How to build a (quite general) linguistic diagram editor

Jo Calder
University of Edinburgh
Language Technology Group
2 Buccleuch Place
Edinburgh
Scotland
J.Calder@ed.ac.uk

Abstract

We propose a design for an editor, *Thistle*, which allows the construction and manipulation of a wide variety of linguistic (and other) diagrams and a general method for attaching semantics to such diagrams. This design represents a generalization of all systems proposed in the computational linguistic literature of which we are aware. We discuss theoretical and practical problems which have hindered the development of such systems to date and then indicate how our approach deals with those problems. We offer an illustrative range of applications for this design. The current implementation permits instances of the editor for linguistic theories such as HPSG, varieties of CCG, DRT, and various kinds of tree diagram. The display engine may be used to deliver diagrams via the World Wide Web. An appendix gives an almost complete specification for a significant class of diagrams.

All of the classes of diagram described or mentioned here are available as on-line demonstrations via:

`http://www.ltg.ed.ac.uk/software/thistle/demos/index.html`

1 Introduction

We propose a novel design, *Thistle*, for an editor for diagrams representing various kinds of linguistic information. We argue for the compromises we suggest as a trade-off between generality and usability. We demonstrate the latter property through a wide range of applications.

2 Motivation

Within linguistics and computational linguistics, diagrams play a crucial role in representing the content of theories (the use of trees to define inclusion hierarchies, for example), in standing as informal demonstrations of the truth of particular claims and, therefore, in sharing ideas with the community as a whole. Popular graphical devices include trees, attribute-value matrices (*AVMs*), e.g. (Pollard and Sag, 1994), and conventions such as those used in Discourse Representation Theory (DRT) (Kamp and Reyle, 1993). It has been clear for a number of years that the linguistic community would benefit from a general purpose "diagram editor" allowing users to construct and manipulate diagrams. A large range of uses exists for such a program, including the debugging of existing grammars, the construction and delivery of teaching and drilling materials and the production of diagrams for publication in some media or other. Even more generally, such a system offers a way of defining and interacting with documents with complex structure.

Why hasn't the community produced such an obviously desirable program? First, change has been a characteristic of the technical devices used in many branches of linguistics. Further, it seems in principle impossible to predict which graphical conventions are likely to gain currency in linguistic discourse and publications. Moreover, if diagrams can vary in unpredictable ways, there might seem to be no hope of providing a uniform interface for the user. A consequence of these factors is that the implementation and maintenance costs of such a program appear unacceptably high, perhaps unquantifiably so.

There seem to be two responses to this situation. One response, as seen in the tree editors described by (Paroubek *et al*, 1992) and by (Calder, 1993) and in the feature structure editor designed by (Kieffer and Fettig, 1995), is to fix a relatively small amount of graphical devices and restrict the operations defined over, and potential combinations of, those devices (perhaps to the extent that only operations which don't violate consistency with respect to a particular grammar are allowed).

An alternative response is to aim for the generality of the kind seen in the general field of diagram editing and visual programming, of which (Viehstaedt and Minas, 1995), other papers from that source, and (Myers *et al*, 1990) are good examples. And, of course, constructing diagrams by hand in a generic drawing package represents a common, but *in extremis* measure. There are several reasons why, for our purposes, generality is a disadvantage.

First, generality in this context typically goes along with complexity in the mathematical objects

to be depicted, often requiring the use of sophisticated layout algorithms, cf. (Battista *et al*, 1994). Second, there is a corresponding complexity in the specification of diagrams. That complexity may require arbitrary computation to be performed and therefore demand the power of an unrestricted programming language to describe that computation. Finally, it seems to be an assumption of such approaches that the well-formedness of a diagram should equate with the consistency of the interpretation of that diagram in the domain represented by the diagram. See (Serrano, 1997) for a clear statement of this position. This is much too strong a requirement in the cases of interest to us: one may wish to construct an inconsistent AVM, for example, precisely to verify that some other processor correctly detects the inconsistency. One may also wish to construct diagrams in formalisms which are undecidable, for example formulae in first or higher order logics. In that situation, it cannot make sense to ask an editor to enforce consistency. In the end, in an appropriately general system, it should be possible to decide on a case-by-case basis whether consistency with respect to the domain in question should be enforced.

We discuss in the next section how the design we present here obviates these problems, and allows the inexpensive and portable implementation of an appropriately general editor.

3 Design

We provide in this section a high-level specification of the editor. Details of implementation are given in section 5.

