SEdit Internal Documentation

How Relinearization Works

The process by which SEdit optimizes formatting recomputation is strange and wonderful, so this is a long overdue attempt at explaining it.

We will start with a quick recap of SEdit's formatting model and the responsibilities of three node type methods: assign-format, compute-format-values, and linearize. We then describe the assumptions SEdit makes about when these have to be redone, and then describe the algorithm it uses to achieve this. We'll only go as far as getting the linear form fixed up; the step from there to updating the bits in the window is a whole 'nother story...

The formatting model

Linearization is the generation of a sequence of format tokens (space, string, bitmap, line break) from the internal tree representation of the data structure being edited. Doing a reasonable job for complex hierarchical structures involves a large number of constraints; SEdit uses a three pass algorithm to get its results:

formats: first, the presentation of a data structure often depends on the context in which it appears. for instance, a list occurring as the second element of a list beginning with let gets special treatment. another example is collapsing lists at a nesting level cut-off (actually, now that i think about it this would be much better done at parse time). SEdit currently does the first, but not the second (it ought to do both). to achieve this, each node can pass a 'format' to each of its subnodes. exactly what constitutes a format is arbitrary; it's up to the parent and child nodes to agree on what information will be passed (all current methods ignore format information they don't understand). at present lists pass their sublists list-format structures describing the appropriate presentation, and sometimes pass the atom :keyword to atomic elements which are to be printed in boldface.

each node type provides an assign-format method, which is called when the format of a node changes, and is repsonsible for propagating that change by calling assign-format on each of its subnodes; hence width estimates propagate from the top down.

width estimates: second, the presentation of a composite data structure will often depend on the size of its components. once the format information has been propagated to a node it will be asked to provide estimates of the size of its presentation, so that the nodes above it in the tree can plan their presentation intelligently. (note: unfortunately, most of the code calls width estimates "format values"; hence the method responsible is called compute-format-values). each node compute two numbers: inline-width and preferred-width. the inline width is the estimated width of this node's presentation assuming that there is room to fit it all on one line, or nil if the node's presentation will always span multiple lines. the preferred width is the width of the node's presentation assuming it will break at convenient spots. in computing these estimates a node needs access to the width estimates of its children; hence width estimates propagate from the bottom up.

linear form: finally, each node is asked to compute its linear form, by outputing a sequence of format tokens interspersed with the linear forms of its subnodes. the linearize method is told the horizontal position at which it is to begin, and the horizontal position of the right margin (which ought to try to stay within, but it's free to ignore). to get the best formatting linearization procedures generally have two formats: a preferred, reasonably-indented format and a tighter "miser" format, and choose the miser format whenever the width estimates indicate that the preferred format won't fit. each node makes this choice independently. the linear form is computed top down.

Incremental changes

The three-pass computation described above places (relatively) simple requirements on the methods, and suffices for a one-shot presentation. However, this is insufficient for SEdit; the presentation changes after each character typed, and repeating the above computation each time over the whole tree would

clearly be unacceptably expensive. Instead, SEdit tries to determine the regions of the tree whose presentation might have changed given the editing operations performed, and calls the presentation methods only when the results might be different.

formats: in determining the format for its subnodes, a node is only allowed to base its decisions on its type, its own format, and the structure of it and its immediate subnodes. thus a list can change the format of its subnodes if it is edited, or one of its immediate subnodes is edited, but not if a nested subnode is edited. thus we only need to rerun a node's assign-format method if

- it was edited, or
- one of its subnodes was edited, or
- the assign-format method of its supernode was run (for one of these three reasons) and it assigned a different format to this node than previously

width estimates: the width estimates of a node must be determined based on its structure, its format, and the width estimates of its subnodes. thus we only need to rerun a node's compute-format-values method if

- it was edited, or
- its format changed, or
- the compute-format-values method of one of its subnodes was run (for one of these three reasons) and it changed that node's width estimates

linear form: the linear form of a node also depends on its structure, its format, and the width estimates of its subnodes. in addition, it can depend on the space between its starting horizontal position and the right margin. thus we only need to rerun a node's linearize method if

- it was edited, or
- its format changed, or
- the width estimates of any of its subnodes changed, or
- changes to one of its supernodes has caused its presentation to begin at a different horizontal position relative to the right margin

Relinearization

to implement the optimizations suggested above, SEdit must first know what parts of the tree have changed since it was last presented. therefore all procedures which change the tree structure are responsible for calling note-change on any node they change. also, all nodes added to the tree are marked as needing re-presentation. note-change inserts the changed node into a queue (the changed-nodes field in the edit-context) which is kept sorted by increasing depth. relinearize-where-necessary then implements the following algorithm:

> [inconsistency: note-change actually maintains the queue in order of decreasing depth (despite its comments to the contrary), and relinearize-where-necessary reverses it before it looks at it. this turns out to be fine, since it wants the queue in decreasing depth for the next step. ought to fix the comments though...]

1. for each node on the queue (from top to bottom), assign-format to its super node and add it to the queue (unless it's already in the queue) and then assign-format to it.
2. for each node now in the queue (from bottom to top) recompute its width estimates, and if they've changed push its super on the queue (where it will be picked up later in this step); if they don't change this is a point to start relinearizing from so push it onto another queue.
3. finally, reverse the new queue created in step 2 (so that it's now ordered from bottom to top) and for each node on it check to see that none of its super nodes are awaiting relinearization — if one is found, mark all the nodes between them changed so that the super's relinearization (yet to come) will include this node — otherwise just relinearize it.

relinearizing a node is guaranteed to call its linearize method. In addition, it will call the linearize method of any subnode which (a) has changed, or (b) has been moved (relative to the right margin) so

that it's linear form is no longer likely to be appropriate (the test for this is at the beginning of generate-linear-form) (and so on recursively into its subnodes). (and as i mentioned before, there's a whole extra story about how changes to the linear form are used to determine changes to the screen; this is yet to be documented at all). when relinearization of the original node is complete, an additional test is made: if the last line of the new linear form is a different length than the last line of the old linear form, this relinearization continues with the linear form of the super node, starting after the node just linearized (since the super may have made linearization decisions based on where that node ended). this process terminates when (a) a node's new linear form ends at the same horizontal position as before, or (b) the top of the tree is reached.