Node Types and Node Type Methods in SEdit

All SEdit editing operations and presentations are controlled by methods associated with classes of nodes in the edit tree. By defining a new class of nodes with appropriate methods, or modifying an existing class, SEdit may be configured for a wide range of editing tasks. This document describes the programming involved in defining such a class, using the basic Interlisp-D type definitions as examples.

Node Types and the Edit Tree

The edit tree is a data structure maintained by SEdit as a representation of the structure being edited. The tree is initially constructed by parsing the data structure to be edited as a hierarchical data structure. Each node in the tree corresponds to a part of the edited structure, either an instance of a lisp datatype or a combination of several data structures. For instance, atoms and strings are usually represented by separate nodes in the tree, but a sequence of several cons cells may be represented by a single list node. Instances of a single data type may be represented by a variety of node types, depending on the context in which they appear. All editing operations are defined in terms of these nodes. The definitions for editing Interlisp code define ten node types:

1. atoms (actually litatoms and numbers)
2. strings
3. lists (NIL-terminated sequences of cons cells)
4. dotted lists (sequences of cons cells terminated by something other than NIL)
5. CLisp expressions (if, fetch, iteration, etc.)
6. forms (lisp function calls)
7. LAMBDA expressions (also includes LETs, PROGs, etc.)
8. quoted structures (i.e. two element lists whose first element is the atom QUOTE)
9. unknown (any data type other than litatom, number, string, or cons)
10. root (a special node type for the root of the edit tree)

Node types 5 through 8 are special cases of type 3, which are recognized in some contexts.

Creating the Edit Tree

To create the edit tree, SEdit conducts a preorder traversal of the given data structure. At each step, it dispatches on the datatype of the structure and calls the corresponding function, which is responsible for building the rest of the tree by making a call to \\create.node and recursive calls to the parser (\\parse). Each node built will record (among other things)

- the node type (chosen by the parse function)
- the node's super node
- the node's subnodes (one for each recursive call to \\parse)
- the node's depth in the tree
- the structure which was parsed to create this node

Because the correct parsing of a node often depends on contextual information, the parser allows each parse method to pass an argument to its subnodes. For no particularly good reason this argument is called the *parse mode*. The interpretation of the parse mode is up to the individual parse methods. The default mode is NIL; most parse methods ignore any unrecognized mode. In addition to the default mode, the Interlisp code definitions use a couple of other modes (the atoms `Binding`, `BindingList`, and `KeyWord`).

Edit Node Types

The type information SEdit associates with each node is actually a set of methods, i.e. functions to perform various actions on nodes of that class. Each node type must provide methods to perform seventeen different actions. This is, of course, poor man's object oriented programming. One day in the grand and glorious future Interlisp-D will metamorphose into something which properly supports object oriented programming, whereupon SEdit will be rewritten. In the meantime, this works well enough.

The methods for node types fall into four groups:

- those used to generate the presentation of a node (2 methods)
- those used to place selections and points (6 methods)
- those used to effect editing commands (5 methods)
- those which perform various housekeeping functions (4 methods)

We'll consider each of these groups in turn, specifying in detail how the methods are invoked and what they are expected to do.

Presentations and the Linear Form

The visual presentation of the data structures is represented in SEdit by a structure called the linear form. The linear form is a sequence of presentation commands which produce the desired presentation. Presentation commands are very simple, and there are only four of them:

- insert a string of characters in a given font
- insert horizontal space
- insert a given bitmap
- start a new line, with a given indentation and vertical separation from the previous line

The `Linearize` method for a node type must construct the sequence of presentation commands for a node of that type, by inserting the appropriate commands in the correct order. It may call the `Linearize` method of each of its subnodes. For instance, a very simple algorithm for linearizing lists might be:

```
output "(" in the default font
for each subnode
    if this isn't the first subnode, output some space
    linearize the subnode
output ")" in the default font
```

The actual algorithm used by SEdit is somewhat more complicated; for instance, lists usually don't fit all on one line, so it inserts line breaks at appropriate points.