3.1 Assumptions

We make two basic assumptions. First, the well-formedness of diagrams is stated in terms of a context free grammar. This point will be illustrated below. Such an assumption is entirely in accord with practice in the areas of the specification of syntax and semantics of linguistic and semantic formalisms, including the graphical conventions used by such formalisms. Second, there is a small set of graphical primitives to state the layout of diagrams and a means for labelling parts of diagrams. Our context free assumption above means that, generally, we can require the layout problem to be deterministic for each proper subpart of a diagram and thus for diagrams as a whole, as well.

3.1.1 Graphical primitives

Our current specification makes use of three kinds of primitives. *Leaf* elements which specify the typeface in which to set a sequence of characters, for example `plain`, `italic`, *et cetera* (a total of seven). *Shape* primitives surround a single figure, for example with brackets of various kinds or with a box or circle and so on (a total of five). *Layout* primitives arrange one or more figures into larger diagrams, and these provide for vertical, horizontal, tree and array layouts (six primitives).[1] So, leaf primitives are fully specified by a series of characters; layout primitives take one or more operands each of which may be any of the primitives; shape primitives require a single operand.[2] These primitives have been selected on the grounds of generality, while preserving the property that layout is deterministic.

3.1.2 Specifying diagrams

In addition to specifying layout, we also need to indicate when a type of diagram has variable subparts, and what types of diagram may appear in those subparts. To take a particular example, we may wish to say that a *drs* consists of a *universe*, which is a collection of *referents*, and its *conditions*. Each of the conditions may be *atomic*, an *implication* or of still other types. As a point of terminology, where any number of diagrams may appear in a particular location, we will say that the diagrams that may occur there represent a *repeating type*.

Each of the elements in italic above indicates the type of a particular subpart of a larger diagram, and constitute a context free rule relating a diagram and its subparts. In the abstract (i.e. ignoring details of layout) and with the usual interpretation of Kleene star, we end up with the following characterization:[3]

(1) $\text{drs} \rightarrow \text{referent}^* \ \ \text{condition}^*$

In order for the content of a diagram to be interpretable, we allow the subparts of a diagram to be named, for example (and again in the abstract):

(2) $\text{drs} \rightarrow \ \text{universe:referent}^*$
$\text{conditions:condition}^*$

The names of subparts must be unique within any one type of diagram. All that remains is for such specifications to include layout information. A possible specification would then be as follows, where square brackets delimit sequences of specifications, and `hbox` and `vbox` provide horizontal and vertical

[1] In general, these primitives may take options to control details of layout, for example the selection of smaller or larger fonts, or alignment within layouts. In examples here, these options have been suppressed for clarity. Similarly, primitives for controlling the appearance of branches and horizontal and vertical padding are not described here.

Available tree layouts include the "standard" vertical orientation commonly used in linguistic presentations, and horizontally disposed dendro- (or clado-)grams.

[2] See Figure 2 for an example of a tree described fully using some of these primitives.

[3] Details of the concrete syntax our prototype adopts are given in section 5 below.

dispositions.

```
drs → box(vbox([hbox(universe:referent*),
                line,
                vbox(conditions:condition*)
                ]))
```

(3)

In some cases (see for example the treatment of trees shown in the Appendix), more than one type of diagram may appear in some position in a diagram. In this case, one may specify a 'union' of diagram types. Overall (and ignoring labels), a grammar of diagrams allow two kinds of production rules:

$$M \rightarrow C_1 \ldots C_m, m \geq 1 \quad (4)$$
$$N \rightarrow C_1 | \ldots | C_n, n > 1$$

where M and N are non-terminal symbols and the rewrite for any non-terminal is unique. C is a non-terminal or terminal symbol.[4] The first states that a diagram of type M consists exactly of subdiagrams of types $C_1 \ldots C_m$. The second, a diagram *union* states that diagram types $C_1 \ldots C_n$ are alternative ways of realizing a diagram of type N. It is clear that any context free grammar can be rewritten so as to fall within this class. This choice of organization contributes greatly to the simplicity of the editor's user interface.

The labelling of subparts of a diagram allows the content of a diagram to be represented in terms of sets of paths through the diagram. In general, a path is a sequence of elements of one of the following forms (where t is a diagram type, v the name of a subpart and n an integer):

$$tv \quad tvn \quad (5)$$

The first assigns a diagram type and picks out a subpart of the diagram. The second references the nth diagram within a repeating type. A path may be terminated by a pair t s where s is a sequence of characters. So, a path such as

```
(6)  drs conditions 1 implication left
```

refers to the LHS DRS in an implication which appears as the (say) first element in the conditions of a DRS. Similarly

```
(7)  drs universe 1 id "x"
```

identifies the content of the first referent in a DRS's universe.

