ISI Reprint Series
ISI/RS-87-196
September 1987

University of Southern California

Gabriel Robins

The ISI Grapher: A Portable Tool for Displaying Graphs Pictorially

Reprinted from the *Proceedings of Symboliikka '87,* held in Helsinki, Finland, on August 17-18, 1987.

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RS-87-196	5. MONITORING ORGANIZATION REPORT NUMBER(S) ---------------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute	6b. OFFICE SYMBOL *(If applicable)*
7a. NAME OF MONITORING ORGANIZATION ---------------	
6c. ADDRESS *(City, State, and ZIP Code)* 4676 Admiralty Way Marina del Rey, CA 90292	7b. ADDRESS *(City, State, and ZIP Code)* ---------------
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-81-C-0335	
8c. ADDRESS *(City, State, and ZIP Code)* DARPA 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
---------------	---------------	---------------	---------------

11. TITLE *(Include Security Classification)*
The ISI Grapher: A Portable Tool for Displaying Graphs Pictorially [Unclassified]

12. PERSONAL AUTHOR(S)
Robins, Gabriel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Research Report	FROM ________ TO ________	1987, September	23

16. SUPPLEMENTARY NOTATION
Reprinted from the *Proceedings of Symboliikka '87,* held in Helsinki, Finland, on August 17-18, 1987.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
artificial intelligence tools, graphs, graph algorithms, intelligent systems, layout algorithms, user- interfaces

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The advent of inexpensive personal workstations with high-resolution displays has helped to drastically increase end-user productivity. However, the same technology has also served to highlight the deficiencies inherent in current pieces of software and existing user-interfaces. A small set of concepts (e.g., windows, menus, icons, etc.) has established itself as a good model for user-interface design. We propose an important addition to this collection, namely the concept of a "grapher"; that is, the ability to interactively display and manipulate arbitrary directed graphs. We illustrate the usefulness of this idea, develop a practical linear-time algorithm for laying-out graphs, and describe our implementation of a prototype, the ISI Grapher.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Sheila Coyazo Victor Brown	22b. TELEPHONE *(Include Area Code)* 213-822-1511 — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ISI Reprint Series
ISI/RS-87-196
September 1987

University of Southern California

Gabriel Robins

The ISI Grapher: A Portable Tool for Displaying Graphs Pictorially

Reprinted from the *Proceedings of Symboliikka '87,* held in Helsinki, Finland, on August 17-18, 1987.

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903-81-C-0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

The ISI Grapher:
a Portable Tool for Displaying Graphs Pictorially

Gabriel Robins
Intelligent Systems Division

Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, Ca, 90292-6695, U.S.A.
gabriel@vaxa.isi.edu

Abstract

The advent of inexpensive personal workstations with high-resolution displays has helped to drastically increase end-user productivity. However, the same technology has also served to highlight the deficiencies inherent in current pieces of software and existing user-interfaces. A small set of concepts (e.g. windows, menus, icons, etc.) has established itself as a good model for user-interface design. We propose an important addition to this collection, namely the concept of a "grapher"; that is, the ability to interactively display and manipulate arbitrary directed graphs. We illustrate the usefulness of this idea, develop a practical linear-time algorithm for laying out graphs, and describe our implementation of a prototype, the ISI Grapher.

Keywords: user-interfaces, intelligent systems, graphs, graph algorithms, layout algorithms, artificial intelligence tools.

1. Introduction

The advent of inexpensive personal workstations with high-resolution displays, fast processors, and large memories has helped to drastically increase end-user productivity. However, the same technology has also served to highlight the deficiencies inherent in current pieces of software and existing user-interfaces. In particular, a good user interface is now considered to be singularly important in determining the usefulness and success of many kinds of systems. Considerable emphasis has been placed on the uniformity, universality, and consistency of user interface design [Kaczmarek, Mark, and Wilczynski].

A small and integrated set of concepts -- desktops, windows, menus, icons, dialog boxes, forms, mouse clicks, etc. -- has been established as a good model for user interface design. We propose an important addition to this collection: namely the

This research was supported in part by the Defense Advanced Research Projects Agency under contract number MDA903 81 C 0335. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. government, or any person or agency connected with them.

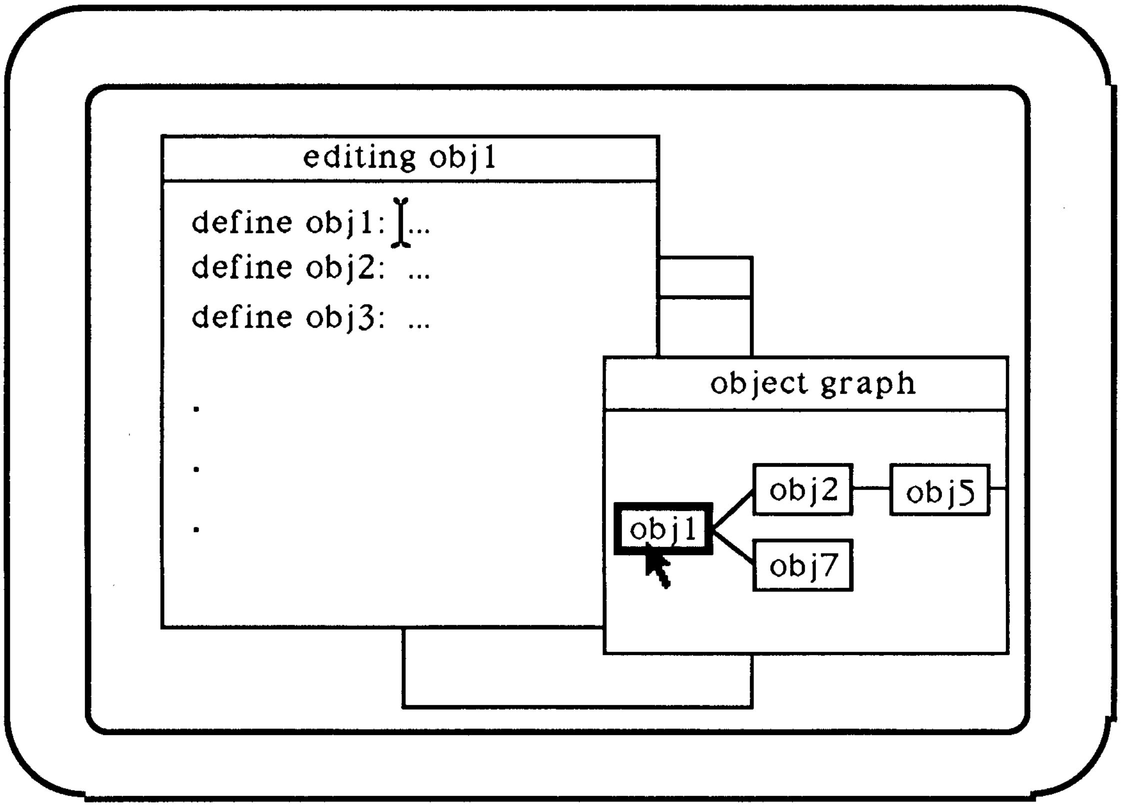
An invited talk in Symboliikka '87, August 17-18, 1987, Helsinki, Finland.

