Displaying IDL Instances

Richard Snodgrass

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

Abstract

Debugging a complex system can be aided by displaying the internal data structures manipulated by the system. We examine one generic Unix tool and four IDL-specific tools, focussing on their functionality and ability to handle large data structures gracefully.

One often cited advantage of using IDL in the implementation of complex systems is that it can aid in debugging [Lamb 1983]. The software system is partitioned into a collection of IDL processes, each of which may be debugged individually. Each process is tested be examining the output(s) produced from given input(s). It is no longer necessary to debug the system *linearly*, where a process is debugged only after the processes producing the data structures needed by the one process are debugged. Instead, debugging can proceed *concurrently*, since input test cases for each IDL process can be generated by hand.

This paper examines one aspect of the debugging task, that of viewing instances of IDL structures. The problems encountered involve functionality and scale. The obvious solutions are difficult to use of instances of medium size, and are impractical on large instances, which unfortunately are common. We will examine a range of approaches. Two examples will be used. Example A, from [Warren et al. 1986], is quite short, containing three nodes and 18 attribute-value pairs. Example B is somewhat larger, containing 103 nodes and 309 attributes.

1 The ASCII External Representation

Instances of IDL structures can be written to external storage in a variety of formats. Every implementation is required to support the *ASCII External Representation Language*, in which an arbitrarily graph-structured instance may be encoded as a linear character sequence in a restricted character set. This representation allows users to create IDL instances via a conventional text editor, and supports viewing via conventional text outputing commands. Example A in this representation is shown below.

```
-- file billing/customers.in
L1: customer_list [list <L2^ L3^ L4^>]
L2: commercial_customer [
name "Innovation, Inc.";
address "Freedom Trail";
active TRUE;
customer_number 1;
balance 9546782.00;
industry_code 12
]
L3: state_customer [
name "Department of Obfuscation";
address "Bureaucracy Boulevard";
active FALSE;
customer_number 2;
balance -1000000000.85;
state_code 50
]
L4: federal_customer [
name "Office of the Director, OMB";
```

```
address "Wonderland";
active FALSE;
customer_number 3;
balance -1000000000000.13;
agency_code 1348903
]
#
```

In this representation, nodes are denoted by their types (e.g., "customer_list"), followed by attribute-value pairs separated by a semicolon and delimited by square brackets. Attributes with nodes as values are denoted either inline or by a node reference (e.g., "L2^"). Sequences and sets are delimited by "<>" and "{}", respectively.

At least four problems arise when using this format. First, the ASCII external representation does not enforce any line break or indentation conventions, hence a tool may produce output that is difficult to read. Example B, generated directly by the IDLC tool[Shannon 1985], exhibits this problem:

```
compilationUnit[lex_info 0;
syn_body <StructureEntity[syn_name "customers";
sem_duplicate FALSE;
lex_namepos 31;
lex_endpos 1109;
lex_beginpos 21;
sem_definitions {};
sem_assertions {};
syn_root NamedTypeRef[syn_name "customer_list";
lex_namepos 46;
sem_entity L336516:Class[sem_name "customer_list";
sem_copiedfrom void;
sem_definitionPoint 133;
sem_rep ClassRep[sem_id <>;
 ...
```

It can be quite tedious matching the initial and final delimiters for nodes, sets, and sequences.

The second problem is that of scale. Example B is 311 lines long; we gave less than 5% above. Scanning that many lines looking for a particular subtree is not easy, especially when some arcs are indicated inline and others are represented as references. In fact, Example B is shorter than most IDL instances, as it is the internal representation of a 22 line specification (the customers structure given in the tutorial). Several hundred to several thousand line programs are the norm, resulting in intermediate representations containing 10^3 to 10^6 nodes. As a concrete example, the internal representation of a 2584 program contains 7141 nodes; its ASCII representation is 40,000 lines (almost a million characters) long. In such situations, some editors cannot even read the long files, and printouts of these files could be many feet thick.

A third problem is that the graphical nature of IDL instances is hidden in the ASCII representation. Even gross characteristics such as the overall shape of the instance are difficult to determine.

A final problem is that these instances may be viewed only after the tool completes. While IDL aids debugging by partitioning a tool such as a compiler into many IDL processes, it is still possible to view instances only at process boundaries.

