CHAPTER 2
A SURVEY OF VISUAL PROGRAMMING

2.1. INTRODUCTION

The field of visual programming has been active since 1969, and many different systems have been built to both create and visualize computer programs. Our goal in this chapter is to identify and examine the most important of these systems. It will become clear that all systems are drawing on a common set of visual conventions.

We find that an understanding of diagramming gives insights that will be useful in later chapters when we discuss the limits of graphic representation. Therefore this survey includes many examples of diagrams from the computer science literature, as well as examples of work in the field of visual programming. We have also defined the set of conventions that recur in all computer science diagrams.

2.1.1. Terminology

Shu (1988) defines Visual Programming as:

> The use of meaningful graphic representation in the process of programming.

This definition is too general - we suggest:

> The use of diagrams in the process of programming

For *diagram* we start with James Maxwell's definition (Encyclopedia Brittanica, 11th edition):

> a figure drawn in such a manner that the geometrical relations between the parts of the figure illustrate relations between other objects.

To be more accurate, we change the term *geometrical relations* to *geometrical or topological relations,* as in most of the diagrams shown here, it is connections rather than distances that are significant.

We contrast graphic representation with textual representation, which may name relations, but does not illustrate them. Other texts make the point that visual programming makes use of two dimensions, while textual representation makes use of one. Yet the dimensional distinction does not communicate what we regard as the main difference - a diagram *shows* relations, while a text *names* them. In a basic sense it is the difference between a visual mode of expression and a mode of expression that, of all our senses, is closest to the auditory.

The *process of programming* includes both the visual creation of programs and the visualization of programs executing. These two fields share many of the same problems, such as how to represent data structures. We believe that the visual creation of programs is the harder problem of the two, and put much of our concentration on this area.

2.1.2. Major Sources

There are five important source books relating to the topic. The first is Shu (1988), a book-length survey of the field with many diagrams and illustrations. The second book is a Springer-Verlag Lecture Notes in Computer Science, called *Visualization in Programming,* edited by Gorney (1986). The third is a book edited by Chang (1986). Two collections of papers contain many of the notable articles in the field: *Visual Programming Environments: Applications and Issues*, and *Visual Programming Environments: Paradigms and Systems*, both by Ephraim P. Glinert (1990).

The Ph.D. theses by Raeder (1984), Glinert (1985), Brown (1987), and Curry (1978) all contain survey chapters. There is a special issue of *Computer* devoted to the topic (August 1985). The Internet news group comp.lang.visual contains announcements and current debate.

In the realm of diagrams, the observations here are influenced by the thinking of Peirce (1934) and Nadin (1984). Also, the work of Tufte (1983) is an important source for clear thinking about graphic representation.

2.1.3. History

Visual programming, like much else in computer science, has a relatively short direct history. Yet the visual part of visual programming owes much to conventions for representing information that go back much longer. As our later discussion will involve issues of representation, we now take a short look at the history of diagrams, then discuss the origination of visual programming.

2.2.1. History of Diagrams

The earliest abstract illustrations are maps. Maps as navigation tools are rooted in a metric space, meaning that there is a relationship between the physical distance between two locations in the world and the physical distance between the representations of the locations on paper. Yet maps by their nature abstract out detail. Roads are often represented as straight lines, rather than pairs of curved lines. Coastlines are abbreviated. The concept of mapping extends to the field of medical illustration, where the terrain of the human body is abstracted and mapped.

Geometric diagrams also have a long history. Manuscripts containing proofs with diagrams of the Pythagorean theorem exist from the time of the ancient Greeks. These diagrams are also based on a concept of metric space, intended to prove theorems concerning distance in Euclidean space. Descartes' Cartesian coordinate system in the 17th century rationalized the use of graphs in analytic geometry. And scientific discoveries in physics and chemistry resulted in increasingly abstract diagrams.

The history of topological diagrams is difficult to trace. Tree hierarchies show up in the middle ages as a way of documenting lineage. Religion and cosmology of the middle ages and renaissance made great use of diagrams, including graphs, to show the relationships between concepts. Gardener (1982) and Yates (1982) document the combinatorial diagrams of Raymond Lull in the 13th century. In the Renaissance, the rediscovery of Greek thought resulted in many diagrams of philosophical and scientific concepts, as in the following illustration.