concept of a ***grapher***. that is, the ability to interactively display and manipulate arbitrary directed graphs. We illustrate the usefulness of this idea, develop a practical linear-time algorithm for laying out graphs, and describe our implementation of a prototype, the ISI Grapher.

2. The Usefulness of a Grapher

Consider the following model of an interactive environment: several editor windows are active, each containing one or more objects which may reside over several files or machines. For the sake of concreteness, let us say that the objects being edited are procedures or semantic nets. It is very likely that due to the sheer complexity and number of these objects, the user soon lose track of which objects he has modified or of the relationships between these objects.

To alleviate this problem, we may introduce an additional ***grapher window***, containing a picture of the relations between the objects that are being edited/inspected in other windows. In the following example, nodes represent functions and procedures, while edges represent static lexical scoping. Now the user has a global view of his current state of his editing session. This grapher window is highly interactive; for example, clicking with the mouse on a graph node would cause the definition of the corresponding object to appear in an editor window, whereupon the user may modify that definition. Pictorially, the scenario we envision would appear as follows:

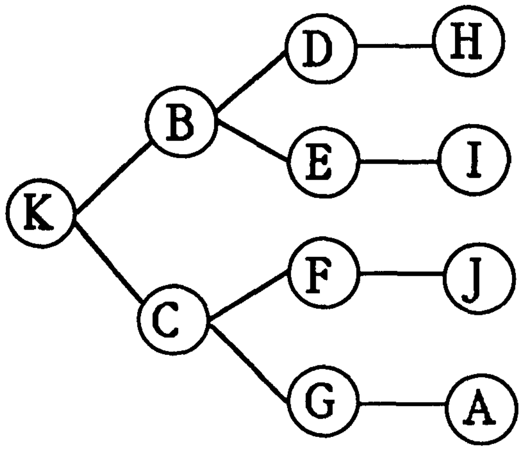


Other interpretations of the graphs are possible and are equally useful. For example, in an AI knowledge representation environment, nodes may represent concepts and edges may represent logical subsumption; in a grammar system, nodes may represent symbols (terminals or nonterminals) and edges may represent productions; in a file system, nodes may represent files and edges may represent directory containment; in a distributed environment, nodes may represent machines and edges may represent communication links, and so on.

3. Pictorial Display vs. Syntactical Display

If "a picture is worth a thousand words," then it can well be argued that "a graph is worth a hundred well-formed formulas." For example, consider the following directed graph, given by its edge set: G = {(D,H), (E,I), (C,G), (B,E), (F,J), (C,F), (K,B), (B,D), (K,C), (G,A)}. Visualize the structure of the graph G from this representation is not trivial. Suppose that we were told that G is in fact a tree; is it then obvious whether G is a binary tree? The answer is still not apparent at first glance. Even if we are told that in fact G is a binary tree, how easy would it be for us to determine what the root of G is? And even if we are further told that G is a binary tree with root K, how quickly could we determine whether G is a balanced tree?

The answers to all these queries would be obvious if instead of specifying G formally using set-theoretic notation as above, we had simply been given a pictorial diagram of G, as follows:



The pictorial representation of G.

This discussion alludes to the conclusion that because humans are good at pattern-recognition, it is often preferable to display the pattern pictorially rather than its equivalent formal syntactic representation. We may wonder why certain properties are difficult for us to infer from formal descriptions, yet are trivially apparent from appropriate diagrams; this is partially due to the fact that *transitive closures* are difficult to compute mentally. Perhaps this is related to the fact that the transitive closure predicate *can not* be specified in first order logic. A compounding problem is that humans find it difficult to keep track of a large number of identifiers, even if they are mnemonic. However, such questions are best left for cognitive scientists to muse about.

The heart of any grapher would be an algorithm for laying out directed graphs. However, finding optimal layouts for graphs is quite a difficult problem. Even in the special case of laying out binary trees optimally on the lattice plane, the problem turns out to be NP-hard, as is its approximation of within 4 percent! [Supowit and Reingold]. By *optimal* layout we mean a layout that minimizes some parameters, such as the total width of the resulting diagram, or the number of edge crossings. Other researchers have proposed various algorithms to layout and display graphs [Vaucher] [Reingold and Tilford], some of which are similar to the one adopted by the ISI Grapher.

Interestingly enough, laying out binary trees on the continuous real plane reduces to linear programming. This is a small consolation, however, as solutions to continuous problems do not directly map (via rounding) into solutions to the corresponding discrete versions, but the discrete and the continuous solutions can instead be arbitrarily far apart [Papadimitriou and Steiglitz, p. 327]. This is rather discouraging, as we would like to be able to layout large graphs (of several thousands of nodes) interactively and in "real-time." Moreover, the notion of "optimality" with respect to a layout is quite subjective; in the above discussion, a reasonable set of "esthetic" heuristics with respect to binary trees had to be fixed. We then need to ask ourselves how important optimality is to us in the resulting layout; from this point on we use the assumption that users in most interactive applications would be willing to sacrifice some "beauty" in exchange for a considerable increase in speed.

