Modeless Structure Editing

In honour of Tony Hoare on his retirement from Oxford

Bernard Sufrin and Oege de Moor
Programming Research Group

1 Introduction

Tony is a stubborn colleague. He refuses to accept that any aspect of programming should forever remain complex. If such complexity seems to be present, then he refuses to believe that there can be no conceptual tool capable of managing it. His approach is to build a model of such disarming simplicity that the complexity seems, at first, to have been ignored. Only after further exploration does it become clear that he has begun to vanquish the complexity with the rich theory that follows from his simple model. We have benefited much from Tony's ability to build elegant models, and from his gentle (but stubborn) insistence that we try harder when it is clear to him that our own work could be further simplified. In this paper we honour Tony by building a simple model of structure editing to serve as a basis for the design of a family of editors.

Structure editors make it easy to perform edit actions in which structural units (commands, expressions, chapters, paragraphs) are manipulated. Unfortunately these editors can make it rather difficult to enter certain kinds of new structure. For example, from the earliest days of structure editors for programming languages users have complained about awkwardness in entering expressions with infix operators.

To illustrate this consider the expression

$$a * b + c * d$$

It takes 7 insertion actions, each invoked by a single keystroke, to enter this expression in an ordinary text editor. In a simple-minded structure editor, these elementary insertions are interspersed with tree navigation commands – the keystroke sequence might look something like this:

$$+ \downarrow * \downarrow \texttt{A} \rightarrow \texttt{B} \uparrow \rightarrow * \downarrow \texttt{C} \rightarrow \texttt{D}$$

(where $\downarrow$, $\rightarrow$ and $\uparrow$ respectively navigate downwards to the first child, rightwards to the next sibling, and upwards to the parent)

Of course there are structure editors that offer much better interfaces than this – indeed our own interest in such systems was reawakened by the advanced structure editor of the *Intentional Programming* system [10].

Some designers have attempted to solve the problem of unnatural input by having different modes in the editor: one for structural commands, and another for editing textual representations of structures [8]. In some editors the textual representation is "continuously" parsed as it is edited, in others it is parsed as text mode is left.

But, as is well known, spurious modalities in an interface can be confusing and irritating for users. So our goal here is to develop a *modeless* structure editor, where expressions such as $a * b + c * d$ can be entered just as easily as in a text editor.

The development will be in four stages: first we present a very primitive editor that requires a lot of navigation. Then we present a refinement that does away with the need for navigation in consecutive insertions. The price the user has to pay for such economy of expression is, however, high: all expressions have to be entered in prefix form. The next step, therefore, is to consider a model that takes account of operator precedences: again no navigation is necessary, but trees can be entered in left-to-right order. Finally, we show how the model is extended to deal with bracketing.

We are not the first to consider the systematic design of structure editors, and a recent survey of the field can be found in [13]. Our own understanding of the subject matter was shaped by [4, 8, 7].

The typeless editors we develop here permit the composition of structures that are syntactically incorrect with respect to the language being edited: they are no more concerned with syntactic correctness than an ordinary text editor would be. A more complete model would address the pragmatic details of applying soundness checks of various kinds, but we will not address these matters here.

For concreteness, we present our model as an executable Haskell program and fix on a programming-language-like object language. Program fragments are displayed in an upright typewriter font, and form part of the complete program that is contained within the LaTeX manuscript of this paper.

2 Trees and subtrees

The trees our editor manipulates are defined by

```
data Tree = Fork Label [Tree]
```

That is, a tree consists of a labelled *fork*, with a list of descendant trees. The type `Label` describes the various constructs of the language being edited. For example, when editing programs, typical labels might be `While`, `Declare`, `Plus`, `Times` and `If`, as well as identifiers and literals.

