Syntax-Directed Editing of General Data Structures

Christopher W. Fraser

Department of Computer Science, University of Arizona,
Tucson, Arizona 85721

Abstract

Program editors help users create syntactically correct programs. Though such editors normally edit parse trees, applying similar techniques to other tree structures that need editing helps both users and implementors. This paper describes an editor that accepts a grammar describing a hierarchical data structure and allows the user to enter and edit arbitrary trees having this structure. It displays the pros and cons of this approach using instances of this editor that edit formatted documents, simple line drawings, and stick figures for trees.

1. Introduction

Program editors [Sandewall, Teitelbaum, van Dam] help users create programs. They prevent the entry of syntactically incorrect programs, they offer abbreviations for verbose constructs, and they display programs in a pleasant, consistent fashion. Though program editors are typically syntax-directed, and though structures other than parse trees require editing [Fraser, Fraser and Lopez, van Dam], little has been said about exploiting the generality that syntax-direction allows. For example, a syntax-directed editor might be given a description of the structure that document formatters impose on text. Users would be able to move sections and paragraphs as logical units, and, just as program editors compile code as it is entered, a document editor might format text as it is entered, displaying the formatted result instead of interleaved text and formatter commands. Implementors should also benefit. Just as compilers driven by formal language descriptions are usually easier to to understand, code, and modify than their ad hoc counterparts, an editor driven by a formal structure description should make it easier to create new editors (e.g., for new structures) and to modify old editors (e.g., to accommodate different tastes in formatting).

This paper describes a syntax-directed editor, sds, and its application to the problem of editing general data structures. It displays the pros and cons of sds' approach through examples of its instances. Though sds has been used to build a typical program editor (for a subset of the C programming language [Kernighan]), this paper will focus on less conventional applications: a binary-tree editor, an interactive document formatter, and a graphics editor. Section 2 presents these instances. Section 3 discusses sds' implementation and future.

2. Example Editors

2.1 A Binary-Tree Editor

sds is best understood through examples, and the simplest instance of sds edits uninterpreted binary trees that it displays on a graphics terminal by connecting nodes with arrows. sds extracts all of its structure-dependent parameters from a grammar that resembles grammars accepted by typical compiler-compilers [Johnson]. The grammar describing binary trees has only one production:

```
tree = value tree tree : dotree(value,tree,tree2)
```

The syntax description appears before the colon. It says that a tree is a value and two subtrees. The grammar need not say that trees may be empty because sds allows any field to be left empty until is it convenient to fill it.

The semantic action appears after the colon. Like sds, it is written in SNOBOL4. In general, id in the semantic action refers to the first occurrence of nonterminal id in the syntax description, and id*n* refers to the *n*th occurrence of nonterminal id for *n*>1. Thus tree2 in the semantic action above refers to the second occurrence of tree in the the syntax description. The semantic action above displays binary trees by passing the value and subtrees to subroutine dotree, which formats them for display. The code for the binary-tree editor's semantic routines is shown below. It is included only to suggest how sds is instantiated — SNOBOL4 details not are important here. An explanation follows the code.

```
        px = 39;  py = 0;  dx = 20
        DEFINE('dosub(x,bpx,bpy,px,py,dx)')
        DEFINE('dotree(v,l,r)') :(ends)

dosub   dosub = DIFFER(x) line(bpx,bpy+1,px,py-1) put(x)
        :(RETURN)

dotree  dotree = curpos(px - SIZE(v) / 2, py) v
        dotree = dotree dosub(l,px,py,px-dx,py+4,dx/2)
        dotree = dotree dosub(r,px,py,px+dx,py+4,dx/2)
        :(RETURN)

ends
```

The first line initializes variables that hold the screen coordinates at which the root is to be displayed (px,py) and the horizontal displacement between the root and its descendants (dx). The remaining lines define dotree and its subroutine dosub that draws subtrees. If a subtree is empty, dosub returns nothing. Otherwise, it returns code that draws a line from a node to one of its subtrees (line(...)) and then draws the subtree itself (put(x)). put is sds' display routine. It invokes the semantic actions, and most semantic actions call it recursively to display subtrees. dotree produces code that centers a value at position (px,py), and then calls dosub to produce code for the left and right subtrees. These calls temporarily adjust px, py, and dx to move subtrees down, right or left, and closer together.