Ultimately, this type of specification is interestingly reminiscent of proposals for "rule-to-rule" semantics, for example (Gazdar *et al*, 1985), where the interpretation (and in our case that can be taken to mean "graphical interpretation") of a structure is given in terms of a function of its subparts. More practically, one effect of the restriction to context free rules is that it is extremely easy to generate an SGML document type definition (DTD) (Goldfarb, 1990) for the content of a particular class of diagrams. This at once provides a validator for data that the editor may be expected to display and a means of specifying stream-based communication protocols between the editor and other applications. Needless to say, the existence of a declarative specification of diagram types goes a long way towards avoiding the problem of obsolescence. In our implementation, SGML is used as the 'persistence format' for user's data.

3.2 User interface

One of the most obvious benefits of the above assumptions is that the range of possible actions a user may perform on a diagram is extremely limited, regardless of how complex a class of diagrams is. In general, the actions of the user consist only of selecting a subpart of a diagram and choosing one of the diagram types allowed at that point or of performing some other action on the selected subpart. Notice how the grammar is used to constrain the range of possible types at any one location. The only "structure-based" editors we are aware of with comparable generality are those, such as `psgml` (Staflin, 1996), which interpret an SGML DTD to determine allowable material in a context dependent way.

The virtues of this simplicity should be obvious, but are worth stating. First, for educational purposes, users unfamiliar with some class of diagrams are explicitly guided through possible choices, in a way which provides immediate feedback on the consequence of choices. Second, this form of interaction is efficient. Effectively, the user provides all and only that information required to fully specify a diagram. Finally, there will be a corresponding simplicity in the relationship of the editor with a back-end processor controlling the operations of the editor for the purpose of animating operations over diagrams.

3.3 Limitations

There are substantial restrictions in the design we propose. There are many classes of diagrams used in linguistics which are more complex than trees, for example autosegmental diagrams, cf. (Bird and Klein, 1990), state transition diagrams, as used in finite state morphology, or the networks of Systemic Functional Grammar. In order to support the construction of diagrams in those particular areas, more complex systems are inevitably required. Our proposal is not intended to be so general, for precisely the reasons and benefits discussed above.

On the other hand, there are other limitations

[4] For completeness, a treatment of terminals is required and can be given straightforwardly in terms of arbitrary sequences over a limited alphabet.

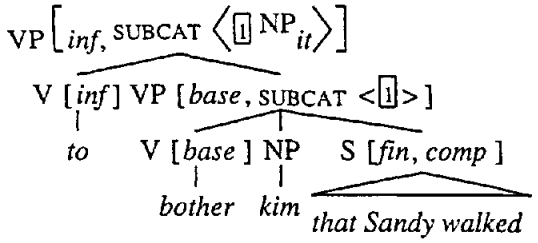

Figure 1: From (Pollard and Sag, 1994, p225).

closer to home. A natural operation over attributes in an AVM is to order them (and their values) in some way. Similarly, an AVM editor might allow type constraints as discussed in (Carpenter, 1992) to be automatically verified. One might build such information into a diagram specification (and it may be feasible in some cases to do so automatically). These limitations stem from the essential part of our design which separates clearly the graphical conventions at use in some class of diagrams from the interpretation of the content of diagrams. Under that view, if one requires some formally equivalent, but graphically different representation of some information, it makes sense for the determination of equivalence to be made by a processor dedicated to a particular formalism. In other words, issues to do with the interpretation of a diagram are not to be decided by the editor. It is our opinion that the benefits fully justify this distinction.

4 Applications

This system has been used to deliver drilling materials to undergraduates studying syntactic trees and a simplified form of DRT. Figure 6 in the appendix below shows how an editor based on the relevant class of diagrams. Experiments reveal (Cox *et al*, 1998) that viewing dynamic diagrams (perhaps with an accompanying discussion by one or more people) enhances performance significantly on tasks such as syntactic category labelling and tree construction. This enhancement is seen even when the grammar rules and categories are novel, and is (most intriguingly) still significant if no verbal explanation of the diagrams is provided.