In the next four sections, we will examone approaches that address each of these problems individually. We will conclude with a tool that attempts to solve all four problems simultaneously.

2 IDLFORMAT

The first problem identified with the ASCII external representation was determining node, set, and sequence boundaries in the absence of lexical clues. A tool in the UNC IDL toolkit, IDLFORMAT, inserts line breaks and spaces at appropriate places to achieve readability:

```
compilationUnit
 [lex_info 0;
  syn_body
   <StructureEntity
     [syn_name "customers";
      sem_duplicate FALSE;
      lex_namepos 31;
```

```
    lex_endpos 1109;
    lex_beginpos 21;
    sem_definitions
     {}
    ;
    sem_assertions
     {}
    ;
    syn_root NamedTypeRef
     [syn_name "customer_list";
    ...
```

We can now see which attributes are associated with each node using indentation as a guide. IDLFORMAT does not address the other problems, including that of scale. In fact, it exacerbates the scalability problem, by making the file longer (Example B was increased in length by 73%, from 311 lines to 539 lines; the number of characters increased by 40% due to the added spaces).

3 TREEWALK

The TREEWALK tool allows the user to interactively examine an IDL instance. It starts by displaying the root, then accepts commands to traverse the graph. Attributes are numbered, allowing easy identification. After each command, indicated by "+>", the current node is displayed. TREEWALK retains the path of nodes encountered; the "up" command returns to the previous node. Through the appropriate "up" and "down" commands, it is possible to view the entire instance, albeit one node at a time. This process is illustrated on Example A below, where all the nodes are eventually visited:

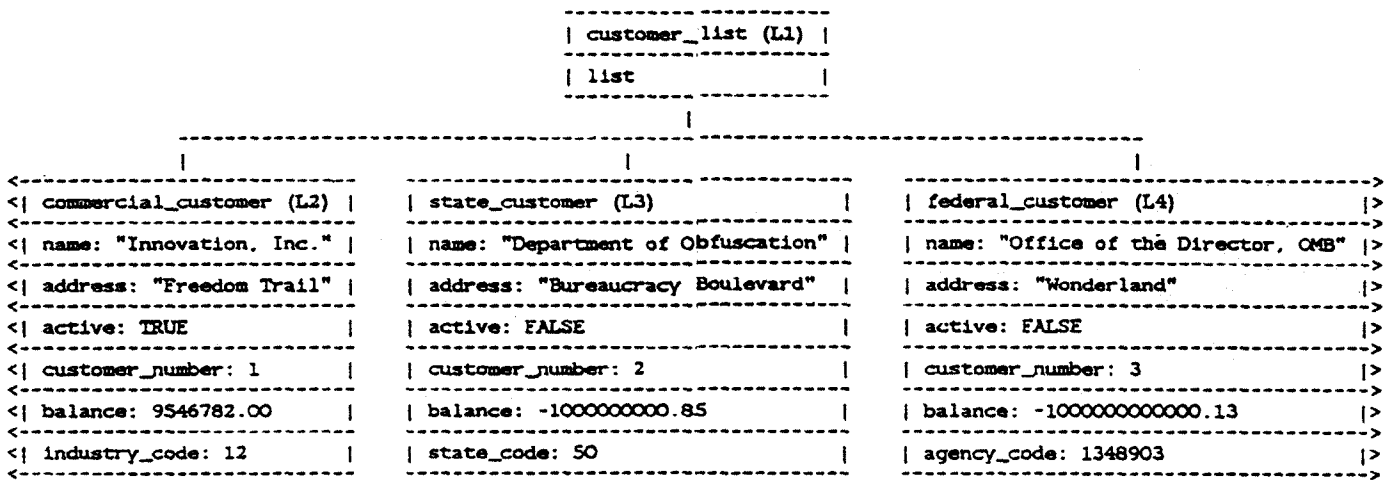
```
L1:customer_list                              Depth:  0
 1 list  Sequence Of
        1               L2: commercial_customer
        2               L3: state_customer
        3               L4: federal_customer
+> d 1 1
L2:commercial_customer                        Depth:  1
 1 name                 -> "Innovation, Inc."
 2 address              -> "Freedom Trail"
 3 active               -> TRUE
 4 customer_number      -> 1
 5 balance              -> 9546782.00
 6 industry_code        -> 12
+> u
L1:customer_list                              Depth:  0
 1 list  Sequence Of
        1               L2: commercial_customer
        2               L3: state_customer
        3               L4: federal_customer
+> d 1 2
L3:state_customer                             Depth:  1
 1 name                 -> "Department of Obfuscation"
 2 address              -> "Bureaucracy Boulevard"
 3 active               -> FALSE
 4 customer_number      -> 2
 5 balance              -> -1000000000.85
 6 state_code           -> 50
+> u
L1:customer_list                              Depth:  0
 1 list  Sequence Of
        1               L2: commercial_customer
        2               L3: state_customer
        3               L4: federal_customer
+> d 1 3
L4:federal_customer                           Depth:  1
 1 name                 -> "Office of the Director, OMB"
```

```
 2 address              -> "Wonderland"
 3 active               -> FALSE
 4 customer_number      -> 3
 5 balance              -> -1000000000000.13
 6 agency_code          -> 1348903
+> q
Goodbye
```