The following projection functions take trees apart:

```
label :: Tree -> Label
label    (Fork l _)  = l

children :: Tree -> [Tree]
children (Fork _ ts) = ts
```

[*In Haskell definitions the underscore pattern matches any value.*]

A tree is said to be *atomic* if it has no children:

```
atomic :: Tree -> Bool
atomic = null . children
```

We need a distinguished tree, with label `Hole` and no descendants, to stand for a node that remains to be filled in:

```
hole:: Tree
hole = Fork Hole []
```

All operations on trees maintain invariant the fact that `Hole`-labelled trees are atomic:

```
atomic t, if label t = Hole
```

[*Text in the slanted typewriter font, as above, is not part of the Haskell program.*]

The following function lists all descendants of a given tree in prefix order:

```
preorder :: Tree -> [Tree]
preorder t = t : concat (map preorder (children t))
```

Like most text editors, structure editors have a notion of *current selection* within the structure being edited. The current selection is where most of the action takes place, so it is important for us to find an appropriate declarative representation for a "tree-with-a-selection". The simplest kind of selection is a simple subtree, and one approach, taken in [3] and [11], and which we ourselves took at first, is to represent it as the containing tree (the *host*), together with a list of directions to the selected subtree (the *path*).

```
type SubTree = (Path, Tree)
type Path    = [Int]
```

This appears to have the virtue of simplicity: the path and the host are extracted by projections, and the selected subtree is extracted by

```
selected :: SubTree -> Tree
selected ([],          t)  = t
selected (n:ns, Fork _ ts) = selected(ns, ts!!n)
```

[*Here* `ts!!n` *returns the* `n`*th element of the list* `ts`.]

But operations on this representation must maintain the well-formedness invariant:

```
wfSub :: SubTree -> Bool
wfSub([],   t)          = True
wfSub(n:ns, Fork _ ts) = n<length ts && wfSub(ns, ts!!n)
```

Although this is not hard to achieve, it can make certain properties harder than necessary to verify, so we will look for a simpler representation.

In [12] a pair of sequences is used to represent a "sequence-with-a-current-position" without requiring an invariant. Using an analogous idea here leads to the following less obvious representation, which is more suitable for our purposes:

```
type SubTree = (Path, Tree)
```

A subtree `(p,t)` represents the selection of the tree `t` at the position indicated by the path `p`. A `Path` is a list of context `Layer`s. It represents the context as it is encountered when moving upwards from the selected subtree towards the root:

```
type Path  = [Layer]
type Layer = (Label, [Tree], [Tree])
```

Each `Layer` represents, as its name suggests, a layer in the context, and a tree `t` is embedded in the context layer `(l, left, right)` with the function

```
embed :: Tree -> Layer -> Tree
embed t (l,left,right) = Fork l (reverse left++[t]++right)
```

[*The ++ operator catenates lists. The reversal of* `left` *is a consequence of the asymmetry of lists in Haskell and the fact that we need easy access to the last element of the left context. Tony would doubtless say that we have taken a needless refinement step here in order to make our abstract model more computationally efficient, and he would be right.*]

The path and the selected subtree are extracted by projections:

```
selected :: SubTree -> Tree
selected  (_, t) = t

path :: SubTree -> Path
path (p, _) = p
```

and the host of a subtree is extracted by

```
host :: SubTree -> Tree
host (p,t) = foldl embed t p
```

[*The function* `foldl embed t` *starts with seed value* `t` *and traverses a list of context layers from left to right. At each step it embeds the result so far in another layer of the context. In other words:* `host`$([l_1, l_2, ..., l_n], t)$ = `embed`(...(`embed` (`embed` t l_1) l_2)...) l_n]

Finally, the function `root` selects a whole tree within itself:

```
root :: Tree -> SubTree
root t = ([],t)
```

The following equations relate `host`, `selected`, and `root`:

```
host    (root t) = t
selected(root t) = t
```

3 Navigation

Basic Navigation Operations

Navigation is the process of moving around a tree from one subtree to another. There are four basic navigation operations: `left` and `right` (for moving among the children of a given node) and `up` and `down` (for moving between children and parents).