This code is appended to the grammar, and a syntax preprocessor compiles the result into a record declaration for datatype tree (with fields value, tree, and tree2) and code to check the syntax of input and to format a binary tree for output. The resulting code is loaded with sds to form a complete editor.

sds is a screen editor. It always displays the current version of some record (called the 'current' record) in the structure being edited. Most semantic actions display records by recursively displaying their subfields, so sds usually displays an entire subtree of the complete 'parse' tree, though less ambitious semantic actions may be given. The user moves about by striking the terminal's cursor control keys: down moves to the current record's first field, up moves back, right and left move to an adjacent field, and home moves to the root. For example, if sds is displaying

```
   1
  / \
 2   3
/ \ / \
4 5 6 7
```

down causes it to display

```
1
```

because the first component of a record of type tree is its value field. A subsequent right causes it to display

```
 2
/ \
4 5
```

because the second component of a record of type tree is its first subtree. A subsequent up would move back to the first display. Fields are traversed in this order because the grammar gives them in this order. Were the syntax of trees changed to

```
tree = tree value tree : dotree(value,tree,tree2)
```

the initial down would move to the left subtree, and the subsequent right would move the value field.

All other sds commands are entered by typing a line of text.† To enter a terminal string or leaf, the user types the string that is to replace the current record. Subtree deletion is a special case of this command — the user merely replaces the current record with the null string. To enter a record corresponding to a nonterminal, the user types the name of the nonterminal preceded by a period. After such a command, sds drops down to focus on its first empty field. For example, if the user types the command .tree, sds creates a new node of type tree and drops down to its value field so that the user may start filling in the new subtree.

sds offers a few structure-independent commmands. .hide suppresses the current record (and thus its descendants) in subsequent displays, saving screen space, and .show causes the current subtree, if hidden, to appear once again in subsequent displays. .w*file* writes the current record and its descendants to *file*, and .r *file* reads a subtree from such a file and replaces the current record with it. .pick saves a pointer to the current record, and .put replaces the current record with the last-picked record. .pick and .put can be used to insert and delete parts of trees. For example, a new node may be inserted above the root by picking the root, replacing it with a new node, and putting the old root down as one of the new root's subtrees. Alternately, the root may be deleted and replaced with one of its subtrees by picking the subtree and putting it down on the root.

2.2 A Document Editor

Another instance of sds edits simple documents, displaying not interleaved formatter commands and text, but the final formatted result [Coulouris, Shaw, Shaw et al.]. It uses the same code as the binary-tree editor, but its grammar and semantic routines differ:

```
paper = title sect       : center(title) nl nl put(sect)
sect  = header pp sect: header nl nl put(pp) put(sect)
pp    = text pp          : break(text) nl put(pp)
```

That is, a paragraph is some text and a pointer to the next paragraph, a section is a header and pointers to its first paragraph and the next section, and a paper is a title and a

†sds' command language has been influenced by the Cornell Program Synthesizer [Teitelbaum].

pointer to the first section. Further, a paper is presented by centering the title (center(title)) and appending two 'newline' characters (nl nl) and the formatted sections (put(sect)); a section is displayed similarly, though its header is not centered; a paragraph is displayed by inserting newlines so as to fill each line (break(text)). The code for center and break is appended to the grammar, and the result is compiled into a list of record declarations (for datatypes paper, sect, and pp) and code to check syntax on entry and to format a document's parse tree for output.

Though a written description is a poor substitute for an interactive demonstration, the following trace suggests how sds is used. Each bit of indented text below describes the effect of one command. The prompt orients the user with the list of field names used to reach the current record.†

prompt:
command: .paper
tells sds to create a record of type paper, drop down to its (null) title field, display it, and wait for a command. Before creating any record, sds checks to see that it is syntactically correct at this point in the tree. Had the user requested a paragraph, sds would have created nothing.

prompt: title
command: Syntax-Directed Editing
tells sds to enter the given string as the title, advance to the paper's sect field, display it, and wait for a command.

prompt: sect
command: .sect
tells sds to create a sect record, attach it to the paper, and drop down to its first field.

prompt: sect header
command: Introduction
tells sds to enter 'Introduction' as the section header and to advance to the section's pp field.

prompt: sect pp
command: .pp
tells sds to create a pp record and to drop down to its text field.