4. Definition of the Problem

In order to make the problem more tractable and concrete, we make several design decisions: we assume that nodes are to be represented by rectangles, and that edges are to be represented by straight line segments. Next, we constrain all the children of a node to appear in the layout to the right of all of their ancestors. Furthermore, we sidestep the problem of having to draw cycles, via a structure-preserving mapping of directed graphs to labelled acyclic directed graphs. This approach will be discussed later in greater detail.

Given an arbitrary graph (or relation), the problem then to map the nodes (or identifiers) onto the lattice plane (that is, to assign integer coordinates to them), and to display the result in such a way as to exhibit the original structure of the graph as much as possible, while also making it convenient for a user to inspect, browse through, and manipulate the resulting representation.

The ISI Grapher is an implementation of a solution to this problem: it is a set of functions which converts a given arbitrary directed graph into an equivalent pictorial representation, and then graphically displays the resulting diagram. Nodes and edges in the abstract graph now become boxes and lines on the workstation screen, and the user may then interact with the Grapher in various ways via the mouse and the keyboard. The ISI Grapher is both powerful and extendible, allowing an application-builder to easily and comfortably build other tools on top of it.

5. Salient Features of the ISI Grapher

Other graphers and browsers exist, so the salient features of this system are now enumerated:

Portability - The ISI Grapher is implemented strictly in Common LISP, except for a tiny bottom-layer having to do with low-level graphics. This makes the ISI Grapher very portable. The ISI Grapher already runs on several versions of TI and Symbolics workstations/environments, with only about a dozen lines of code of difference between the two implementations! These few differences are managed through conditional compilation, so that the same file compiles and runs as-is on all machines.

Speed - The ISI Grapher has a graphing speed of over 2,500 nodes/edges per minute (of real time, on a Symbolics 3600 workstation with garbage-collection turned off), almost an order-of-magnitude improvement over other systems. Moreover, the asymptotic time behavior of the ISI Grapher increases only linearly with the size of the graph being drawn. This was achieved through careful design of the data structures and the layout algorithm.

Nice layout - In addition to being time efficient, the layout algorithm employed by the ISI Grapher compares favorably with the output of layout algorithms employed by other graphers. Figures 1 and 2 at the end of the paper illustrate typical ISI Grapher displays.

Versatility - The ISI Grapher interfaces to other system tools, such as the editor and the inspector. This allows for a more uniform environment for the user/application-builder.

Extensibility - The design of the ISI Grapher allows other applications to be built on top of it quickly and elegantly. Several such tools will be described later. This is a very important property, because graph structures are a recurring theme throughout computer science (in data structures, knowledge bases, grammars, searches, etc.) Thus the usefulness of a system greatly increases when individual researchers can easily tailor it to their specific needs and requirements.

Innovations - The ISI Grapher incorporates several novel features. Chief among them is the linear-time layout algorithm, as well as the "continuous-update" scheme utilized by the global scrolling mode. The latter is designed to sharpen the user's awareness and sense of direction and location while "navigating" through a large graph.

6. Other Graphers and Related Work

A notable effort to produce a graph browser called Grab was put forth in [Meyer] and is further developed in [Rowe et al], where a system to visually display graphs was implemented. Unfortunately for AI researchers, it was written in C. An additional problem was the usage of numerous time-consuming heuristics (to optimize edge-crossings, for example), rendering the system very slow when laying out large graphs.

Another scheme for drawing graphs is proposed by [Lipton et al]. To draw a graph, this scheme entails detecting and exploiting various properties of the given graph with respect to symmetry and the induced automorphism group. While possessing some mathematical elegance, such a scheme can hardly be expected to yield an efficient implementation. It is recognized that systems which run very slowly but optimize layouts to some degree have their applications, but for our purposes, we regard speed as having paramount importance: users are not likely to tolerate layout times measured in hours.

An experimental graph-layout system was produced by the Symbolics Corporation in early 1985 for internal use. However, its heavy dependence on flavors and other specialized Symbolics features, has made it completely non-portable. Additionally, this system used so much space, that attempting to use it on a graph with more than a couple of hundred nodes would typically lead to hopeless disk thrashing (due to massive swapping). In contrast, the ISI Grapher has been successfully used on graphs of up to 25,000 nodes without incident.

7. Invoking the ISI Grapher

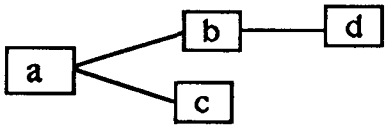
The ISI Grapher is invoked at the top-level by calling the function **graph-lattice** with a list of roots/options and a *sons-function*. This provides a means for the ISI Grapher to deduce the complete description of the graph by recursively calling the sons-function on the roots and their descendents. Next, a reasonable graphical layout is computed for the graph, and is drawn on the display. Various mouse sensitivity and functionality is automatically provided, creating a versatile and user-friendly browsing environment.

7.1. An Example

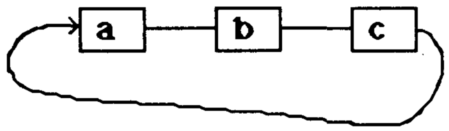
For example, if our graph is {(a,b),(a,c),(b,d)}, our root is {a}, and our sons-function is:

```
(defun sons (x)
   (cond  ((eq x 'a) (list 'b 'c))
          ((eq x 'b) (list 'd))
          (t nil)))
```

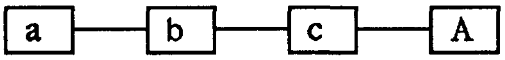
Note that the sons-function returns NIL if and only if the given node is a leaf in the graph (that is, the given node has no children.) Now, the call **(graph-lattice 'a 'sons)** would produce the picture of the graph:



Directed cycles in the graph will be "broken" for displaying purposes by the introduction of "stub" nodes. For example, the graph {(a,b),(b,c),(c,a)} which looks like this:



will be actually displayed as follows:



where "**A**" represents the same graph node as does "**a**", so in a sense the graph node represented by "a" is displayed twice (with an obvious indication that this has occurred, such as the usage of a bolder font; this is automatically provided by the ISI Grapher). All directed edges are displayed with the direction implicitly going from left to right. The first argument to **graph-lattice** may in fact be a command list with a special syntax, allowing selective pruning of the graph; this facility may also be used interactively in various ways.