The `left` operation is defined as follows:

```
left :: SubTree -> SubTree
left  ((lab, last:l, r):ls, sel) = ((lab, l, sel:r):ls, last)
left  st@((_, [], _):_, _)       = st
left  st@([], _)                 = st
```

That is, `left` moves to the left when possible, but stays put if it is already in a leftmost position, *i.e.* when the path is empty, or there is nothing to the left of the selected subtree in its parent fork. [*The notation* `st@...` *introduces the shorthand* `st` *for the pattern that follows.*]

```
leftmost :: SubTree -> Bool
leftmost ((_, [], _):_ ,_)  = True
leftmost ([], _)            = True
leftmost _                  = False
```

The operations `right` and `rightmost` are similarly defined. Note that `left` and `right` are almost mutual inverses:

```
left (right st) = st, if not (rightmost st)
right (left st) = st, if not (leftmost st)
```

This is an instance of a design pattern that crops up a lot in editors, namely that of *do/undo* pairs. This pattern is generally described as part of the *command pattern* [6], in the context of implementing "Undo/Redo" facilities. One simple way of implementing such facilities is to accumulate a list of editing operations and their inverses as the operations are applied, and traverse this list in reverse when undoing, and forwards when redoing. Thus an important, though secondary, design consideration for many of the editing operations we introduce below will be that their inverses should be straightforward to calculate at the point of application.

An upwards movement takes us from a subtree to its container tree, so (like `host`) it involves embedding the selected tree in a context layer.

```
up :: SubTree -> SubTree
up (layer:above,t)       = (above, embed t layer)
up st@([],_)             = st
```

We have made this operation total by specifying that nothing is changed when the subtree is in its topmost position.

```
topmost :: SubTree -> Bool
topmost ([], _)          = True
topmost _                = False
```

The operation `down` and predicate `bottommost` are defined in the obvious way:

```
down (p, Fork lab (t:ts))= ((lab, [], ts):p, t)
down st@(_, Fork _ [])    = st

bottommost (_,Fork _ []) = True
bottommost _             = False
```

Note that we always go down to the *first* descendant. Consequently `down . up` is not the identity function. But we have the following more restricted equalities

```
down (up st) = st, if not (topmost st) && leftmost st
up (down st) = st, if not (bottommost st)
```

Before discussing composite navigation operations, it is worthwhile noting that we could have endowed some of the basic operations with better inversion properties (*i.e.* without side conditions). For example, `left` could wrap round from the first child to the last child, and dually for `right`. The resulting operations would, however, be awkward to use in practice.

Composite Navigation Operations

We build more interesting, composite navigation operators from the four basic ones. For example, the `next` operator takes a subtree and returns the next subtree in the prefix ordering:

```
next :: SubTree -> SubTree
next st = if bottommost st then rightup st else down st

rightup :: SubTree -> SubTree
rightup =
  right . until (\st -> topmost st || not (rightmost st)) up
```

[*In Haskell, the expression* `(\x->E)` *denotes the function that maps x to E.*]

The operation `rightup` moves upwards until it can move rightwards, or reaches the root of the tree. Note that we exploit the fact that `right` is idempotent on the root.

The connection between `next` and the prefix ordering can be made explicit:

```
preorder
=
map selected . takeOneWhile (not . topmost) . iterate next . root
```

That is, to find the preorder traversal of a tree, first make it into its own subtree using `root`. Then repeatedly `next` until the topmost node is reached again. The function `takeOneWhile`

`p` returns the longest non-empty prefix of its argument all of whose elements satisfy the predicate `p`. The result is a list of subtrees, which are projected to ordinary trees by `selected`.

We sometimes need to traverse the tree in prefix order until we reach a subtree that satisfies a given criterion:

```
nextSuchThat :: (SubTree -> Bool) -> SubTree -> SubTree
nextSuchThat p st =
  if p st' then st' else st
  where
  st'  = until (\ st -> topmost st || p st) next (next st)
```

The search stops if a topmost node is reached, but note that

```
nextSuchThat p st = st
```

only if `st` has no proper `next` successor that satisfies `p`.

