (as subnodes of an included node), and the width estimates of these nodes have not been changed by the editing (hence the linear form of their super nodes won't have changed). The linear form of each of these nodes is recomputed. Second, during this relinearization, subnodes of a relinearized node will not be relinearized if their structure hasn't changed, and SEdit is able to determine that their resulting linear form won't have changed (e.g. their margins haven't changed). In practice, this typically results in a major decrease in the amount of relinearization involved.

The second problem is more difficult. The idea is that we want to reuse as much of the existing window display as possible. If part of the linear form is already displayed in the correct place, no changes should be made to it. If part of it is already displayed, but in the wrong place, SEdit tries to use a bitblt call to copy those bits to the correct location rather than reconstructing them. This is complicated by the fact that material copied may be less than one line or span several lines. The line its moved to may have greater or less ascent and descent than the one it came from; in fact, the ascent and descent of the destination line won't be known until it's completed. To deal with this, SEdit builds a second description of the line, in parallel with the construction of the linear form. This structure describes the line as a sequence of blocks, each of which represents a subsequence of the linear form. Each block may also already appear in the window, in which case the coordinates from which it can be retrieved are also recorded. At the end of each line, SEdit examines the sequence of blocks, determines which ones actually represent useable blocks of bits, copies them, and repaints any gaps. To avoid overwriting bits which it may later want to use, SEdit will sometimes shift whole parts of the screen out of the way; this introduces additional complications, since coordinate transformations are now required to determine the actual location of the desired bits.

A Word About Lines and LineStarts

SEdit defines a `LineStart` data type to record information about a line in the linear form. The `LineStart` does not record the actual linear items which appear on that line; that information is implicitly recorded in the linear form itself. The linear form is an list of linear items, and `LineStart`s are items. Thus, the items on a line are those which follow it in this list (until the next `LineStart`). Since this information is often needed, many of the places where one might expect a pointer to a `LineStart` actually contain a pointer to the linear form which start at that `LineStart`, i.e. a list, the first element of which is the `LineStart`. This is referred to as a `Line`.

Pretty Printing

The code listings at the end of this document were generated by SEdit's formatting routines. This seemed like an obvious thing to do, but a couple of caveats are in order. First, the formatting rules are still incomplete; in particular, they do a miserable job on **create** expressions (which have a very non-LISP like syntax). Second, quite little modification was required to add this capability to SEdit, but the result isn't an ideal pretty printer. It does a good job (better than Interlisp's pretty printer), and hopefully will soon do an even better job, but it's a very expensive way to do it. Constructing the complete edit tree for a one-shot linearization consumes excessive memory and time, and consequently printing large files of functions requires considerable patience.

Contexts, in detail

(an annotated description of the `Context` data structure definition)

Environment: an EditEnv
The Environment provides a number of parameters controlling the editing process. The idea is that different environments are built to describe different editing tasks (editing different languages, or just different edit styles), and then shared between all edit contexts of that type.

DisplayWindow: a WINDOW
This is the primary window, in which the edited structure is displayed.

EditType: probably a litatom
Doesn't affect the editing, but used (in conjunction with IconTitle) to help identify the source of the structure being edited. One of VARS, FNS, PROP, etc.

IconTitle: a string or litatom
The name of the structure being edited, used to title the window and icon. Also, in conjunction with EditType, used to determine whether an existing SEdit session is already editing a structure the user has asked to SEdit.

ContextLock: a MONITORLOCK
The monitor lock used to control access to this Context.

CompletionEvent: an EVENT
SEdit will signal this event when the user shrinks or closes the window, or otherwise indicates that they have done enough editing. When called under the TTY editor, the process which spawns the SEdit command loop will wait for this event.

WindowLeft, WindowRight, WindowBottom, WindowTop: integers
During screen update processes, SEdit caches information about the window dimensions here, since they're used so frequently.

CurrentX: an integer
CurrentLine: a Line
While generating the linear form, SEdit uses these to record the start of the current line and the horizontal position within that line.

LinearPointer: a position within the linear form
This is the current position within the linear form, for purposes of comparison and insertion.

LinearPrev: a position within the linear form
This is one step behind the LinearPointer, to allow fixing up pointers when an item is inserted. If LinearPointer points to the first item within the linear form of a node, LinearPrev will point to that node.

Root: an EditNode
Points to the root node in the edit tree.

LastLinearizedSubNodeIndex: an integer
During linearization, this field is used to record the last subnode linearized of the current node (as a consistency check).