The cycle-breaking may be viewed as a pre-processing pass on the graph, and operates as follows: a topological sort is initiated, beginning at the roots (the parentless nodes, or else an arbitrary user-specified set of nodes.) A topological sort is an ordering of the nodes of a directed graph so that all the parents of a given node in the ordering appear **before** that node in the ordering. It is well known that it is possible to topologicaly sort the nodes of a directed graph if and only if the graph does not contain any directed cycles. Moreover there are numerous linear-time algorithms to achieve such an ordering when one exist (or detect that none exist if that is the case.)

When the topological sort becomes "stuck" and cannot "proceed" any further on any given node, we have detected a cycle. We now "break" the cycle via the introduction of a "stub" node as discussed above, and continue with the topological sort. We repeat this process until all the nodes in the graph have been processed, thus eliminating all cycles. If one is a little careful in the implementation of this scheme, the total amount of computation required remains linear in the size of the graph.

Once a graph has been layed out and is displayed in a window, various commands are available from the main command menu. This menu is activated by clicking the mouse anywhere inside the currently active Grapher window. If the mouse cursor was pointing to a particular graph node during the mouse click, additional commands (tailored for and directed towards that particular node) shall become available on the main command menu. Appropriate documentation/explanation lines are available at the bottom of the display when the corresponding menu entry is highlighted, and a mechanism is provided that allows the user to customize the menus.

8. Performance and Efficiency

The time required by the ISI Grapher to layout a graph is linearly proportional to the size of the graph[3]. Moreover, the constant of proportionality in this linear relation is relatively small, yielding both a theoretical optimum, as well as practical efficiency. In benchmark runs, speeds of up to 2,500 nodes per real-time minute have been achieved by the ISI Grapher when running on a Symbolics workstation.

It is worth noting that the computational time bottleneck of most graph-layout systems tends to be embedded in the layout algorithm which finds the X and Y positions for nodes on the display. It is further noteworthy that there are numerous algorithms and heuristics to discretely lay-out graphs on the lattice-plane. However, the esthetic criterion that dictate what is a "nice" or "pleasing" layout vary greatly across users, and is very subjective. It can even be shown that under some simple esthetic assumptions, "optimal" layout becomes NP-hard (which in plane language means that polynomial-time algorithms for such layouts are **not** likely to exist) See, for example, [Supowit and Reingold].

The author does not advocate his layout scheme as the final word on such algorithms: rather it is his belief that the layout scheme employed here yields very high returns in terms of esthetic appeal per unit computation time, and is also quite simple to describe. For other layout schemes see [Wetherell and Shannon].

9. The Layout Algorithm

The layout algorithm employed by the ISI Grapher has several novel aspects. First, as previously mentioned, the asymptotic time and space performance of the layout algorithm is linear in the size of the graph being processed; this situation is clearly optimal. Secondly, the layout algorithm employed by the ISI Grapher exhibits an interesting symmetry: layout is performed independently in the X and Y directions. That is, first all the X coordinates (of the nodes in the layout) are computed, and then all the Y coordinates are computed **without** referring to the value of any of the X coordinates. This property implies a certain logical "orthogonality" in the treatment of the two planar dimensions, and is the source of the simplicity of the layout algorithm (the heart of the layout algorithm is only about two pages of code).

The Y coordinates of a node N is computed as follows: if N is a leaf node (that is, if N has no children in the graph) its Y coordinate is selected so that is it as close as possible to, but not overlapping any node previously layed out. If N has any children, their Y coordinates are computed first, and then N's Y coordinate is set to be the arithmetic average of the Y coordinates of N's children. Note that the second rule implies depth-first recursion, which is indeed how the algorithm is implemented. The Y-direction

[3] More formally, the asymptotic time (and space) complexity of the ISI Grapher for a graph $G=(V,E)$ is $O(|V| + |E|)$, where $|V|$ is the size of the node set, and $|E|$ is the size of the edge set.

layout is sensitive to the heights of the objects being displayed. On the other hand, the Y-direction layout is completely oblivious to the X-coordinate values.

Similarly, the X coordinates of a node N is computed as follows: if N is a root node (that is, if N has no parents in the graph), its X coordinate is set to zero. If N has any parents, their X coordinates are computed first, and then N's X coordinate is set to be some fixed amount larger than the maximum of the X coordinates of N's parents. Again, note that this implies depth-first recursion. The X-direction layout is sensitive to the lengths of the objects being displayed, and is completely oblivious to the Y-coordinate values.

For the sake of completeness, we specify the X and Y layout algorithms more formally. The layout algorithm for the Y coordinates is specified as follows:

```
For N in Nodes do Y[N] := 0;
Last-y := 0;
For N in Roots(G) do Layout-Y(N);

Procedure Layout-Y(N);
begin
if Y[N] = 0 then                    /* N was not yet layed-out */
        If N has any unlayed-out children then
                begin               /* layout the children first. */
                for C in Children(N) do Layout-Y(C);
                Y[N] := average-Y(Children(N));
                end
        else    begin               /* layout a leaf. */
                Y[N] := Last-y + Height(N);
                Last-Y := Y[N];
                end;
end;    /* of procedure Layout-Y */
```

The layout algorithm for the X coordinates is specified as follows:

```
For N in Nodes do X[N] := 0;
For N in Leaves(G) do Layout-X(N);

Procedure Layout-X(N);
begin
if X[N] = 0 then                    /* N was not yet layed-out. */
        If N has parents then
                begin               /* layout the parents first. */
                for C in Parents(N) do Layout-X(C);
                X[N] := Max{X[i] + Width(i) | i in Parents(N)} + constant;
                end
end;    /* of procedure Layout-X */
```

From the recursive layout scheme specified above, it should be clear that each node gets processed only once during the two independent passes (one for each of the two planar axes.) What is not so obvious from this discussion, however, is whether such layouts actually appear pleasant given real graphs. This question is best answered via inspection of some examples, such as the ones included at the end of this paper.

10. Portability and Code Organization

In trying to keep the ISI Grapher as portable as possible, the code is divided into two main modules. The first and largest module consists of pure Common LISP code; this code is responsible for all the layout, control, and data-structure manipulation algorithms. The second module is substantially smaller, and consists of numerous low-level primitive calls which are quite likely to be implementation-dependent. The intent here is that when the Grapher is to be ported to another (Common LISP) environment, only the second module should require modification. In order to further minimize porting efforts, the calls from code in the first module to functions in the second module were designed to be as generic as possible.