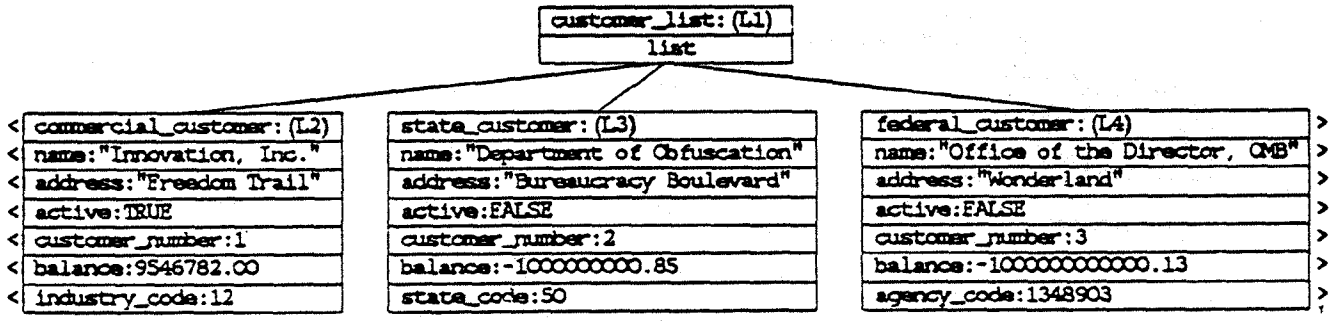
One advantage of TREEWALK is its interactive nature—the user can focus on a subtree rather quickly (finding any particular node in Example B takes less than a dozen commands). Scale is not a major problem if the user is concerned with a small part of the graph and knows approximately where it is. Finally, note that in TREEWALK there is no distinction between inline and referenced nodes.

4 TREEPR

Neither IDLFORMAT nor TREEWALK display the IDL instance as a graph, with nodes and edges. Viewing the instance as a graph can be quite informative. Gross aspects of the instance, such as shape, depth, degree of sharing, and proximity to a tree, can often be determined visually. Finding particular nodes is also much easier. The TREEPR tool allows instances to be displayed graphically on a character or laser printer. The following is the output generated for Example A for a character printer:



and for a laser printer:

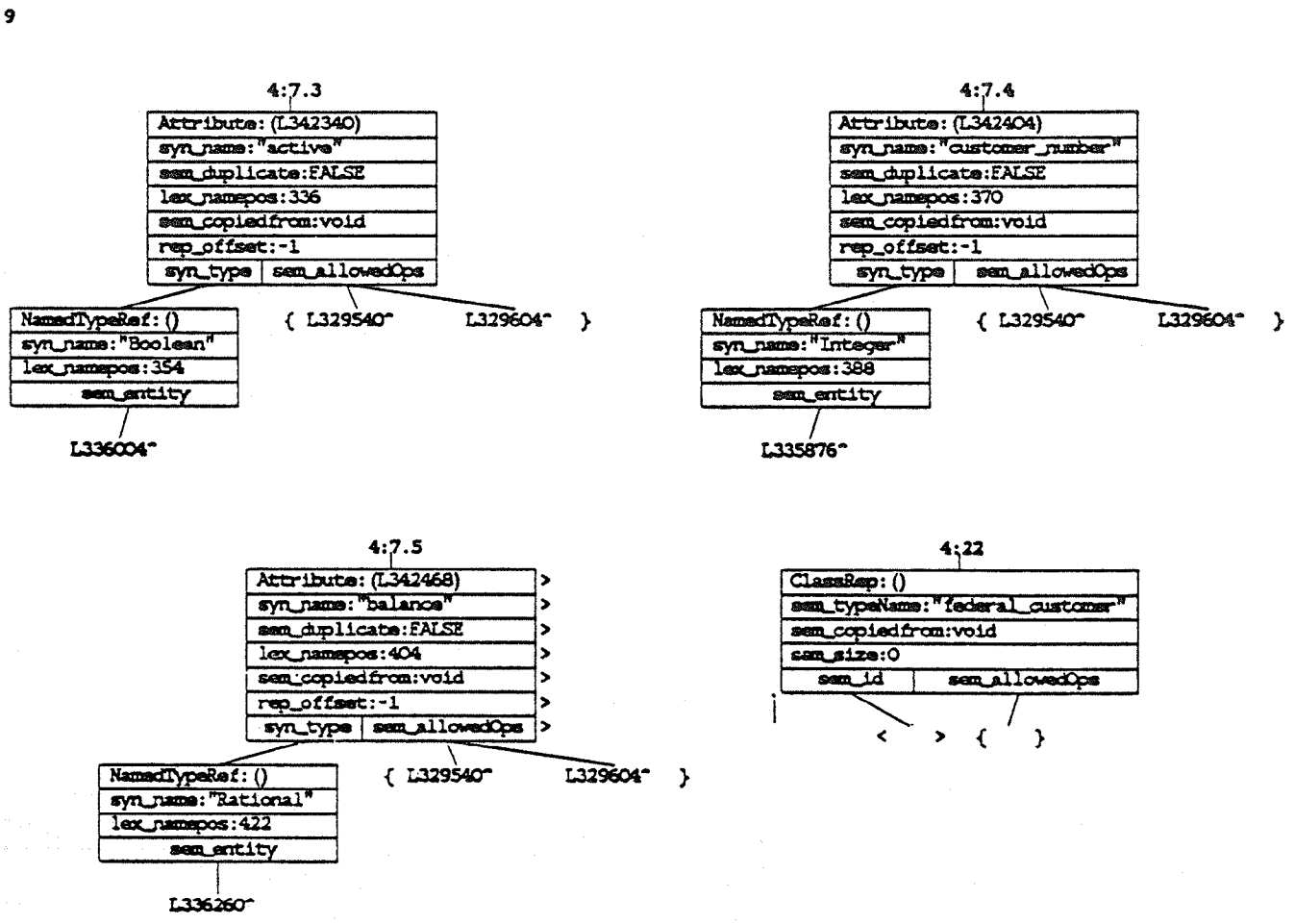


It is important to note that TREEPR technically displayes *trees*. It reads in an IDL instance, embeds a tree in the graph, outputs the arcs of the embedded tree as arcs, and outputs the remaining arcs as node references (e.g., "L2^"). Sets and sequences are displayed left to right, with distinguished delimiters ("{" and "}" for sets, "<" and ">" for sequences). TREEPR is able to divide the output into pages, to be later taped together.

TREEPR attempts to minimize the size of the output while producing attractive trees. It first assigns a height and a width to each node. The virtual x-coordinate of each node is assinged in a single left-to-right postorder traversal of the tree. Nodes are assigned a tentative position, then shifted to the right as needed. Parent nodes are centered over their children. Sets and sequences are handled as if they were sons of the parent node. The y-coordinate is computed during printing. This algorithm is a variant of that presented by Vaucher [Vaucher 1980]. A standard clipping procedure is used to compute the portion to be printed on each page [Newman & Sproull 1979].

Such a display can be quire large even for moderate instances: example B is 7 square feet in size. One interesting insight gained from using TREEPR is that most parse tress are not really "trees" at all—they should be called "parse bushes" or "parse weeds"! The reason is that parse trees are usually very wide and thin. Example B, an augmented syntax tree of a 22 line specification, is 12 feet long and 7 inches deep when output by TREEPR. There are two reasons for this phenomenon: TREEPR displays sets and sequences left-to-right, and most programs are composed on many units (procedures, statements) each with limited internal nesting.

