SEdit Internal Documentation

The formatting methods for lists

The assign-format, compute-format-values, and linearize methods of lists are now driven by tables encapsulated in list-format objects, allowing easy special formatting for particular lisp forms. This note primarily documents the format of list-format objects, and along the way mentions how they are used to implement those three methods. (note that dot-lists currently aren't handled by this mechanism)

Finding the right list-format

assign-format-list finds an appropriate list-format to control the list's formatting, using one of four options:

if the list is assigned a format which is a list-format object, use that;
else if the car of the list can be found in a list-formats table,
    use the associated object from that table;
else if the car of the list has a known clispword property,
    use the clisp list-format
else use the default list-format

since the same list-format object will be needed for computing width estimates and and linearization, assign-format caches it in the unassigned file. there are two types of list-formats; standard and non-standard. standard list-formats contain information to control the standard list formatting methods. nonstandard list-formats are an escape mechanism for situations where the required formatting is too hairy for the standard methods; they simply provide replacements for the assign-format, compute-format-value, and linearize methods. this is implemented by all list-formats having a non-standard? field, and list-formats whose non-standard? field is t having 3 additional fields: set-format-list, cfv-list, and linearize-list. there's not much else to say about non-standard list-formats, except that at present the only one is the format for clisp. for the rest of this document we'll talk about standard list-formats.

A general rule

several list-format fields contain lists of entries which correspond to the list node's subnodes. these lists all have roughly the same form:

(*last first second ... nth*)

where *first* is the information to be used for the formatting the first subnode, *second* the information for the second, ..., *nth* the information for the nth and all subsequent nodes, except that *last* is the information to be used for the last (some forms (e.g. `il:selectq`) have special formatting for the very last item). n depends on how many of the nodes need special formatting. `lambda`, for instance, uses lists of the form (a b c a) — thus the first element is formatted using b, the second with c, and all subsequent ones with a. the default list format uses lists of the format (a), since all subnodes are formatted similarly. `il:selectq` uses lists of the form (a b c d) — so b applies to the `il:selectq` atom, c to the evaluated expression, d to each clause, and a to the final otherwise clause.

the other important fact about these lists is that they're blind to comments; comment subnodes are ignored when figuring out which information goes to which subnode, and the comments themselves are formatted by hardwired rules.

Assigning formats

assign-format-list uses the list-formats field of the list-object, which is a list in the format described above. each element of the list is the format to assign to the corresponding subnode (nil, :keyword, or a list-format object — or :recursive, which means to assign this node's list-format to its subnode).

Linearizing

each list-format contains descriptions of two possible presentations of that list — a “preferred” format and a “miser” format, in the list-pformat and list-mformat fields of the list-format. linearization decides which presentation is appropriate, based on the width estimates of this node (see below for a description of where they come from) and the horizontal space available.

the list linearization will always have the form "(" *subnode* <space> *subnode* <space> ... *subnode* ")", where <space> is either a one-space-wide horizontal movement, or a line break (with some indentation) (we assume the list contains no comments for now). thus, the problem of formatting the list reduces to specifying, for each element after the first, whether to space or break, and if you break, how much to indent. using a list in the above format, a specification is given for each subnode after the first as to how this decision is to be made.

these spacing specifications are expressions in a simple language. each expression must at a minimum specify the indentation if a line break is made here. the simplest expressions are just that — an integer, giving the indentation (in pixels) from the opening "(". this expression won’t break unless it has to, but if it does it will use that indentation. to force a break, an expression of the form `(break . exp)` is used (where *exp* is a nested expression). to line up the presentation of subnodes, spacing specifications can set a tab stop and then later position relative to it. `(set-indent . exp)` works just like *exp*, except that after it has been determined where this subnode is to be positioned, the tab stop is set at that point. a later spacing specification of `(from-indent . exp)` means that if we break at this point, the indentation is to be taken relative to the tab stop. note that the order of parts in an expression isn’t important; `(break from-indent . 3)` has the same effect as `(from-indent break . 3)`.

the remaining types of expressions allow the formatting to depend on a variety of conditions, such as whether the previous node’s presentation fit on one line, or whether this node is atomic. they all have the form `(`*condition exp1* `.` *exp2*`)`, where *condition* is one of the atoms below, *exp1* is the spacing specification to be used if the condition holds, and *exp2* is the spacing specification to be used if it doesn’t. (remember that a spacing specification is interpreted to determine the space preceding a subnode; in the list below, "previous node" is the node before the one whose placement is being determined, and "next node" is the node whose placement is being determined).

`prev-inline`:	did the previous subnode’s presentation fit on one line?
`next-inline`:	does the next subnode’s width estimate indicate that it will fit on this line?
`next-preferred`:	does the next subnode’s width estimate indicate that it will fit in preferred format?
`prev-atom`, `next-atom`:	is the previous/next node atomic (i.e. has no subnodes)?
`prev-keyword`, `next-keyword`:	is the previous/next node an atom in the keyword package?
`prev-lambdaword`, `next-lambdaword`:	is the previous/next node a lambda keyword (&aux, &rest, etc)?
`whole-inline`:	do the width estimates of the node being formatted indicate that it will fit on one line?

Formatting comments

the preceding discussion assumed that the list being formatted contained no comments. the formatting of comments is completely automatic (i.e. out of the control of the list-format), primarily to simplify list-formats. they are formatted like other subnodes, except that they don’t use the spacing specifications. single-semi comments are positioned at a fixed horizontal position (at the end of the current line if it isn’t too long, otherwise on a new line). triple-semi comments always start on a new line, at the left margin. double-semi comments are a little trickier; they start at the current tab stop, after the spacing specification for the next node has been interpreted. this requires looking ahead in these cases, and determining what the result of interpreting the space specification will be. it would probably be worth figuring out simpler ways to do this.

subnodes following comments always start on a new line (i.e. their spacing specifications are interpreted as if they began `(break . ...)`).

Computing width estimates

cfv-list must compute two values: the width of this node if its presentation can fit on one line (or nil if it can't), and the width of this node in its "preferred" presentation. the second of these is computed by simulating the linearization process using the preferred spacing specifications, and always assuming the worst when evaluating the prev-inline, next-inline, next-preferred, and whole-inline conditions. the inline width is determined by an even simpler scheme: if all of the subnodes can go inline, and the list-inline? field of the list-format isn't nil, the list can go inline with width equal to the sum of its subnode's inline widths, plus the appropriate blanks and parens.