In summary, if a new environment has a window-system which supports a reasonable set of window and graphics primitives (such as open-window, draw-line, print-string, etc.), then porting the ISI Grapher to this new environment or machine should require a minimal coding effort, probably all of which would be confined to the second section of the ISI Grapher code.

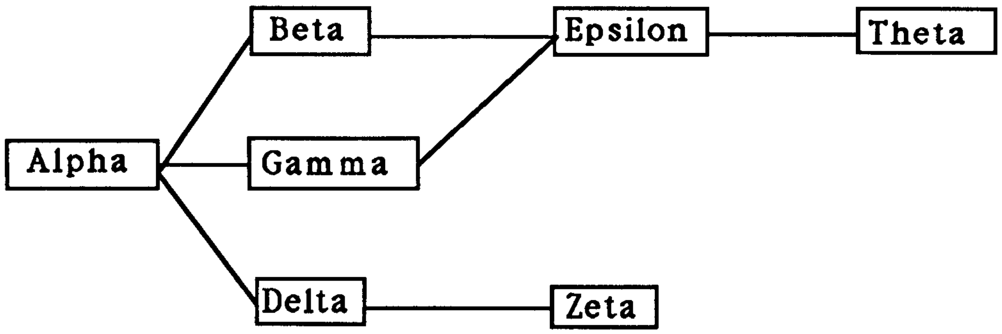
11. Existing Applications

To demonstrate how easily other applications may be built on top of the ISI Grapher, several such applications have already been built and are provided alongside the ISI Grapher. We now describe some of these applications:

The List Grapher - This application displays the natural correspondence between lists and trees. For example, the call

```
(graph-list  '(alpha(beta (epsilon theta))
                          (gamma epsilon)
                          (delta zeta)))
```

would produce the following picture:



The List Grapher provides an easy means of quickly obtaining large or complex graphs.

The Flavor Grapher - This application displays the interdependencies between LISP "flavors," where nodes are flavor names, and edges mean "depends on." This type of a diagram could be quite useful in system development. For example, the call **(graph-flavor 'tv:window)** would graph all the flavors that depend on the tv:window flavor.

The Package Grapher - This application produces a picture of the package interdependencies between a Common LISP package and all packages which use it. An example of a call is **(graph-package "global")**.

The Divisors Grapher - This application displays the divisibility graph of a given integer; that is, all the divisors of an integer are represented as nodes, where an edge between two nodes means "is divisible by." This is also a quick method to produce large graphs. An example of such a call would be **(graph-divisors 360)**.

Coding and testing the above 3 tools (the Flavor Grapher, the Package Grapher, and the Divisibility Grapher) took only half an hour of work!

The NIKL Browser - This application is a browsing tool for NIKL networks and graphs a concept taxonomy below a given concept list. Concepts are fundamental objects in NIKL, and are partially ordered by subsumption (i.e. set inclusion). NIKL is a knowledge representation environment developed at ISI and is a popular tool in artificial intelligence research [Robins, 1986] [Kaczmarek, Bates, and Robins].

Other applications include the Function Grapher (which draws function-call hierarchies based on lexical scoping,) and the Loom Grapher (Loom is the successor to NIKL.)

12. Extendibility and Overriding Default Operations

Several basic Grapher operations may be controlled via the specification of alternate functions for performing these tasks. These operations include the drawing of nodes and edges, the selection of fonts, the determination of print-names, pretty-printing, and highlighting operations. Standard definitions are already provided for these operations and are used by default if an application-builder does not override them by specifying his own functions for performing these tasks.

For example, the default method of highlighting a graph node when the cursor points to it on the screen, is to invert a solid rectangle of bits over the node. Suppose that the user is not satisfied with this mode of highlighting and would like to have thin boxes drawn around highlighted nodes instead. He may then write a highlighting function that does exactly that, and tell the Grapher to use that function whenever a node needs to be highlighted. The details and semantics of this process are fully described in [Robins, 1987].

As another example, suppose the user is not satisfied with the way nodes are displayed on the screen; ordinarily nodes are displayed on the screen by printing their ASCII print-names at their corresponding screen location, but the user would prefer that some specialized icon be displayed instead. The user may then specify his icon-displaying function as the normal node-painting function and from then on, whenever a node needs to be displayed on the screen, that function will be called upon (along with arguments corresponding to the node, its screen location, and the relevant window) to achieve the desired effect.

In particular, the following basic Grapher operations may be overridden by the user:

- Deciding which font should be used to display an object's print-name. Different fonts may thus be used to distinguish various types of objects.

- Determining the dimensions (width and height) of an object. This information is used by the other Grapher functions, such as the layout algorithm (as placement of objects is sensitive to their sizes) and highlighting operations (as the size of the highlight-box depends on the size of the object being highlighted.)

- Determining the ASCII print-name of an object.

- Highlighting and unhighlighting an object. This operation is most often performed when the mouse points to a given object.

- Describing or explaining an object. This is the function that gets executed when the corresponding explain (or pp) command is selected from the main menu.

For each one of the categories above, the Grapher keeps a *function precedence list*, consisting of a primary function, a secondary function, a tertiary function, and so on, for as many functions as are currently available to perform the task associated with that particular category. Whenever a new function is introduced to perform a certain task, each function is "demoted" one "notch" in precedence. Each category also is associated with a default function, which is initially the only function associated with that category. The default function for a particular category has the least precedence.

When a certain task needs to be performed during the normal operation of the Grapher, the corresponding primary function is called with a graph node object[4] and a window. It is then up to the called function to perform the given task and return an answer of non-NIL if it indeed performed the said task, or return an answer of NIL if it did not (or could not or chose not to) perform the said task. In the former case the Grapher merrily goes about its business, while in the latter case, the secondary function is similarly called, with this process repeating until some function has successfully performed the given task (this event being signaled by the return of non-nil by that function.), or until all the available functions have been exhausted and the task has not yet been performed. In the latter case the default function is called, the default function being guaranteed to perform the associated task successfully.