There are two ways available for dealing with such an awkward printout. The first is to *compress* it, using the "-c" option to TREEPR. Of course, in compressed mode, much less information is displayed. One can also use the "-p" option to get a *paginated* output (a portion from Example B is shown below) that can be placed in a notebook rather than on a wall.



5 DBX

The three tools just described, IDLFORMAT, TREEWALK, and TREEPR, all display IDL instances that have been manually written or output by an IDL process. While such representations are extremely helpful in debugging, they are often less useful than a view of the instance in memory as it is being manipulated by the process. The symbolic debugger DBX distributed with Unix allows the memory-resident instances to be viewed, although in a quite awkward fashion. We will illustrate with a session with DBX. We first enter DBX and view the main algorithm of the IDL process, which simply

reads in an instance. DBX prompts with "(dbx) "; the user's input follows.

```
dbx version 3 of 3/2/85 10:17 (thorin).
Type 'help' for help.
reading symbolic information ...
(dbx) list 4,10
    4   main()
    5   {
    6     customer_list Acustomer_list;
    7
    8     Acustomer_list = InCustomerList(stdin);
    9
   10   }
```

After stopping directly after the reader, we examine the root of the instance.

```
(dbx) stop at 8
[1] stop at 8
(dbx) run < customers.in
[1] stopped in main at line 8
    8     Acustomer_list = InCustomerList(stdin);
(dbx) step
stopped in main at line 10
   10   }
(dbx) print *Acustomer_list
(IDLhidden = (TypeID = 4, Touched = 0, Shared = 0), list = 0xc1d4)
(dbx) print *Acustomer_list->list
(next = 0xc1e4, value = [union])
```

The value of the `list` attribute is a pointer to a linked list cell, whose first entry is a C `union`, since it is a sequence of a class type. We determine that its type is 2, which we decode to be the `commercial_customer` node type by scanning the include file produced by the IDL translator. Knowing its type, we can then examine the node's attributes.

```
(dbx) print Acustomer_list->list->value.IDLclassCommon.IDLhidden.TypeID
2
(dbx) print *Acustomer_list->list->value.Vcommercial_customer
(IDLhidden = (TypeID = 2, Touched = 0, Shared = 0), name = "Innovation, Inc.",
address = "Freedom Trail", customer_number = 1, balance = 9.54678e+06,
active = '^A', industry_code = 12)
```

This process continues until we have examined the entire structure.

```
(dbx) print *Acustomer_list->list->next
(next = 0xc1f4, value = [union])
(dbx) print Acustomer_list->list->next->value.IDLclassCommon.IDLhidden.TypeID
8
(dbx) print *Acustomer_list->list->next->value.Vstate_customer
(IDLhidden = (TypeID = 8, Touched = 0, Shared = 0),
name = "Department of Obfuscation", address = "Bureaucracy Boulevard",
customer_number = 2, balance = -1e+09.0, active = '\0', state_code = 50)
(dbx) print *Acustomer_list->list->next->next
(next = (nil), value = [union])
(dbx) print Acustomer_list->list->next->next->value.IDLclassCommon.IDLhidden.TypeID
6
(dbx) print *Acustomer_list->list->next->next->value.Vfederal_customer
(IDLhidden = (TypeID = 6, Touched = 0, Shared = 0),
name = "Office of the Director, OMB", address = "Wonderland",
customer_number = 3, balance = -1e+12.0, active = '\0', agency_code = 1348903)
(dbx) quit
```

This example illustrates several difficulties in using DBX. First, all print commands must start at a named variable in the program, making paths of more than a few nodes impractical. Secondly, the user is forced to contend with the specific C implementation, including various fields that should not be seen by the user (e.g., `Touched` and `Shared` in `IDLhidden`). The C interface supports multiple representations for some IDL constructs, and hides the representation through the use of macros and functions. For example, sequences may be represented as linked lists or as arrays [Shannon & Snodgrass

1986A]. In either case, the programmer employs identical macros, so he need not be aware of which internal representation is used. The tools discussed previously will display the sequence identically independent of the representation. However, DBX displays the linked list cells in a format different from that of arrays, emphasizing the representation. Finally, we note that a prototype extension to DBX has been constructed that displays data structures graphically []; a similar facility is also available in the Cedar environment [Myers 1980]. While these tools effectively utilize a workstation's high resolution display, they still exhibit the scalability and representation hiding difficulties discussed above.