ChangedNodes: a list of EditNodes
This list records which nodes in the tree have had their structure changed and hence require relinearization. It's headed with a dummy item (NIL) to simplify insertion, and sorted by decreasing depth. Most of the time, this list should contain exactly those items whose Changed? field is T.

CaretPoint: an EditPoint
Records the current insertion point.

Selection: an EditSelection
Records the current selection.

Caret: a `CURSOR`
This is the mark to be displayed at the current caret point (hollow or solid, depending on the type of point).

SelectionDisplayed?: a boolean
Records whether the current selection has been underlined, outlined, or whatever.

LastMouseX, LastMouseY: integers
LastMouseType: one of (`Atom`, `Structure`)
Records the position of the last mouse click, and the button used, to detect multi-click sequences.

\X, \T: integers
These slots in the `EditContext` are used at one point in the program to return multiple values from a function; too bad there's no better way.

FirstBlock: a `LineBlock`
The first in the sequence of blocks constructed to describe the current line.

CurrentBlock: a `LineBlock`
The last (so far) in the sequence of blocks being constructed to describe the current line.

Matching?, Below?, Visible?: booleans
These are used while constructing the block sequence to keep track of our current state.

RepaintStart: a position within the linear form
RepaintLine: a `LineStart`
RepaintX: an integer
Also used to keep track of state while constructing the block sequence.

ShiftY, ShiftDown, ShiftRight: integers
When the block shifter moves the rest of the window contents out of the way, these variables record what has been done to allow translating coordinates. Everything below ShiftY has been shifted down by ShiftDown; everything on the current line past the current position has been shifted right by ShiftRight.

RelinearizationTimeStamp: an integer
During relinearization, the position of a line may move. However, the old position of that line may be later needed to locate useful bits. When the position or dimensions of a possibly useful line are changed, the old values are cached. Since the values are only good for the current relinearization, they are marked with this time stamp, and each relinearization its value is increased.

Environments, in detail

(an annotated description of the `Environment` data structure definition)

ParseInfo: a `PLIST`
This `PLIST` maps TYPENAMEs of edited data structures to the functions which can parse them.

ParseInfoUnknown: a function name
This is the function called to parse any data structure whose TYPENAME doesn't appear in ParseInfo.

DefaultFont, ItalicFont, KeywordFont: font descriptors
These are the fonts to be used when formatting.

DefaultLineSkip: an integer

If the linearization procedure doesn't specify the vertical spacing between lines this value will be used.

ReadTable: a READTABLE
This is the table used to determine the syntax of keyboard input and the interpretation of command characters.

SpaceWidth: an integer
This is the default space to leave between adjacent items in lists, etc.

DefaultIndent, MinIndent, MaxIndent, MaxWidth: integers
These values control the formatting of list structures.

LParenString, RParenString, DotString, QuoteString: StringItems
To prevent repeatedly constructing StringItems for the standard punctuation symbols, they're cached in the environment and shared. These must be reconstructed if the fonts are changed.

EditNodes, in detail

(an annotated description of the EditNode data structure definition)

NodeType: an EditNodeType
The type of this node, providing the set of methods.

ParseMode: a litatom
The parse mode in which this node was parsed.

SuperNode: an EditNode
This node's super node; NIL if this is the root.

Depth: an integer
The depth of this node within the tree; the root has depth 0.

SelfLink: a WeakLink
To avoid building uncollectable circular structures, the linear form ends with WeakLink back to the node. To avoid consing WeakLinks, we cons one and cache it here.

SubNodeIndex: an integer
This is the index of this node within its super node's subnodes.

Structure: anything
This is the structure this node actually represents.

Changed?: a boolean
True if this node has been changed and will require relinearization. Most of the time, true iff this node is on the context's list of changed nodes.

InlineWidth, PreferredWidth, PreferredLLength, MinWidth, MinLLength: integers
The width estimates computed by this node's ComputeFormatValues method.

SubNodes: a list of EditNodes
The subnodes of this node.

LinearForm: a list of linear items
The linear form of this node, terminating in a WeakLink back to the node.

LinearThread: a list of linear items
The position of this node within its super node's linear form (hence a list whose CAR is this node)

Unassigned: anything
Available for use by this node's methods to cache any additional information they wish to record.

StartX: an integer
The horizontal position at which the linear form of this node begins.

RightMargin: an integer
The right margin used when formatting this node.

ActualWidth: an integer
The maximum horizontal offset of any part of the presentation of this node from the starting position.