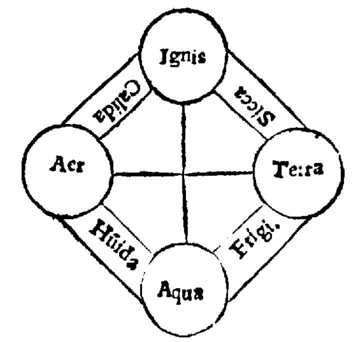


Figure 2.1. Sixteenth century representation of the four basic elements. The qualities the elements share are labeled in the edges. Opposites are joined by the unlabeled edges. From Charles de Bouelles, Liber de generatione. Reproduced from Heninger (1977).

In the 19th century, the work of Boole inspired the invention of Venn diagrams. Also in that period, Charles Peirce proposed extensions to these diagrams and invented his own system known as existential graphs. Notwithstanding the 18th century graph-theoretic discoveries of Euler, it has been in this century that the explosion of abstract mathematics has resulted in the proliferation of tree and graph representations. Now, even subway routes are represented as topological diagrams rather than metric maps.

The Origins of Visual Programming

Flowcharts are the first and best known diagrams of software. Goldstine (1972: 266) claims he created the first flowchart for computers in 1947, while he was working with Von Neumann. Yet these early charts were entirely decoupled from the computer itself. It wasn't until the creation of graphic display technology in the 1960s that such a coupling became possible.

W. Sutherland, in 1966, created the first interactive visual programming language:

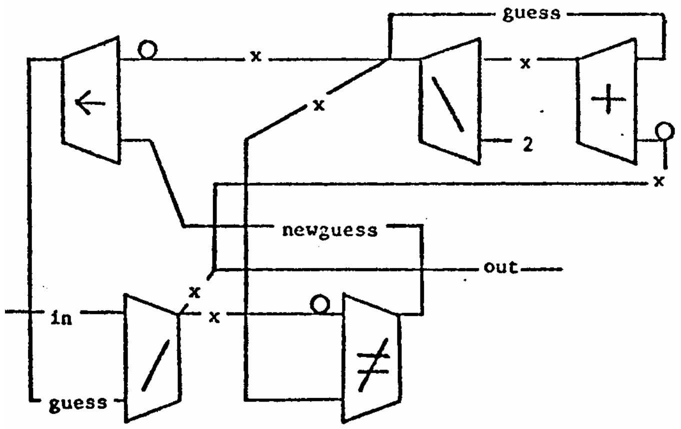


Figure 2.2. Sutherland's diagram for calculating a square root. Reproduced from Curry (1978).

As we will see, it is notable how much current visual programming systems still resemble this first example of Sutherland's. In 1969, researchers at Rand created a language based on flowcharts (Ellis 1969).

Starting in the early 1970s, researchers at Xerox Parc created the first visual programming environments. Bitmapped graphics, mice, and window systems can be mainly credited to this research laboratory. The culmination of the work came in the form of Smalltalk, an operating system/programming environment/programming language (Goldberg 1983). The present graphic user interfaces differ little in concept from the Xerox Parc vision. In the 1980s, Apple Computer, Sun Microsystems, and M.I.T. (X Windows) spread graphic user interfaces to researchers and consumers; recently, the creation of working windowing systems for PCs (Microsoft, IBM, NeXT) has created a flurry of systems based on graphic user interfaces. In the language realm, a commercial version of PROGRAPH exists on the Macintosh platform.

THE VISUAL BASIC PHENOMENON

When many think of visual programming languages, the immediate association is to Visual Basic, a Microsoft product. By our definition, it is not a purely visual language. Instead of being based on diagrammatic representation, its underlying language is an enhanced textual version of the Basic language. In front of this textual language is a well-thought-out graphic user interface, which allows the programmer to construct windows and all their corresponding components such as buttons, slider bars, and menus, by selecting graphic icons and dragging them onto a graphic representation of a window. The programmer then writes textual source code fragments that are essentially event handlers for the different possible mouse and keyboard events. This code is linked to the graphic representation of the window, so that instead of scrolling through long files of source, a programmer can access relevant code by clicking on a physical location. In other words, the interface provides, first of all, a way of constructing the framework of a user interface by manipulating graphic objects, and, second of all, a way of spatializing access to the textual code that needs to be written. Part of the success of Visual Basic is the flatness of the language - many verbs are provided, and many software vendors have been encouraged to create modules that add more verbs. As we will discuss in chapter 7, the hybrid nature of Visual Basic, along with the flatness of the language, are good design decisions that allow graphic representation to be used in a productive way. But what we are searching for here is whether a more extreme language that is purely diagrammatic is possible.