This mechanism gives the user great flexibility in displaying and highlighting

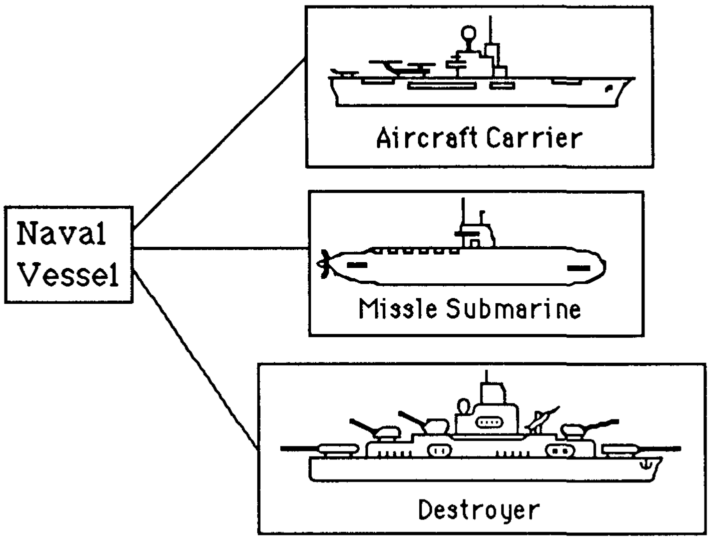
[4] This is a complex structure from which a lot of other information may be extracted.

graph objects. These operations may depend heavily on the type and size of the object being displayed or highlighted, and so different functions may be used to handle each type of object. It should be noted that this discussion implies the ability to mix various types of objects in the same graph (each having unique size, appearance, and highlighting characteristics) with relative ease and uniformity. This scheme is reminiscent of a primitive *flavor* mechanism, where "inheritance" has a non-standard semantics.

In summary, many of the basic Grapher operations are parametrized by a set of default methods. This set may be extended by the application-builder in order to make the ISI Grapher behave in ways not provided for by the author. Any operations left unspecified by the application-builder will default to some reasonable pre-defined method. This scheme results in a portable, flexible, and extendible system.

13. Icon Displays

There are numerous ways to make ISI Grapher displays even more visually striking. For example, the user could utilize icons to display nodes, whereupon the BBN Naval Model (a NIKL network depicting a naval scenario) could take on the style of the following diagram:



An example of an icon-based ISI Grapher display

This may be accomplished by using a font-editor to create a specialized font which would include the above icons as special characters. As the ISI Grapher is capable of working with arbitrary fonts, the above display would readily be achieved by adding the proper (trivial) node-paint function.

13. Hardcopying

Hardcopying is system- and device-dependent, but the ISI Grapher does provide a mechanism which automatically scrolls the current window incrementally in the X and Y directions and calls the proper system function (that is responsible for the actual hardcopying of that portion of the graph which is currently visible in the current window.) The idea here is that since most hardcopying devices are capable of producing an image of only a small (page-sized) bitmap, it is necessary to hardcopy small sections of it one at a time, and then cut-and-paste the resulting "jigsaw-puzzle" together to create the final wall-sized diagram (which may be several square meters in area). The automatic scrolling also provides a small overlap margin between adjacent panes which has proved to be quite handy during the final cutting-and-pasting process.

In summary, the ISI Grapher provides an automatic means of scrolling in order to hardcopy a graph in small sections, but the environment is responsible for providing a hardcopying function which can properly hardcopy each section.

14. Conclusion and Further Research

In summary, the fundamental motivation which inspired the ISI Grapher is the belief that being able to quickly display, manipulate, and browse through graphs may greatly enhance the productivity of a researcher, both quantitatively and qualitatively.

We have shown that various applications can benefit greatly from an interactive grapher-like facility, and then we described an implementation of a prototype, the ISI Grapher. Some of the novel features of the ISI Grapher include its linear-time layout algorithm, its portability, and its extensibility. Although our implementation is in a high-level programming language (Common LISP), in future user interface designs for personal-workstations, it would be preferable to have a Grapher facility available at the window-system level alongside the other primitives of the environment. We believe that the usefulness and applicability of such a graphing facility merits this commitment.

Further research may concentrate on using some heuristics in order to further optimize various properties of the layout, although it is certainly not clear how much of an improvement in the layout can really be achieved while preserving the linear-time complexity of the layout algorithm. It would also be interesting to port the ISI Grapher to other machines and environments. Several such ports are currently in progress; the ISI Grapher already runs on several versions of Texas Instruments Explorers and Symbolics workstations. Finally, numerous extensions to the ISI Grapher are possible, and it is encouraged that applications begin to use the ISI Grapher, or other grapher-like tools, as a building-block in their user interface.

15. Obtaining the sources

Further documentation [Robins, 1987], as well as the source code for the ISI Grapher may be obtained by contacting the author: Gabriel Robins, Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, Ca, 90292-6695, U.S.A.; ARPAnet address: "gabriel@vaxa.isi.edu". The author has already received and responded to a large number of requests for the source code and for the full documentation/manual.

16. Acknowledgements

The author is grateful to the Intelligent Systems Division Director Ron Ohlander, for providing excellent leadership, as well as interproject support funds for further development effort. The supervision and advice of Larry Miller is greatly appreciated. In addition, the following individuals deserve credit for various suggestions and comments: Bob MacGregor, Bob Kasper, Ray Bates, Norm Sondheimer, Robert Albano, Tom Galloway, Steve Smoliar, Neil Goldman, and Eli Messinger. The patient help of Leslie Ladd and Larry Friedman with tedious photocopying, binding, and pasting is gratefully acknowledged. Ching Tsun Chou carefully proofread this paper, and made numerous valuable suggestions; further thanks for proofreading go to Victor Brown, Tom Galloway, and Ann Bettencourt. Finally, the author is especially indebted to Tom Kaczmarek, under who's leadership the ISI Grapher was initially born.

17. Bibliography

Kaczmarek, T., Mark, W., Wilczynski, D., The CUE Project, Proceedings of SoftFair, July 1983.

Kaczmarek, T., Bates, R., and Robins, G., Recents Development in NIKL, AAAI, Proceedings of the Fifth National Conference on Artificial Intelligence, August 1986.

Lipton, R., North, S., and Sandberg, J., A method for Drawing Graphs, ACM Computational Geometry Conference Proceedings, June 1985, pp. 153-160.