We have also provided an interface to a locally developed tokenization engine. This tool provides a graphical interface to complex rules. Off-the-shelf technology, in the form of an SGML processor (Thompson and McKelvie, 1996), provides a simple mapping to the format required by the tokenizer. We have developed (on the basis of (Smithers, 1997)) a treatment of diagrams in (Pollard and Sag, 1994), used to construct Figure 1. Finally, we have provided Web-based visualization tools for a major corpus of dialogues (Anne Anderson *et al*, 1991).

Other classes of diagrams for which we have provided reasonably comprehensive grammars are: trees with unlimited branching and multipart node labels; categorial derivations in alternative styles; metrical trees; cladistic or cluster diagrams.

There are many other kinds of applications which can be envisaged for such a system. Here we mention just a few. The "derivation checkers" or tree editors of (Calder, 1993) and (Paroubek *et al*, 1992) can be viewed as a mode in which each action by a user is verified for consistency with respect to a grammar. Recasting that mode within the context of delaying systems for the interpretation of constraint-based formalisms (e.g. (Dörre and Dorna, 1993)) would provide a debugger in which the grammar writer could perform an instantiation and view the results, perhaps in an animated fashion. On the other hand, the "off-line" construction of trees would provide a way of querying tree banks in a more perspicuous way than via the manual construction of a query in some query language.

5 Implementation

The system described here has been implemented in Java. Figure 6 is a screen capture of an editor instance using a diagram class specification very much like that given in the Appendix. There, a tree has been constructed and a partial conversion of another tree to a DRS has been performed. In this implementation, a box containing an ellipsis indicates a position permitting one or more occurrence of a diagram type or types, a box containing a question mark indicates a location allowing a single occurrence of the available types, and a question mark on its own indicates a location where characters may appear. In the state shown in the figure, the lowest ellipsis (i.e. the one immediately below 'Pip') is *selected*. The state of the buttons labelled by diagram type names reflect the choice open to the user at that position in structure. On instantiating a diagram at a location marked by an ellipsis, a new diagram is introduced and the location of the ellipsis moved rightward or downward according to the enclosing layout.[5] Ellipses may be hidden (or revealed) by choosing the option Show ... (or Hide ...). The operation Kill allows the deletion of any selected diagram, while Yank will be available if the most recently deleted material is of a type compatible with the currently selected position. Other operations include preparing a printable form of the image or a DTD for the class of diagrams.

We use a function-like syntax to indicate the primitives and their operands. To indicate how drawing

[5]There is also an operation Insert which inserts an ellipsis to the left or above the current selection.

```
tree(plain("NP"),
  [tree(plain("Det"), [italic("the")]),
   tree(plain("N"), [italic("cat")])])
```

Figure 2: A description of a tree in terms of graphical primitives

```
diagram_spec(drs,
 box(
  vbox([hbox(var(universe, [referent])),
        line(),
        vbox(var(conditions, [condition]))
       ])))
```

Figure 3: Concrete syntax for DRSs

primitives may be combined, Figure 2 illustrates the use of a description of a diagram and could be processed by the editor to draw a subtree of the tree on the left of Figure 6.

A diagram type is specified by means of a statement such as shown in Figure 3. (Further examples are given in the Appendix.) A variable subpart of a diagram is indicated by the syntax `var(`*`name, type`*`)`. That is, a diagram of the stated type may appear in this position and be referred to by the stated name. The use of square brackets, as in both uses of `var` above, is equivalent to the Kleene star in the abstract formulation of section 3.1.2, i.e. any number of diagrams of that type may occur at this position. As a further illustration, consider the definitions shown in Figure 4. As their names suggest, the first of these limits the daughters of a tree to two, while the second allows any number of daughters. The last line illustrates the concrete syntax for diagram unions.

```
diagram_spec(two_branch,
  tree([var(mother, category),
        var(left, leaf_or_tree),
        var(right, leaf_or_tree)])
diagram_spec(arbitrary_tree,
  tree([var(mother, category),
        var(daughter, [leaf_or_tree])]))

diagram_union(tree_top, [one_branch,
                         two_branch])
```

Figure 4: Some example tree specifications

6 Conclusions, and current and future work

We have presented a design for a linguistic diagram editor which, although limited in the range of graphics it permits, nevertheless provides a configurable system of substantial benefit to a wide class of users. An implementation is available, and already in use for a wide range of applications.

We have recently extended the system to allow sequences of diagrams to be constructed and viewed. In our current work, development of a back end processor for DRSs is in hand. More generally, a range of potential architectures for interaction are under consideration. We expect that a variety of kinds of interaction will be necessary. Evaluation of the educational usefulness of the system continues. In the future, we expect to provide diagram specifications for still other formalisms, and an interface allowing the dynamic control of the editor by other programs. We anticipate that the restriction to context free organization of diagrams will be acceptable for many purposes. On the other hand, extensions to the system to allow at least some of the diagram types discussed in Section 3.3 would make the system more useful still and, in future work, we are keen to examine strategies which involve the semiautomatic layout of complex diagrams.