prompt: sect pp text
command: Program editors help users...
tells sds to enter a string in this field and to advance to the field that will hold the next paragraph. This string may be arbitrarily long, and sds provides facilities like those of a conventional display editor [Irons] that may be used to edit a string before entering it into the structure.

prompt: sect pp pp
command: .pp
tells sds to create a second paragraph and to drop down to its text field.

prompt: sect pp pp text
command: This paper describes...
tells sds to enter a string there. At this point, the command

prompt: sect pp pp pp
command: up
will format and display that paragraph,

prompt: sect pp pp
command: up
will format and display both paragraphs, and

prompt: sect pp
command: down
will return the first paragraph's text field, which sds will display and open for editing with the same conventional display editor that is available when typing commands.

Note that, besides checking the document for syntactic correctness as it is entered, sds guides the dialog with prompts that suggest what is to be entered next. While the dialog has been rather long, the user has typed only the commands, which are no wordier than those typed to a conventional document formatter [Ossanna, Reid]. In fact, sds' representation of trees in permanent files closely resembles the input to conventional document formatters. The .r and .w commands read and write trees in a prefix form. .w writes the type of the node and then it recursively writes each of the node's fields. For example, the document created above would be written as

```
.paper
Syntax-Directed Editing
.sect
Introduction
.pp
Program editors help users...
.pp
This paper describes...
```

This approach to document formatting has both advantages and disadvantages. On one hand, sds would be hard put to handle complex typography. For example, global problems like widow-suppression poorly fit the context-free model, and one would want a different command syntax — say, codes embedded in text, either typed or entered via a menu [Ellis] — for specifying frequent font changes. However, sds is adequate for certain forms-driven data entry and editing [Ellis] and for simple docu-

†If a field may be occupied by items of several different types, the type of the current resident is appended to the field name so that subsequent field names will make sense. Alternately, sds could be extended to offer a command to help orient the user. It might ignore the semantic routines and display the entire tree with boxes and arrows, suppressing leaves and highlighting the current record.

ments. For example, it would be easy to extend the grammar above to create a business-letter editor that would prompt for the various fields (e.g., address, salutation) and assemble a properly-formatted letter from the responses. Also, sds makes it fairly easy to adapt semantic routines so that a document can be formatted in different ways to suit different tastes [Reid]. Finally, while the hierarchical view of documents is less conventional and thus less-understood than the linear view, it is useful often enough (e.g., by allowing one to insert, delete, and move whole sections as a unit) that it deserves closer examination.

2.3 A Graphics Editor

A less conventional instance of sds edits simple line drawings. Again, it uses the same driver as the editors above, but its grammar differs:

```
pic    = branch | color | move | scale | line†
branch = pic pic          : put(pic) put(pic2)
color  = newcolor pic     : docolor(pic,curcolor,newcolor)
move   = x y pic          : dotr(pic,ta,tb,tc+x,td,te,tf+y)
scale  = x y pic          : dotr(pic,x*ta,x*tb,x*tc,y*td,y*te,y*tf)
line   = points           : doline(points)
```

That is, a picture is a line or a command to color, scale, or move a subpicture. branch does nothing — it merely allows two subpictures to inherit one set of attributes. The semantic routines are docolor, dotr, and doline. docolor calls sds' display routine to format a picture and surrounds the result with control codes that switch first into color newcolor and then back to old color curcolor. dotr adjusts some global variables — ta-tf, which define a transformation that doline applies to all points before connecting them with lines — displays a subpicture, and restores the original transformation. Code to initialize the global variables and to define docolor, dotr, and doline is appended to this grammar, and sds' preprocessor compiles the result into a list of record declarations, a syntax checker, and a display routine. The resulting editor makes it fairly easy to create and edit simple pictures. For example, the command sequence

```
.branch
.color
red
.line
0,0 100,100
.color
blue
.line
0,100 100,0
```

draws a large 'X' with a red rising stroke and a blue falling stroke, and

```
home
down
down
green
```

makes the red stroke green. Again, a written trace is a poor substitute for a demonstration, because sds would have been changing the display with each command to show the path to, and contents of, the current record as it changes.