To format structures such as lists properly, it's important to know how much space the presentation of each subnode will occupy. This is a problem, since the amount of space the presentation of a structure occupies often depends on the amount of space available when it is presented. SEdit deals with this by computing *width estimates* for each node. These are five values computed by each node for use by its super node:

a) if the node can be presented on a single line, the width of that presentation

b) the width of the most readable presentation of this node (its preferred presentation)

c) the width of the narrowest possible presentation of this node

d) the length of the last line of the preferred presentation of this node

e) the length of the last line of the narrowest possible presentation of this node

These width values allow the super node to make reasonable decisions on indentation and line breaks. The last line lengths are important because the super node may append additional material to the last line of a subnode's presentation, and would like to know how long the resulting line would be. These five values are stored as extra fields in each node, and referred to as the `InlineWidth` (NIL if the node can't be presented inline), `PreferredWidth`, `MinWidth`, `PreferredLLength`, and `MinLLength`, respectively.

In much of the program these width estimates are called *format values*. I've started calling them width estimates as a reminder that they are not required to be accurate, but should be quick to compute. It is important that a node's width estimates not indicate that it can be presented inline when it actually requires several lines, but other width values can be incorrect without breaking the program. On the other hand, incorrect width estimates will often lead to less than optimal presentations. The width estimates computed by the Interlisp code definitions are always correct (at least, that was the author's intention).

Each node type defines a method called `ComputeFormatValues` which will fill in the width estimates of a given node. To do so it may examine the structure the node represents and the format values of its subnodes. For instance, the ComputeFormatValues method for type list looks approximately like this:

```
InlineWidth is
    if all of the subnodes can be presented inline
        the sum of the InlineWidths of the subnodes,
            plus the width of two parentheses,
            plus the width of n-1 blanks
    else
        NIL
PreferredLLength is
    the PreferredLLength of the last subnode,
        plus the preferred indentation,
        plus the width of one parenthesis
MinLLength is
    the MinLLength of the last subnode,
        plus the width of two parentheses
PreferredWidth is
    the largest of
        the PreferredWidth of the first subnode,
            plus the width of one parenthesis
        the maximum PreferredWidth of the other subnodes,
            plus the preferred indentation
        the PreferredLLength of this node
MinWidth is
    the largest of
        the maximum MinWidth of any subnode,
            plus the width of one parenthesis
        the MinLLength of this node
```

Note that these calculations assume the node has at least two subnodes — special case rules are needed for fewer. Also, the formatting rules assumed are that

- the inline presentation of a list is an left parenthesis, followed by the subnodes separated by blanks, followed by a right parenthesis; all of the subnodes must present inline
- the preferred presentation of a list is an left parenthesis, followed by the subnodes, where each subnode after the first is on a new line indented by the preferred indentation; the final subnodes is followed by a right parenthesis
- the minimum presentation is similar to the preferred presentation, except that subnodes are indented only by the width of the left parenthesis

(Indentations are always non-negative, and specified relative to the horizontal position of the start of the node's presentation; thus the presentation of a node always occupies the quadrant to the right and below the point at which its presentation starts.)

The linearization method is given two arguments in addition to the node to be linearized and the usual context information. The first is a right margin, which it should try to keep its presentation within (if possible). This value is actually stored as another field in the node. The second argument is an index. In some situations, SEdit may request that the linear form of a node be recomputed starting part way through. In this case, the index given will be the index of a subnode, and the method should output just that part of the node's linear form which follows the presentation of that subnode.

Linearization methods are permitted to call five lisp functions to create the node's linear form:

(**\\output.string** context string prin2? font)
Insert string in the specified font at this point in the linear form. context is the usual context encapsulation. string may be any Interlisp object; its standard printed representation will be used. if prin2? is true, the PRIN2 representation of string will be used instead.

(**\\output.space** context width)
Insert a horizontal space of the specified width at this point in the linear form.