Acknowledgements

The work reported here was supported in part by grant TTT *Text Tokenization Tool* from the Engineering and Physical Science Research Council and by grant *The Vicarious Learner* from the Economic and Social Research Council. The author would like to thank Richard Tobin for critical comments on earlier proposals.

A Specification for a DRS and tree editor

The specifications shown in Figure 4 and 5 provide an almost complete specification for an editor like that shown in Figure 6, permitting the editing of trees with limited branching and DRSs. It has been simplified by the omission of some options controlling details of alignment and of the definitions of the diagram types `three_branch`, `implication`, `referent` and `equation`. The definition of the diagram union `tree_top` needs to be extended from that given in Figure 4. Also not included is the statement of diagram types allowed at the outermost level and their layout. The options to `hbox` and `bracket` control the separator used within the horizontal box and the shape of bracket respectively.

B Screen layout of the editor

Figure 6 shows the on-screen layout of the instance of the editor discussed in Section 4 above.

```
diagram_union(leaf_or_tree, [one_branch, two_branch, three_branch, lexical, referent])

diagram_spec(one_branch, tree([var(mother, category), var(daughter, leaf_or_tree)]))

diagram_spec(category, plain(var(Name, Text)))
diagram_spec(lexical, italic(var(Lex, Text)))

diagram_union(condition, [atomic, equation, implication, tree_top])

diagram_spec(atomic, hbox([separator(plain("")],
                          [plain(var(relation, Text)),
                           bracket([delimiter(round)],
                           hbox([separator(plain(", "))],
                                [var(referents, [referent])]))]])))
```

Figure 5: Part of a diagram specification for the diagram editor shown in Figure 6.

References

Anne Anderson *et al.* 1991. The HCRC Map Task Corpus. *Language and Speech*, 34.4, pp351–366.

Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. 1994. Algorithms for drawing graphs: an annotated bibliography *Computational Geometry Theory and Applications* 4, pp235–282.

Steven Bird and Ewan Klein. 1990. Phonological events. In *Journal of Linguistics*, 26, pp33–56.

Jo Calder 1993 Graphical Interaction with Constraint-based Grammars. In *Proceedings of the Third Pacific Rim Conference on Computational Linguistics*, Vancouver, 22–24th April, 1993, pp160–169.

Bob Carpenter 1992. *The Logic of Typed Feature Structures*, Cambridge Tracts in Theoretical Computer Science, Cambridge: University Press.

Jochen Dörre and Michael Dorna. 1993. CUF: A Formalism for Linguistic Knowledge Representation in Jochen Dörre (ed.) *Computational Aspects of Constraint-Based Linguistics Description*, ILLC/Department of Philosophy, University of Amsterdam, DYANA-2 Deliverable R1.2.A.

Richard Cox, Jean McKendree, Richard Tobin and John Lee. (to appear) Vicarious learning from dialogue and discourse: A controlled comparison. *Instructional Science.*

Gerald Gazdar, Ewan Klein, Geoffrey Pullum and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*, Basil Blackwell: Oxford.

Charles F. Goldfarb. 1990. *The SGML Handbook.* Clarendon Press: Oxford.

Hans Kamp and Uwe Reyle. 1993. *From Discourse to Logic*, Kluwer Academic: Dordrecht and London.

Bernd Kiefer and Thomas Fettig. 1995, Fegramed: An Interactive Graphics Editor for Feature Structures, Research Report RR-95-06, Universität des Saarlandes, Saarbrücken.

Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. 1990. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. In *IEEE Computer* 23.11, pp71–85.

Patrick Paroubek, Yves Schabes and Aravind K. Joshi 1992 XTAG—A Graphical Workbench for Developing Tree Adjoining Grammars. In *Proceedings of the Third Conference on Applied Natural Language Processing*, Trento, Italy, 31 March–3 April, 1992, pp216–223.

Carl Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar.* CSLI: Stanford and University of Chicago Press: Chicago and London.

J. Artur Serrano. 1997. The Use of Semantic Constraints on Diagram Editors. In *Proceedings of VL'95, 11th International IEEE Symposium on Visual Languages*, Darmstadt, Germany, 5–6 September 1995.