6 IDLVIEW

The IDLVIEW tool was designed to solve all the problems discussed in this paper:

Readability Each node is displayed graphically in a window on a high resolution screeen (currently a Sun workstation). Arcs between nodes illustrate connectivity. The display looks similar to that produced by TREEPR, although displaying subgraphs quickly is more important than producing aesthetically pleasing output or minimizing display space.

Scalability IDLVIEW is incremental. When an IDL instance is to be displayed, a window is created on the workstation, and only enough nodes are displayed as will fit into the window. The user can move the window around the instance, and again, only the nodes needed to fill the window are processed. A small subwindow shows the position of the portion actually displayed in the main window relative to the portion of the instance processed so far.

Graphical nature Nodes and arcs are displayed. By moving the window, the entire instance can be viewed eventually.

In-memory instances IDLVIEW is linked with the IDL process, allowing it to display instances residing in main memory (instances residing on secondary storage are simply a special case). However, the display portion resides in another process (termed the *display*) process, communicating with the IDL process (termed the *primary*) process through Unix sockets. Such an arrangement allows the display process to execute on a different machine from the primary process.

IDLVIEW will also offer other capabilities:

Multiple windows Each window is associated with a particular root node. A node might be displayed in several windows simultaneously; a command identifies such nodes.

Display tailoring IDLVIEW by default displays all the attributes of a node. Certain attributes can be eliminated from the display, either for an individual node or for all nodes. Various levels of detail are supported, from no detail (similar to the compressed mode of TREEPR) through high level (*a/a* the ASCII external representation and the expanded mode of TREEPR) to low level (*a/a* DBX). Automatic sizing also allows fewer or more nodes to be displayed.

Symbolic information IDLVIEW makes use of the Candle [Shannon & Snodgrass 1986B] description of the structure of the instance being displayed. In particular, IDLVIEW can display the IDL specification of an indicated attribute, node, or class on demand.

7 Status

IDLFORMAT, TREEWALK, and TREEPR have been implemented and are distributed with the current release of the UNC IDL toolkit[Snodgrass 1985]. IDLVIEW has been designed and is currently being implemented on a Sun workstation using Sunview. DBX is distributed with Unix.

8 Acknowledgements

IDLFORMAT was implemented by Karen Shannon; TREEWALK, by Dean Throop; TREEPR by Nancy Butler, Joan Curry, Steven Konstant, and Dore Rosenblum; and IDLVIEW, by Lawrence Ross.

9 Bibliography

[Lamb 1983] Lamb, D.A. *Sharing Intermediate Representations: The Interface Description Language.* PhD. Diss. Computer Science Department, Carnegie-Mellon University, May 1983.

[Myers 1980] Myers, B.A. *Displaying Data Structures for Interactive Debugging.* Stanford, June 1980. also published as Xerox PARC Technical Report CSL-80-7.

[Newman & Sproull 1979] Newman, W.M. and R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

[Shannon 1985] Shannon, K. *idlc Users Manual (Version 2.0).* SoftLab Document No. 8. Computer Science Department, University of North Carolina at Chapel Hill. Dec. 1985.

[Shannon & Snodgrass 1986A] Shannon, K. and R. Snodgrass. *Mapping the Interface Description Language Type Model into C.* SoftLab Document No. 24. Computer Science Department, University of North Carolina at Chapel Hill. Mar. 1986.

[Shannon & Snodgrass 1986B] Shannon, K. and R. Snodgrass. *Candle: Common Attributed Notation for Interface Description.* SoftLab Document No. 26. Computer Science Department, University of North Carolina at Chapel Hill. Jan. 1986.

[Snodgrass 1985] Snodgrass, R., editor *IDL Manual Entries (Version 2.0).* SoftLab Document No. 15. Computer Science Department, University of North Carolina at Chapel Hill. Dec. 1985.

[Vaucher 1980] Vaucher, J.G. *Pretty-Printing of Trees. Software–Practice and Experience,* 10 (1980), pp. 553-561.

[Warren et al. 1986] Warren, W.B., J. Kickenson and R. Snodgrass. *A Tutorial Introduction to Using IDL. SIGPlan Notices,* (1986).