(**\\output.bitmap** context bitmap)
Insert a bitmap at this point in the linear form. It will be aligned with the line's baseline.

(**\\output.cr context** indentation lineskip)
Start a new line, with the specified indentation (relative to the start of this node's presentation) and separation from the previous line.

(**\\linearize** subnode context right.margin)
Insert the linear form of a subnode at this point in the linear form. Its linear form should not extend beyond right.margin (if right.margin is not specified, it will default to this node's right margin).

Pointing and Selecting

When the user uses the mouse to place the caret point and/or select part of the edited structure, SEdit is faced with the task of mapping the mouse's (x,y) coordinates to the appropriate structure description. These descriptions take the form of datatypes called EditPoints and EditSelections. An EditPoint records

- the node which owns this point (i.e. the one which will be informed if something is inserted here)
- an index, which the owning node may use to record arbitrary information about the point's location
- a type, one of Structure, Atom, or String, indicating how characters typed here will be interpreted
- a line (in the linear form) and x offset within that line, indicating where the caret should be positioned in the window if this point is displayed

EditSelections are similar, except that they have two indices (since selections may cover a sequence of substructures), and describe two positions (bounding the part of the linear form which should be underlined to display this selection).

The responsibility for translating mouse positions to points and selections is shared between the kernel and the type methods. The kernel determines which part of the linear form is being pointed at, and then asks the node which produced that part of the linear form to determine the point or selection. Sometimes the node will decide that in fact its super node or one of its subnodes should really be responsible, and if so it may pass on the request to them. For instance, if the user tries to insert characters at the beginning of an atom, they may actually point to the space between that atom and a preceding structure. The node which output that space was the atom's super node, i.e. the enclosing list. When it receives a request to position a character point in the space between two

subnodes, it should realize that this was likely an attempt to actually edit one of the subnodes, and pass the opportunity to them.

The first method in this group is `SetPoint`. It is called with (among other things) the context (all methods are passed the edit context; from now on we'll stop mentioning it), the node, the index of the linear form item in which the mouse is positioned, the x offset within that item, and the type of point requested (i.e. the choice of mouse button used to place the point). It must fill in the fields of the point, or call some other node's `SetPoint` method to do it. It can call \\set.point.nowhere to indicate that no point is near this mouse position, and thus the point returned should not allow input.

There are two other calling sequences a `SetPoint` method may be invoked with, to allow `SetPoint` requests to be readily passed between nodes. A subnode may pass a `SetPoint` request to its super node, indicating that the point is to be placed either immediately before or immediately after itself. For instance, this is what atomic structures do when asked to insert structure. To do this, the subnode calls \\punt.set.point, passing a flag to indicate whether the point should be before or after this node. Conversely, a node may pass the `SetPoint` request to one of its subnodes, indicating that the point is to be placed at the beginning or end of that subnode (as in the example with left-clicking the space in a list, above). The `SetPoint` method can determine which case it is being asked to handle by examining its arguments.

`SetPoint` methods usually calculate the position of the point when they set it. If the linear form changes, or the point is set by some other means, it becomes necessary to recompute the position of the point so that the caret may be displayed. The `ComputePointPosition` method of the node owning the point is responsible for filling in the line and x offset values on request.

Similarly, the `SetSelection` method and `ComputeSelectionPosition` methods determine the current selection, given the mouse position. These are very similar to the corresponding methods for points, and may call \\set.selection.nowhere or \\punt.set.selection. \\set.selection.me is a useful default `SetSelection` method; it simply sets the current selection to this node.

There are two additional methods related to selections. `GrowSelection` is called when the user uses multiple mouse clicks to select structures. If the mouse handler detects a multi-click sequence, it calls the `SetSelection` method for the first click, and `GrowSelection` for each subsequent click. The `GrowSelection` method of the owner of the current selection is responsible for enlarging the selection to include the next enclosing level of structure. \\grow.selection.default is the `GrowSelection` method for most types; it simply calls \\punt.set.selection, causing the super node to become the new selection owner.