Meyer, C., A Browser for Directed Graphs, Technical Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Papadimitriou, C., and Steiglitz, K., Combinatorial Optimization, Algorithms and Complexity, Prentice-Hall, New Jersey,1982.

Reingold, E., and Tilford, J., Tidier Drawing of Trees, IEEE Transactions on Software Engineering, **SE-7**, no. 2, March 1981, pp. 223-28.

Robins, G., The NIKL Manual, Intelligent Systems Division Report, USC/Information Sciences Institute, April 1986.

Robins, G., The ISI Grapher User Manual, ISI/TM-87-197, USC/Information Sciences Institute, September, 1987.

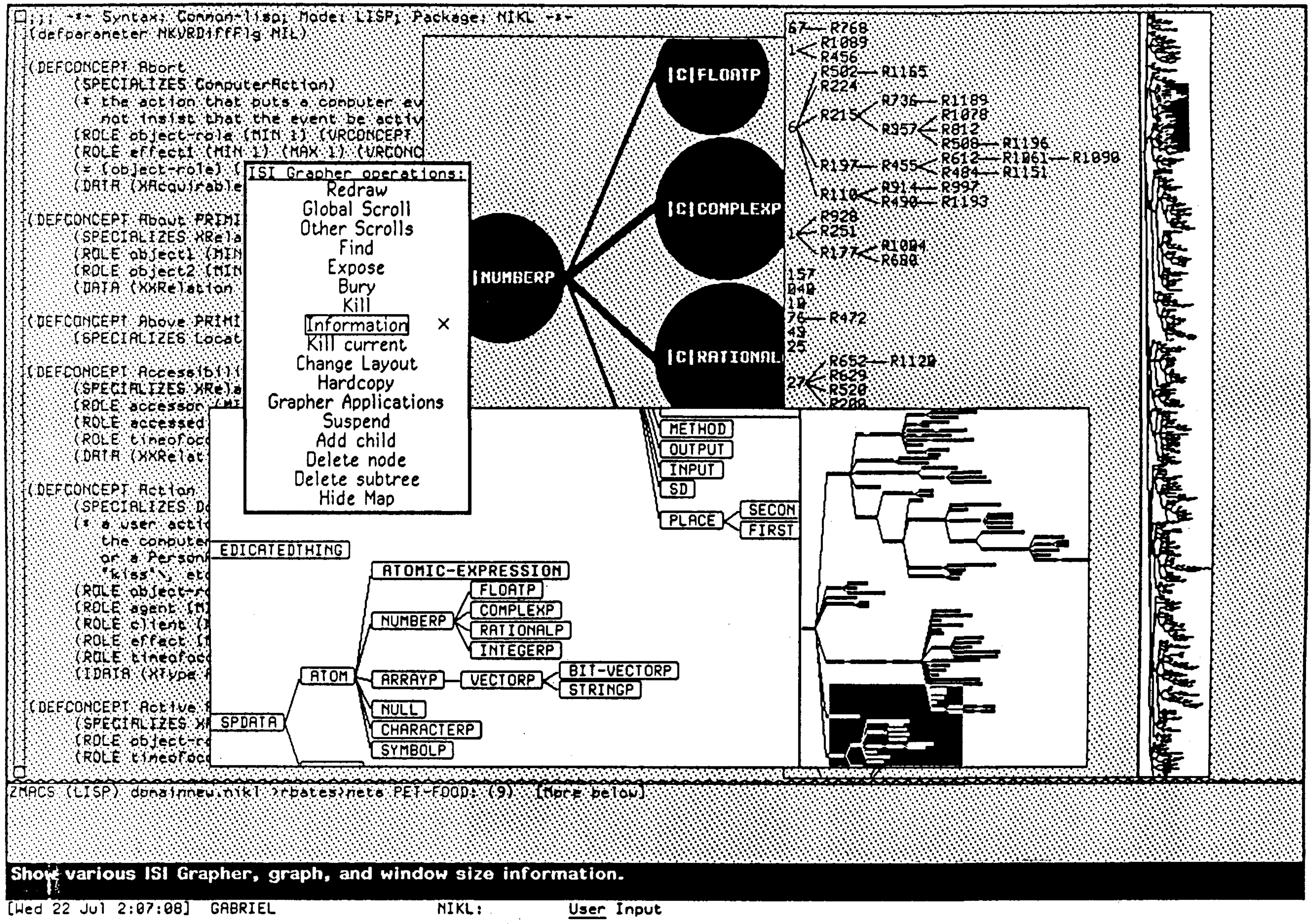
Rowe, L., Davis, M., Messinger, E., Meyer, C., Spirakis, C., and Tuam, A., A Browser for Directed Graphs, Softwa2re - Practice and Experience, **17**(1), January 1987, pp. 61-76.

Supowit, K., and Reingold, E., The Complexity of Drawing Trees Nicely, Acta Informatica, **18**, 1983, pp. 377-392.

Vaucher, J., Pretty-Printing of Trees, Software - Practice and Experience, **10**, 1980, pp.

553-561.

Wetherell, C., and Shannon, A., Tidy Drawing of Trees, IEEE Transaction on Software Engineering, 5, September 1979, pp. 514-520.