4 Modifying Trees

The simplest way to modify a tree is to replace the current selection by a new tree:

```
replace :: Tree -> SubTree -> SubTree
replace s (p, t) = (p, s)
```

Note that this is a destructive operation: the selected tree `t` is lost. On the other hand, the inverse of `replace` is easy to calculate at the point of application.

```
replace (selected st) (replace s st) = st
```

The `replace` function is not intended to be invoked directly by users of the editor, but participates as a primitive in the design of user-invokeable operations. This is not to say that the editor we deliver will be without the functionality of a "replace" operation. Such an operation can only plausibly be defined in a context where it is possible to for the user to compose (at the keyboard) the tree which will do the replacement (*i.e.* the first argument to `replace`), and we have not yet reached such a context.

Insertion

In order to facilitate the insertion of nodes into the tree we require every construct in the language being edited to have its own template — describing an "unfilled" tree that represents such a construct. We stipulate that templates always consist of a `Fork` whose immediate descendants (if any) are holes. The function `template` maps labels to the corresponding unfilled trees.

```
template :: Label -> Tree
```

For example, the template of a conditional expression might be defined by

```
template If = Fork If [hole,hole,hole]
```

To enter a labelled node in the tree, we insert the corresponding template at an appropriate place in the tree.

```
insert :: Label -> SubTree -> SubTree
insert = treeinsert . template
```

The obvious first approximation to this is to replace the current selection, if it is a hole, by the template.

```
treeinsert :: Tree -> SubTree -> SubTree
treeinsert t st =  replace t st
```

But the initially selected expression is obliterated by this action. If it was a hole, then this behaviour is entirely appropriate — but it seems unfortunate to "lose" a substantive expression by making an insertion. A more useful, interpretation for insertion at a non-hole would be to make it "lossless" when this is reasonable, and the function we define next does this. It replaces the selection with the insertion, moves downwards if possible, then inserts and selects the original selection.

```
treeinsert :: Tree -> SubTree -> SubTree
treeinsert t st = if   atHole st
                  then replace t st
                  else (replace (selected st) . down . replace t) st
```

With this definition we can, starting with the empty tree, enter the expression $a + (b + c)$ with the sequence of commands

```
insert A, insert Plus, right,
insert B, insert Plus, right,
insert C
```

Now insertion is truly "lossless". Indeed inserting an atom at a non-hole is lossless because, perhaps counterintuitively, it has no effect at all:

```
selection(treeinsert t st) = selection st, if not (atHole st)
```

In other words, in order to replace a non-hole with an atom, a user first has to transform the non-hole into a hole.

At first sight we have here a conflict between the simplicity of the primitive editing functions and the convenience of users. It would be temptingly easy to resolve the conflict in favour of convenience by changing the `treeinsert` guard:

```
treeinsert t st = if atHole st || atomic t then replace t st ...
```

but in order to forestall catastrophic editing errors we would be forced to add "Undo" machinery and/or a scratchpad holding recently deleted material. At this stage of the editor development this would be inappropriate so we have adopted the simple version.

Now we have a lossless `treeinsert`, but the behaviour of the resulting `insert` operator is still not very convenient. One must take care to ensure that the right expression is selected before each insertion, and at least one navigation step is necessary after the insertion of each infix operator.

Before developing a better solution we will pay some attention to deletion, which should, in a sense, be the inverse of insertion. It is not possible to propose a uniform left-inverse for `treeinsert t` since it isn't injective, but, as we shall see, there are ways of using simple deletion functions to achieve the same goal.