ActualLLength: an integer
The offset of the end of the last line of this node's presentation from the node's starting position.

FirstLineLinear, LastLineLinear: Lines
The line on which this node's linear form begins, and the line on which it ends.

Inline?: a boolean (computed)
True iff FirstLineLinear and LastLineLinear are the same.

FirstLine, LastLine: LineStarts (computed)
The CARs of FirstLineLinear and LastLineLinear.

EditNodeTypes, in detail

(an annotated description of the EditNodeType data structure definition)

Name: a string or litatom
This is not used by SEdit, but provides a handy point of reference when debugging.

ComputeFormatValues, Linearize , ReParse , SubNodeChanged , SetPoint , ComputePointPosition , ComputeSelectionPosition , SetSelection , GrowSelection , SelectSegment , Insert , Split , Delete , Replace , CopyStructure , CopySelection , BackSpace : function names
These are the methods of this node type.

LineStarts, in detail

(an annotated description of the LineStart data structure definition)

NextLine, PrevLine: Lines
These lines immediately before and after this one in the linear form.

Node: an EditNode
This is the node in whose linear form this LineStart was generated.

LineAscent, LineDescent: integers
The maximum ascent and descent of any item on this line, and hence the dimensions of the line.

LineSkip: an integer
The vertical separation of this line from the line preceding it.

Indent: an integer
The amount by which this line is horizontally indented.

LineLength: an integer
The total length of this line (including indentation).

YCoord: an integer
The vertical position of the top of this line (including the LineSkip). Since we set up the window's coordinate system with (0,0) in the top left corner, YCoords are always negative or zero.

CachedY, CachedAscent, CachedDescent: integers
During relinearization, we may need to determine what the position and dimensions of a line were after we've changed them. When this is a possibility the old values are saved here.

CacheTime: an integer
When the old position and dimensions are cached, the current value of the context's RelinearizationTimeStamp is recorded here, so that we can quickly determine when the cached values are out of date.

LineHeight: an integer (computed)
The sum of LineSkip, LineAscent, and LineDescent.

BaseLineY: an integer (computed)
YCoord minus the sum of LineSkip and LineAscent.

NextLineY: an integer (computed)
YCoord minus LineHeight; should be equal to the YCoord of NextLine.

OldTop, OldBottom: an integer (computed)
If the cache values are up to date (comparing their time stamp with that of the context) use them; otherwise use the current values.

StringItems, in detail

(an annotated description of the `StringItem` data structure definition)

String: a litatom or string
The text to be displayed.

Font: a font descriptor
The font in which the text should be displayed.

PRIN2?: a boolean
True if the PRIN2-name of String should be used, rather than the pname (i.e. string delimiters will be displayed, command characters will be escaped).

Width: an integer
The width of this item when displayed.

EditPoints, in detail

(an annotated description of the `EditPoint` data structure definition)

PointNode: an `EditNode`
The owner of this point, i.e. the node in which insertion will take place.

PointIndex: an integer
Used by the PointNode to record the point's position within the node.

PointType: one of (`Structure`, `Atom`, `String`)
Determines how characters typed at this point are to be interpreted.

PointLine: a `LineStart`
PointX: an integer
The position in the window at which the flashing caret should be displayed while waiting for keyboard input.

EditSelections, in detail

(an annotated description of the `EditSelection` data structure definition)

SelectNode: an `EditSelection`
The owner of this selection.

SelectStart, SelectEnd: integers
Used by the SelectNode to record the selection's boundaries within the node.

SelectType: one of (`Structure`, `Atom`, `String`)
If typed input replaces this selection, this determines how it will be interpreted.

SelectStartLine, SelectEndLine: `LineStarts`
SelectStartX, SelectEndX: integers
The boundaries of the area to be highlighted when this selection is displayed.

DeleteOK?, ReplaceOK?: booleans
The use of these flags is not completely implemented yet.

LineBlocks, in detail

(an annotated description of the `LineBlock` data structure definition)

BlockStart: a position within the linear form
The start of the segment of linear form this block represents (the end is determined by the start of the next block).

BlockNewX: an integer
The horizontal position at which this block will be displayed.

BlockWidth: an integer
The width of this block when displayed.

NextBlock: a `LineBlock`
The next block in the sequence.

Bits?: a boolean
True if the segment of linear form represented by this block is already displayed on the screen.

BlockX, BlockBaseLine, BlockAscent, BlockDescent: integers
If Bits? is true, these fields give the current position of the block's presentation.