Because .pick copies pointers, not subtrees, it can violate tree structure. Though this feature is dangerous, users editing structures like the graphics structure may find it handy. By picking a structure and putting it down in several places, all instances of that structure may be changed by changing the single copy. (Because this feature is not universally desirable, it would be better if sds offered both copying and non-copying .pick commands). This observation raises a larger issue: sds can be made to edit arbitrary graph structures. Semantic actions for cyclic structures would have to take care to avoid loops, and the .r and .w commands would have to use a different encoding, but sds does not otherwise assume that it is editing a tree.

The graphics editor is incomplete. For example, it can neither rotate pictures, clip them to fit the screen, nor present objects other than lines (e.g., filled polygons, curves). All of these features are easily added by extending the grammar and code generation routines, but some features resist this attack. For example, sds' command language offers no way to enter coordinates by pointing instead of by typing numbers, so this change would have to be made to sds, not the grammar. A general solution to this problem may be to have one grammar that defines the structure and another that defines the command language.

3. Discussion

sds is written in SNOBOL4, though it could have been implemented in a more conventional, compiled language — interpretation and garbage collection are handy but not required. Its data-independent code is roughly 200 lines long. The syntax descriptions (with semantic routines) for the structures described above are 15-40 lines each, and they are compiled into code roughly 2-3 times that length. An editor like the binary-tree editor can be brought up in about an hour by someone familiar with sds. The semantic actions require most of the effort. Once they are finalized, the editor may be changed quickly. For example, when writing this paper it became obvious that it would be easier to describe Section 2.2's document structure than the original one:

```
paper = title sects    : center(title) nl nl put(sects)
sects = sect sects     : put(sect) put(sects)
sect  = header pps     : header nl nl put(pps)
pps   = pp pps         : put(pp) put(pps)
pp    = text           : break(text) nl
```

While this change resulted in changes to many lines of (generated) code, the ability to change only the syntax

†Nonterminals whose definitions use only alternation are omitted from sds' parse trees to avoid clutter. Accordingly, productions involving only alternation have no semantic action.

description allowed it to be completed in five minutes.

sds is loaded with several short SNOBOL4 routines that handle the terminal interface. Thus text-based editors like the document and C editors run on several models of terminals, though the tree and graphics editors run on only one model because their semantic actions assume that model's control sequences.

sds is experimental. It still needs thorough testing, optimization, and documentation, and many aspects need polishing. For example, sds' cavalier screen refreshing would be tedious were communications slow; it would be better to display more context than just the current record and then indicate the current record by highlighting it or by pointing the cursor at it. While the user interface needs work, more complete versions of the document and graphics editors, and attacks on new data structures, are likely to produce more interesting results.

Acknowledgment

This work has benefited greatly from discussions with Dave Hanson.

References

G. Coulouris et al. The design and implementation of an interactive document editor. *Software — Practice and Experience 6*(2):271-279, April 1976.

C. Ellis and G. Nutt. Office information systems and computer science. *ACM Computing Surveys 12*(1):27-60, March 1980.

C. Fraser. A generalized text editor. *Communications of the ACM 23*(3):154-158, March 1980.

C. Fraser and A. Lopez. Editing data structures. To appear in *ACM Transactions on Programming Languages and Systems.*

E. Irons and F. Djorup. A CRT editing system. *Communications of the ACM 15*(1):16-20, January 1972.

S. Johnson. YACC — yet another compiler-compiler. Technical report, Bell Labs, Murray Hill, NJ, 1975.

B. Kernighan and D. Ritchie. *The C programming Language.* Prentice-Hall, 1978.

J. Ossanna. Troff user's manual. Technical report, Bell Labs, Murray Hill, NJ, 1977.

B. Reid. A high-level approach to computer document formatting. *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*:24-31, January, 1980.

E. Sandewall. Programming in the interactive environment: The LISP experience. *ACM Computing Surveys 10*(1):35-71, March 1978.

A. Shaw. A model for document preparation systems. Technical report 80-04-02, Department of Computer Science, University of Washington, April 1980.

A. Shaw, R. Furuta, and J. Scofield. Document formatting systems: Survey, concepts, and issues. Technical report 80-10-02, Department of Computer Science, University of Washington, October 1980.

T. Teitelbaum. The Cornell program synthesizer: A tutorial introduction. Technical report 79-381, Department of Computer Science, Cornell, 1980.

A. van Dam and D. Rice. On-line text editing: A survey. *ACM Computing Surveys 3*(3):93-114, September 1971.