Deletion

The most obvious deletion operation replaces the current selection with a hole.

```
kill :: SubTree -> SubTree
kill = replace hole
```

Another useful operation promotes the selection to replace its parent, and is idempotent at the root:

```
promote :: SubTree -> SubTree
promote st = (replace (selected st) . up) st
```

Given these operations, we can easily compute the inverse of a tree insertion

```
kill     (treeinsert t st) = st, if  atHole st
promote  (treeinsert t st) = st, if  not (atHole st) && not(atomic t)
id       (treeinsert t st) = st, if  not (atHole st) && atomic t
```

The `promote` and `kill` operations, which are useful in their own right, will themselves need to be inverted. We can easily undo the effect of a `kill` (but note that the complete selection must be preserved in order to implement the inverse)

```
replace (selected st) (kill st) = st
```

Undoing the effect of a `promote` is a little trickier (but note that only the last layer of the context needs to be preserved in order to implement the inverse)

```
graft (situation st) (promote st) = st
```

where

```
situation :: SubTree -> Path
situation    (c:cs, t) = [c]
situation    ([], t)   = []

graft ::     Path -> SubTree -> SubTree
graft        cs (p, t) = (cs++p, t)
```

We have now completed the design of our first structure editor. Our next step will be to see how tree navigation can be avoided during expression entry.

5 Insertion without Navigation

In the present editor we have to navigate rather a lot to enter an expression such as $a * b + c * d$, especially if it is entered in prefix form:

```
insert Plus,  down,
   insert Times, down,
        insert A, right, insert B,
   up,
right,
   down,
   insert Times, down,
        insert C, right, insert D
```

Of course the sequences of navigation operations merely move us to the next hole to be filled. The solution seems fairly obvious: after each insertion look for the next available hole. That is, we define

```
enter :: Label -> SubTree -> SubTree
enter l = nextSuchThat atHole . insert l
```

The above example can now be entered by

```
enter Plus,
      enter Times, enter A, enter B,
      enter Times, enter C, enter D
```

Even though this is a lot nicer, entering large expressions in prefix order is still irritating. Of course our primitive editor already allows one to enter expressions in infix order — that was the point of inserting into non-hole subtrees. But such entry still requires tree navigation: to enter the above expression in infix order, for example

```
enter A, enter Times, enter B,
up, enter Plus,
enter C, enter Times, enter D
```

6 Precedence Parsing

To make it easy to enter expressions like $a * b + c * d$ in a natural fashion, without tree navigation, we need to take account of operator precedences when inserting new trees. The insertion operation should, as it were, perform on-line precedence parsing — *on-line* because we want our editor to be modeless.

Of course precedence parsing is a very well studied subject. In a seminal paper, Robert Floyd [5] showed how to perform precedence parsing in linear-time in a bottom-up fashion. His algorithm retains a stack of trees, which form the right-hand spine of the resulting parse tree. There is an obvious way of taking advantage of Floyd's algorithm in the present context: simply treat the path as the parsing stack – at least the prefix of the path that is a right-hand spine. Readers who are unfamiliar with precedence parsing may wish to consult a more modern presentation, for instance [9, 2, 1].

Our new definition of label entry is

```
entry :: Label -> SubTree -> SubTree
entry l = enter l . reduce l
```

That is, before entering a new label, we reduce the current selection using precedences of the trees on the path, and the precedence of the label `l` whose template we are inserting. The function `reduce` moves up from the current subtree until it encounters an irreducible subtree – and that is the subtree at which `treeinsert` does its work.

```
reduce :: Label -> SubTree -> SubTree
reduce l = until (irreducible l) up

irreducible :: Label -> SubTree -> Bool
irreducible l st = atHole st                              ||
                   topmost st                             ||
                   (not . rightmost) st                   ||
                   (not . producable l . label . selected . up) st
```

Treating a hole as irreducible results in the template replacing the selection if it is a hole, and this means that prefix order input can still be employed. Otherwise a subtree is irreducible if it is topmost (because then we cannot move up further), or if it is not rightmost (because then we can no longer think of the path as a parse stack), or if the result of moving further up would be syntactically incorrect.