Visual basic is one among many emerging graphic user interfaces oriented toward programmers. Microsoft makes Visual C++, a graphical user interface closely coupled with a C++ compiler. Borland and PowerSoft also make products which use visual conventions in a way similar to Visual Basic. Computer-Aided Software Engineering (CASE) companies such as Cadre and IDE have been making visual tools for many years, and are now taking advantage of the current interest in object-oriented notations to widen the availability of their products.

2.2. SOFTWARE DIAGRAMMING TECHNIQUES

In this section we discuss the main type of software diagrams. These types form the basis for all visual programming systems.

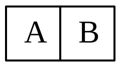
2.2.1. Diagram distinctions

Diagrams portray associations. Associations can be made in three different realms: metric, topological, and symbolic. When we refer to diagrams, we are for the most part referring to associations made in topological space. Yet elements of the other realms are often involved.

Associations in metric space can happen by proximity, as when two cities close on a map are said to somehow be related. In computer systems, above the hardware level, Euclidean space is usually unimportant. But metric space does not have to Euclidean - we can use other scales, the only constraint being some quantitative continuity along a scale. So, as an example, timing diagrams make use of a time metric along the X axis. The tests for a diagram with metric qualities are first, whether there is an implicit scale on either axis, and second, whether modifying positions without modifying connectivity results in a change in meaning.

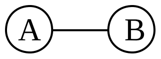
Much more common in software diagrams are associations in topological space. In such diagrams, elements can be imagined joined by rubber bands, and can be slide and stretched without changing meanings. Within topological diagrams associations can happen in three ways. We illustrate this by showing how nodes A and B can be shown to be associated.

The first method is *adjoinment*:



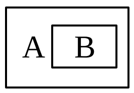
where A and B are cells that physically adjoin - that share a side.

The second method is *linkage*:



where A and B are nodes that are explicitly linked by an edge. Adjoinment can be seen as a degenerate case of linkage - the nodes are so close that the edge cannot be drawn. All graphs and trees are forms of linkage diagrams.

The third method is *containment:*



where one node is contained within another. Containment is most often used to indicate set relationships, as with Venn Diagrams. Containment can be translated to a linkage graph by using a directed edge from the contained node to the container node. This edge can be labeled *is part of*, or *is contained by*.

Besides the metric and the topological, association can also be represented through the symbolic. Using the partially visual mechanism of a matrix, A and B can be linked:

$$\begin{array}{c|cc} & A & B \\ \hline A & & \bullet \\ B & \bullet & \end{array}$$

But the most common way and general way of making an association in the symbolic realm is through naming:

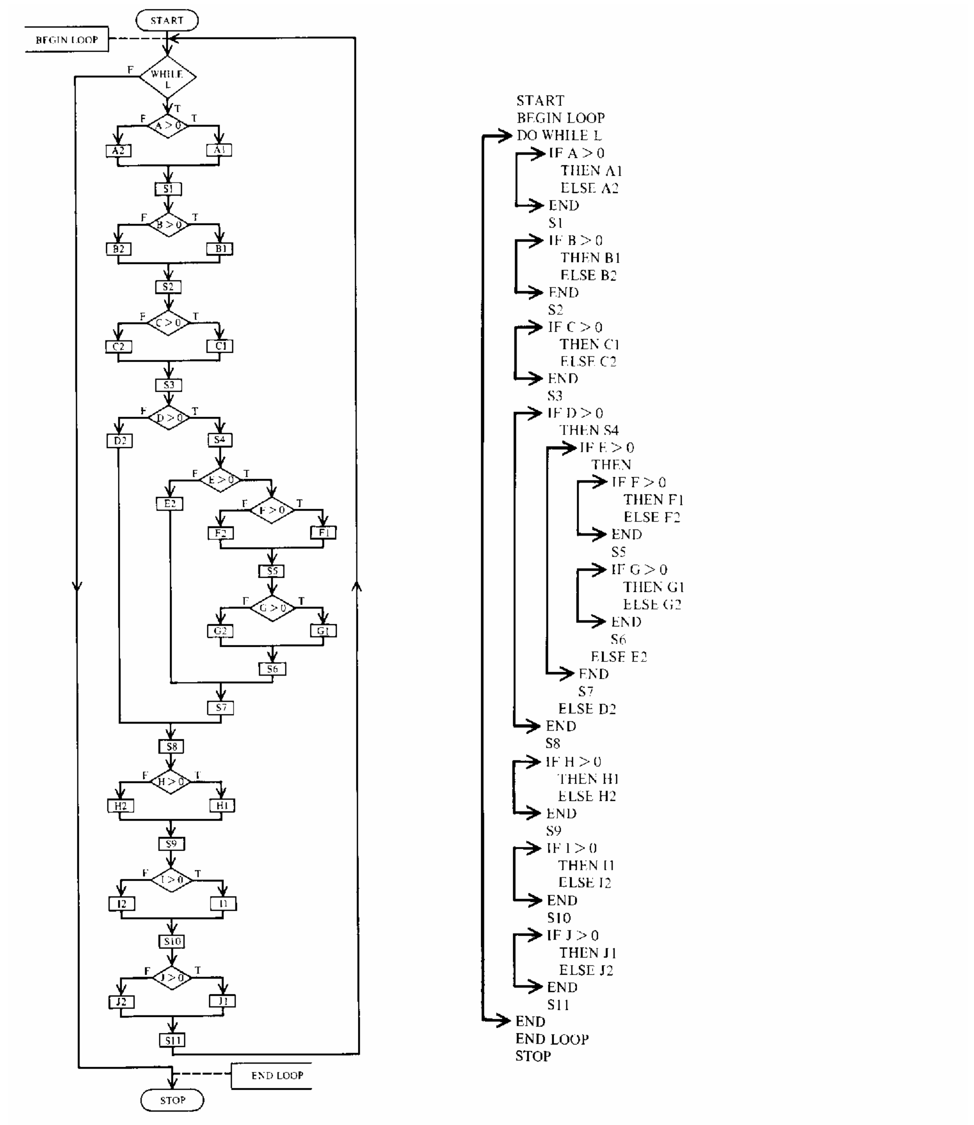
link(a, b).

Any graph can be represented symbolically as a textual list of nodes and edges. And many recursive or infinite structures can only be fully represented in the symbolic realm.

Within the topological realm, it is useful to distinguish between trees, DAGs, graphs, and bipartite graphs. In most graph-like diagrams, nodes may be of different types, often distinguished by shape. Nodes are often labeled. Edges may be directed and undirected, and may also have type, distinguished by either line weight, dottedness, or termination shapes such as arrowheads. Edges are often labeled. Armed with these distinctions, we look at the major ways diagrams are used to portray software.

2.2.2. Flowcharts

Flowcharts are topological, graph-based constructions that often are filled in with program text. The control logic of the program is shown through simple branches and loops:



```
START
BEGIN LOOP
DO WHILE L
  IF A > 0
    THEN A1
    ELSE A2
  END
  S1
  IF B > 0
    THEN B1
    ELSE B2
  END
  S2
  IF C > 0
    THEN C1
    ELSE C2
  END
  S3
  IF D > 0
    THEN S4
      IF E > 0
        THEN
          IF F > 0
            THEN F1
            ELSE F2
          END
          S5
          IF G > 0
            THEN G1
            ELSE G2
          END
          S6
        ELSE E2
      END
      S7
    ELSE D2
  END
  S8
  IF H > 0
    THEN H1
    ELSE H2
  END
  S9
  IF I > 0
    THEN I1
    ELSE I2
  END
  S10
  IF J > 0
    THEN J1
    ELSE J2
  END
  S11
END
END LOOP
STOP
```

Figure 2.3. Flowchart example from Shooman (1983).

In some software methodologies, flowcharts are generated by analysts as a specification to programmers, who then convert the charts into source code. In other cases, flowcharts are generated after the fact, either by hand or automatically.

The usefulness of flowcharts has been hotly debated. Flowcharts for large systems tend to get large and messy, as decisions can have many branches. In order to work around this problem, extensions to flow charts allow for charts to be terminated and then resumed on different pages. Yet, since flow charts require loops to show return arrows, a program with many loops requires return arrows that may span many pages.