Gulliver Smithers. 1997. A Diagram Editor Specification for Head-driven Phrase Structure Grammar. Unpublished dissertation, Department of Linguistics, University of Edinburgh.

Lennart Staflin. 1996. PSGML, a major mode for SGML documents. See `http://www.lysator.liu.se/projects/about_psgml.html`.

Henry S. Thompson and David McKelvie. 1996. A software architecture for SGML annotation in *SGML Europe*, Graphical Communications Association: Alexandria, VA.

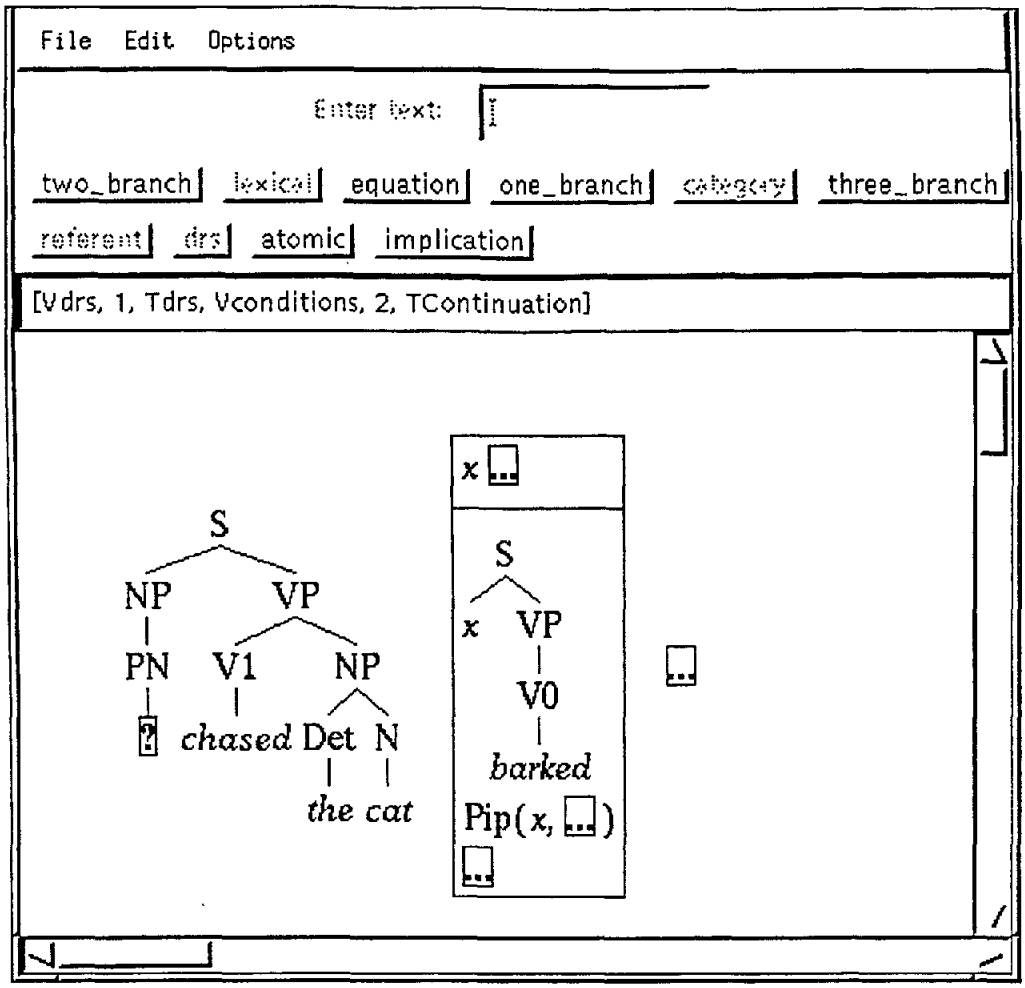

Figure 6: Screen capture of a tree and a DRS constructed using the editor. Although not reflected in this picture, the selected point of structure is the ellipsis immediately below the word '*Pip*'. The shaded words represents types which are not available at that location. They are: 'lexical', 'category', 'referent' and 'drs'. The line immediately above the diagrams indicates the path to the currently selected location.

Gerhard Viehstaedt and Mark Minas. 1995. Generating editors for direct manipulation of diagrams. In Brad Blumenthal, Juri Gornostaev and Claus Unger, editors, *Proc. 5th International Conference on Human-Computer Interaction* (EWHCI'95), Moscow, Russia, LNCS 1015, pp17–25. Springer-Verlag.