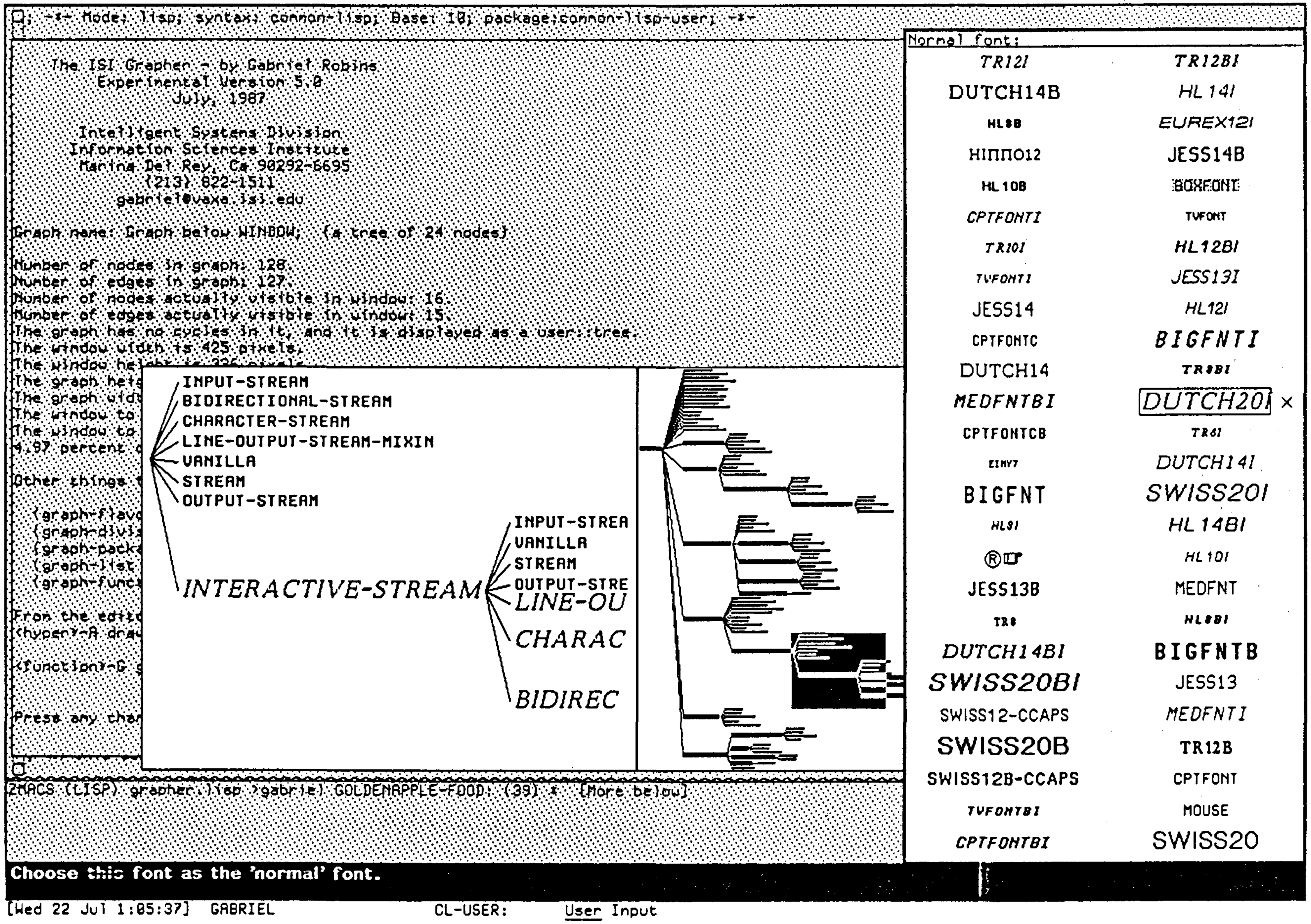


```
;;; -*- Syntax: Common-lisp; Mode: LISP; Package: NIKL -*-
(defparameter NKVRDiffFlg NIL)

(DEFCONCEPT Abort
     (SPECIALIZES ComputerAction)
     (* the action that puts a computer ev
        not insist that the event be activ
     (ROLE object-role (MIN 1) (VRCONCEPT
     (ROLE effect1 (MIN 1) (MAX 1) (VRCONC
     (= (object-role) (
     (DATA (XAcquirable

(DEFCONCEPT About PRIMI
     (SPECIALIZES XRela
     (ROLE object1 (MIN
     (ROLE object2 (MIN
     (DATA (XXRelation

(DEFCONCEPT Above PRIMI
     (SPECIALIZES Locat

(DEFCONCEPT Accessibil
     (SPECIALIZES XRela
     (ROLE accessor (MI
     (ROLE accessed
     (ROLE timeofocc
     (DATA (XXRelat

(DEFCONCEPT Action
     (SPECIALIZES Do
     (* a user actio
        the computer
        or a Person
        "kiss"\, etc
     (ROLE object-r
     (ROLE agent (M
     (ROLE client (
     (ROLE effect (
     (ROLE timeofocc
     (IDATA (XType

(DEFCONCEPT Active
     (SPECIALIZES X
     (ROLE object-r
     (ROLE timeofocc
ZMACS (LISP) domainnew.nikl >rbates>meta PET-FOOD: (9)  [More below]
```

Show various ISI Grapher, graph, and window size information.

```
[Wed 22 Jul 2:07:08]  GABRIEL             NIKL:        User Input
```

Figure 1: The ISI Grapher environment: the foreground window displays a NIKL concept taxonomy, while the Zemacs text editor is visible in the background . Global maps are used to highlight the portion of the graph currently visible in more detail. The main command menu is also visible.



```
; -*- Mode: lisp; syntax: common-lisp; Base: 10; package:common-lisp-user; -*-

    The ISI Grapher - by Gabriel Robins
        Experimental Version 5.0
              July, 1987

      Intelligent Systems Division
     Information Sciences Institute
      Marina Del Rey, Ca 90292-6695
           (213) 822-1511
        gabriel@vaxa.isi.edu

Graph name: Graph below WINDOW;  (a tree of 24 nodes)

Number of nodes in graph: 128.
Number of edges in graph: 127.
Number of nodes actually visible in window: 16.
Number of edges actually visible in window: 15.
The graph has no cycles in it, and it is displayed as a user::tree.
The window width is 425 pixels.
The window he
The graph hei
The graph wid
The window to
The window to
4.97 percent

Other things

  (graph-flav
  (graph-divi
  (graph-pack
  (graph-list
  (graph-func

From the edit
<hyper>-A dra

<function>-G

Press any cha

ZMACS (LISP) grapher.lisp >gabriel GOLDENAPPLE-FOOD: (39) *  [More below]
Choose this font as the 'normal' font.
[Wed 22 Jul 1:05:37]  GABRIEL            CL-USER:       User Input
```

Figure 2: Changing fonts interactively in the ISI Grapher; the foreground window is a scrollable font-selection menu, while the graph itself is a Lisp flavor hierarchy. After a font change, the layout is updated automatically.

INFORMATION
SCIENCES
INSTITUTE

4676 Admiralty Way/Marina del Rey/California 90292-6695