2.2.3. Structure diagrams

A structure diagram is a hierarchical, modular breakdown of a program. Between levels on the tree, there are links, with symbols to indicate the sort of information that is being passed back and forth:

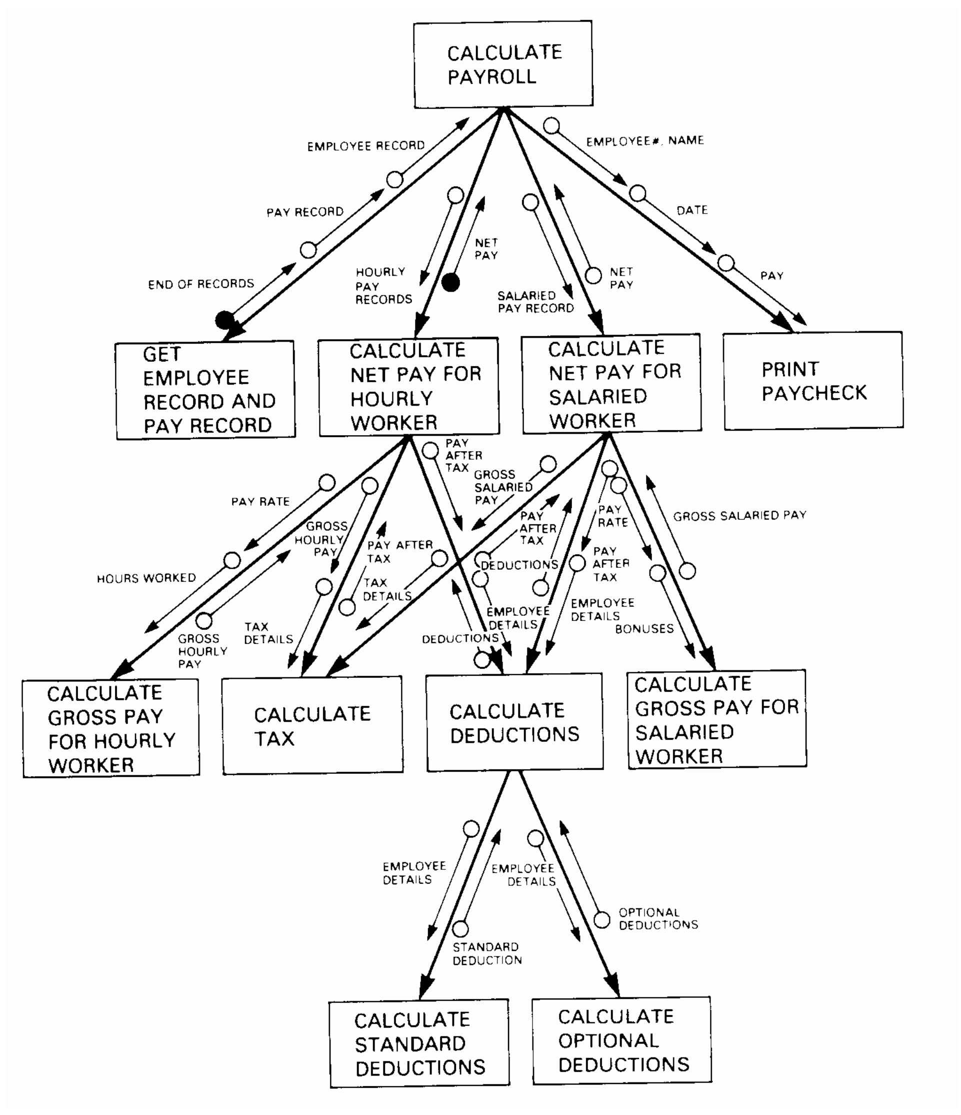


Figure 2.4. A structure diagram from Martin (1985).

These structures are represented either as trees or as directed acyclic graphs (DAGs). The structure chart is usually the end result of the activity known as structured analysis, in which the functions of a system are partitioned in a top-down manner. Note the diagrams are purely topological, with labeled edges and nodes. Note also the difficulty apparent in labeling the edges of such a DAG, even on an small example such as the one shown.