It is this last condition where the precedences, as well as associativity of operators in the object language, comes in:

```
producable :: Label -> Label -> Bool
producable op2 op1  = (op1==op2 && assl op1) || prec op1 > prec op2
```

The first disjunct deals with two occurrences of the same construct. The function

```
assl :: Label -> Bool
```

tells us whether a language construct associates to the left or not, and the function `prec`

```
prec :: Label -> Int
```

gives the precedence.

Our application of precedence parsing is not restricted to binary operators. Non-binary operators usually have very high or very low precedence. For example, by specifying

```
prec If  = 1000
```

we obtain the effect that conditionals do *not* extend to the right, whereas

```
prec Lam = -1
```

specifies that lambdas (as found in functional programming languages) do extend to the right.

We have nearly achieved the effect we wanted, namely the navigation-free, infix-order, input of expressions. If we use the obvious mapping from characters to their corresponding `entry` operations, then by entering `a*b+c*d`, character by character, into the empty tree, we obtain the desired tree representation of $(a * b) + (c * d)$, with the d subtree selected. Unfortunately the behaviour of our editor is still clumsy for expressions that would require bracketing, *i.e.* those in which an operator of lower priority appears as a child of an operator of higher priority. For example, to enter the expression $(a + b) * (c + d)$ we now need to resort to the prefix order `entry` sequence. $* + a\ b + c\ d$. The same problem manifests itself when entering non-binary constructs such as conditionals. Clearly this is undesirable, and we need a way of entering bracketed structures directly.

7 Brackets

We deal with brackets by refining our notion of *subtree*. Instead of considering a single, contiguous path, we shall consider a partition of the original path. Each component of the partition will correspond to a bracketed structure which has been opened but not yet closed. The revised type of subtrees is

```
type SubTree' = ([Path],Tree)
```

We can map new subtrees to old using the function:

```
flatten :: SubTree' -> SubTree
flatten (ps,t) = (concat ps,t)
```

We can lift the existing operations on subtrees to the new type by having them operate on the innermost unclosed bracketed expression (represented by the first partition component)

```
lift :: (SubTree -> SubTree) -> (SubTree' -> SubTree')
lift f (p:ps,t) = (q:ps,s) where (q,s) = f (p,t)

entry' = lift . entry
```

The navigation operators have similarly lifted definitions, but note that navigation outside the innermost *unclosed* bracketed expression is not possible.

An opening bracket adds a new, empty path component:

```
open :: SubTree' -> SubTree'
open (p,t)     = ([]:p,t)
```

A closing bracket extracts and selects the topmost expression of the current bracket structure, then moves right if possible:

```
close (p:ps,t) = right' (ps, s)
                 where s = host (p, t)
```

We have now reached the last stage in our design. As an illustration, consider the expression

```
(if a then b else b) + a
```

Using the symbol `?` to stand for the insertion of the label `If`, we can enter this with the sequence `?abb+a`. Recall that we defined the precedence of `If` to be very high: this accounts for the bracketing to the left. By contrast, lambdas have a very low precedence. To illustrate, consider the λ-expression $\lambda\, a \cdot (b + (c * d))$. Writing the symbol `\` for λ, this is entered with the sequence of insertions `\ab+c*d`. Of course it is also permissible to enter the same in prefix form, that is `\a+b*cd`. The point of our exercise so far has been to design a model of structure editors in which users can employ whatever order of entry seems to them to be most appropriate, without needing to switch modes either explicitly or implicitly.

8 Conclusion

Good interactive structure editors are few and far between. This is because, in the quest for user convenience, design decisions are taken which eventually lead to systems of such complexity that it is hard for anyone to form a mental model simple enough to let them easily understand the outcome of interactions. In this paper we have begun to outline the foundation of a model for structure editors which we hope will support a more rational approach to the taking of design decisions. By defining an appropriately abstract representation for an important part of the state of a structure editor, and defining its basic operations as mathematical functions, we have made it possible for the algebraic properties of the operations to be investigated straightforwardly. By investigating these properties when we introduce new editor operations we can either convince ourselves that a straightforward mental model of their behaviour can be explained to (or inferred by) users, or we can highlight behaviour that may be surprising or confusing. The constructive nature of the representation also makes it easy for the naturalness of the operations to be investigated empirically and directly.