The `SelectSegment` method handles right-button mouse actions, which extend the current selection to include the item pointed to. If the mouse is pointing at part of the linear form of the owner of the current selection, the `SelectSegment` method of that node will be called to fix up the selection. Otherwise, the deepest common super node of the node selected and the node pointed to will be asked to determine the new selection, given which of its subnodes the selection and mouse are in.

Editing Operations

When an editing operation (such as the deletion or insertion of material) is performed, an appropriate method is called for the affected node, and this method is expected to fix up both the tree and the actual structure being edited. These operations are all defined in terms of points and selections, and the method invoked is that of the owner of the point or selection.

The `Insert` method takes a previously created point owned by this node and either a string of characters or a list of nodes which are to become subnodes. The appropriate changes are to be made to the tree and structure, and the point should either be adjusted to be after the inserted material or cancelled, as appropriate. Some appropriate functions to call are:

(**\\note.change** node context)
: This node has changed in some way which affects its presentation. Its width estimates should be recomputed, and the linear form update appropriately.

(**\\subnode.changed** node context)
If the structure resulting from an editing operation is no longer EQ to the original, \\subnode.changed will inform the node's super node that it must make the appropriate updates. For instance, if a character is inserted into a litatom it actually becomes a new litatom. No change actually takes place in either atom, but the structure which contained the original atom must change to refer to the new one.

The Delete method takes a previously created selection and deletes the selected material. It returns a flag indicating whether in fact the material can be deleted; many data structures do not allow material to be deleted. It may also be asked to set the caret point so that inserted material will replace that which has just been deleted.

The Replace method replaces the selected material with either a string of characters or a list of nodes (depending on the type of selection). \\replace.default is a default implementation of this method, which does a deletion followed by an insertion. This is satisfactory for many data structures, but fails for those which do not allow deletions (e.g. quoted structures).

The Split method is only required for those types which allow character points (i.e. atoms and strings). When a delimiter (e.g. blank or cr for atoms, double quote for strings) is inserted at such a point, the node's Split method will be called. The usual behavior is to change the caret point to a structure point, and to separate the structure into two if the point was not at the beginning or end of the structure.

The BackSpace method implements the action of the backspace key. Given the caret point, the method is expected to make the appropriate deletion and adjust the point accordingly. Sometimes only the latter is necessary; for instance, if the point is positioned immediately after a list, backspace merely moves the point so that it is after the last element of the list (this corresponds to the action of the right parenthesis moving the point to after the parenthesis).

Miscellaneous Methods

There are four other methods for node types to implement, which are invoked by SEdit when the effects of some editing command might be important to a node other than that directly affected. The first of these, SubNodeChanged, was mentioned above; when the structure associated with a subnode is replaced with one which is not EQ to the original one,the super node's SubNodeChanged method will be invoked so that it can fix up its structure appropriately.

Two other methods are used to implement copying and moving structures. When a move or copy selection is made, the CopySelection method of the node owning the selection will be invoked, and passed the selection, a flag indicating what type of selection was made, and a description of the destination. The destination will be either the context of this or another SEdit process, or NIL, indicating that the copy or move is being made to a non-SEdit process, so the material should just be BKSYSBUFed. The default method \\copy.selection.default provides an implementation of this method suitable for most applications. If this is used, the CopyStructure method must be defined. CopyStructure is called with a node which has been constructed as a copy of an existing node. Because the copy operation should create new structure rather than just creating another pointer to the same structure, the CopyStructure method must fill in the Structure field of the node with the appropriate newly created structure. The Structure fields of its subnodes will already be copies of their structures, so all that is usually required is to create a new data structure out of these.

When material is moved or copied to other parts of the structure, the position at which it is inserted may imply a different parsing of the structure than that from which it is taken, since the parsing is context dependent (remember the parse mode?). In such a case, the new super node may ask the node to reparse itself, using the ReParse method. This usually involves minor adjustments to presentation, although in the worst case it may involve completely parsing the structure again from scratch.