2.2.4. Software Level charts

At a higher level, the functions of a system are often thought of in layers, resulting in the following type of diagram:

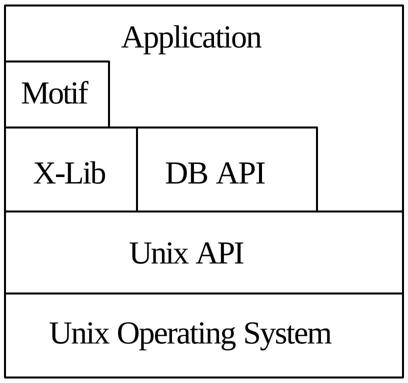


Figure 2.5. Layer diagram.

This diagram represents that, for this system, an application can access Motif, X-Lib, a DB API, and a Unix API. If a layer adjoins a layer below it, then it is allowed to access that adjoining layer.

This sort of diagram will only work on simple access schemes. More complex schemes will result in a complex graph that cannot be represented with adjoining regions. The problem is reducible to a planarity problem by considering regions as nodes and adjoinment as edges; three applications that all can access three libraries is equivalent to a $K_{3,3}$ graph, known to be non-planar.

2.2.5. Structure with Trees: Warnier-Orr diagrams

Many structured analysis techniques result in trees being generated. An alternate way to represent trees is shown by Warnier-Orr diagrams.

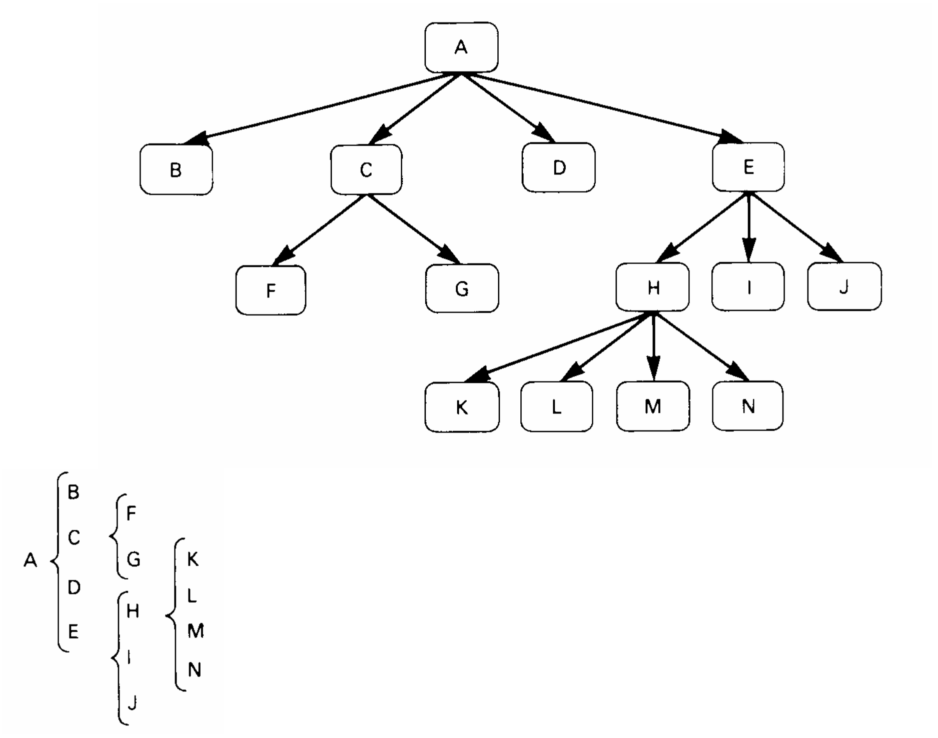


Figure 2.6. A tree structure with its corresponding Warnier-Orr Representation. Reproduced from Martin(1985).

Instead of the root being at the top, as in a normal tree-structure breakdown, the root is indicated in the far left corner, and each succeeding column is at a lower level in the tree. And the use of boxes and lines is reduced. It is possible to produce a Warnier Orr structure entirely without lines:

```
a    b
     c    f
          g
     d
```

Figure 2.7. Simplified Warner-Orr Diagram.

constitutes a Warnier-Orr tree representation of the three leftmost subnodes of A in figure 2.6. Boxes around the nodes and lines between the nodes are understood - the location of the letter in space implies its level in a tree structure. Essentially, when programmers indent code to show nesting structure, a Warnier-Orr convention is being used.