One of the salient features of the model is the representation of a selection as a sequence of context layers paired with the selected subtree. This representation makes it easy to reason about selections – and as an added bonus its Haskell implementation is quite efficient. Moreover it can straightforwardly be generalised to support a style of selection in which a number of adjacent siblings are simultaneously selected. This style of selection is necessary when editing structures which use variadic nodes to represent trees formed of semantically associative operators – such as sequential composition (of commands in a programming language), and juxtaposition (of words, paragraphs, *etc.* in a natural language).

A further layer of design which we hinted at earlier introduces a scratchpad and cut buffer, and supports more radical structural changes and better undoability properties. The details are straighforward and only space considerations led us to exclude them.

Finally, we have ignored user-interface issues related to input of commands, to layout of the visible representation of the tree, and to navigation and selection using mouse gestures. We intend to pursue all these in a subsequent paper in the same constructive mathematical style.

9 Envoy

At this point the question of whether a structure editor with adequate functionality and performance can be implemented *directly* this way with today's functional programming technology remains open.[1] But whatever the answer to that question, the model can always be used as the starting point of a refinement process targeted on a more efficient implementation. Tony Hoare's pioneering expositions of the nature of that kind of refinement have given designers the liberty to investigate the properties of models independently of their intended implementations: we hope that he will continue to liberate others as effectively as he liberated us.

References

[1] A. Augusteijn, *An alternative derivation of a binary heap construction function*, Mathematics of Program Construction (R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, eds.), Lecture Notes in Computer Science, vol. 669, Springer-Verlag, 1992, pp. 368–374.

[2] R. S. Bird, *Lectures on constructive functional programming*, Constructive Methods in Computing Science (M. Broy, ed.), NATO ASI Series F, vol. 55, Springer-Verlag, 1989, pp. 151–216.

[1] But see `http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/sedistr.zip` for the Haskell source of a prototype which further explores this question.

[3] R. S Bird, *Introduction to functional programming in Haskell*, International Series in Computer Science, Prentice Hall, 1998.

[4] V. Donzeau, G. Huet, G Kahn, and B. Lang, *Programming environments based on structure editors: The MENTOR experience*, Interactive Programming Environments (D. Barstow, H. Shrobe, and E. Sandewall, eds.), McGraw-Hill, 1984, pp. 128–140.

[5] R. W. Floyd, *Syntactic analysis and operator precedence*, Journal of the ACM **10** (1963), no. 3, 316–333.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Professional Computing Series, Addison-Wesley, 1994.

[7] L. Meertens, S. Pemberton, and G. van Rossum, *The ABC structure editor: structure-based editing for the ABC programming environment*, Tech. Report CS-R9256, 1992.

[8] T. W. Reps and T. Teitelbaum, *The synthesizer generator: A system for constructing language-based editors*, Texts and Monographs in Computer Science, Springer-Verlag, 1989.

[9] B. Schoenmakers, *Inorder traversal of a binary heap and its inversion in optimal time and space*, Mathematics of Program Construction (R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, eds.), Lecture Notes in Computer Science, vol. 669, Springer-Verlag, 1992, pp. 291–301.

[10] C. Simonyi, *Intentional programming: Innovation in the legacy age*, Presented at IFIP Working group 2.1. Available from URL `http://www.research.microsoft.com/research/ip/`, 1996.

[11] J. M. Spivey, *A functional theory of exceptions*, Science of Computer Programming **14** (1990), no. 1, 25–43.

[12] B. A. Sufrin, *Formal specification of a display-oriented editor*, Science of Computer Programming **1** (1982), no. 3, 157–202.

[13] G. Szwillus and L. Neal, *Structure-based editors and environments*, Academic Press, 1996.