2.2.6. Tree-based Variants of Flowcharts

2.2.3. Rothon Diagrams

Rothon diagrams attempt to resolve the problems of flowcharts. Rothon diagrams treat loops as objects without an arrow back to the top of the loop. Hierarchy is shown by moving left to right through a refinement of the diagram (Brown 1983).

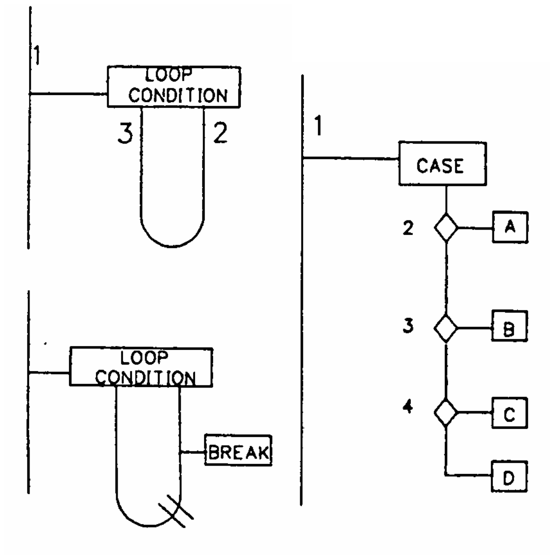


Figure 2.8. Rothon Diagram. From Brown (1983).

2.2.4. Dimensional Flowcharting

Witty (1977) outlines a flowcharting method very similar to Rothon diagrams. Sequential flow is shown along the vertical axis, and parallel constructs are shown along the horizontal axis. In other words, there is simultaneous control flow along horizontal lines.

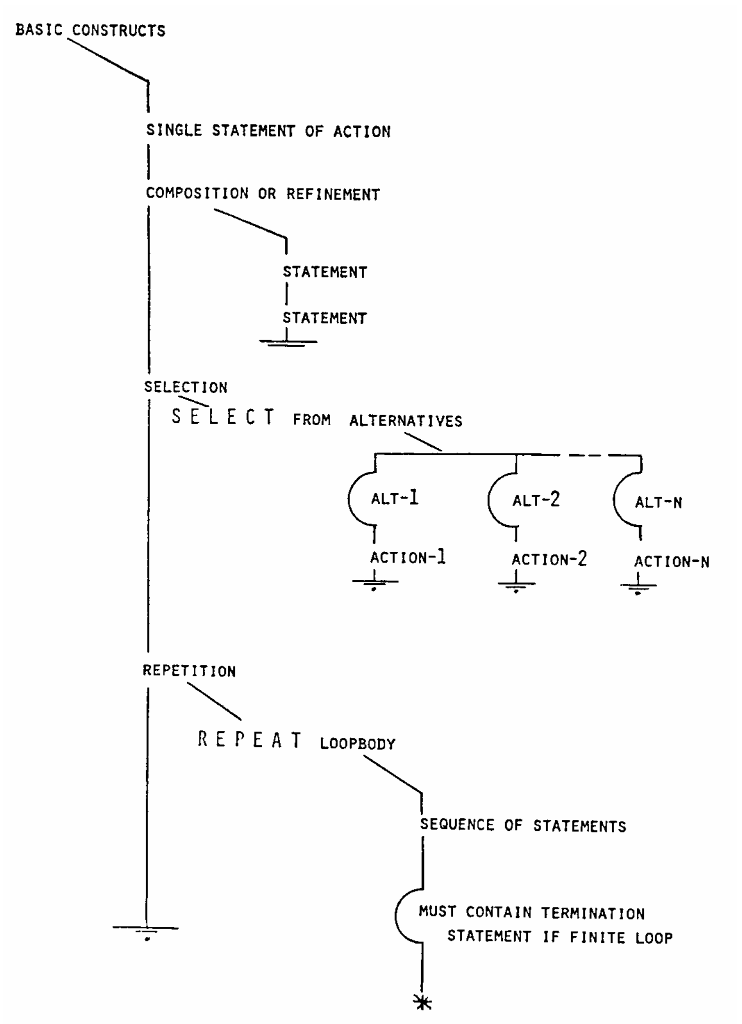


Figure 2.9. Dimensional Flow Chart. From Witty (1977).

Diagonal lines are used to show refinement to lower levels of the hierarchy. One feature of the diagrams is that, on paper, they can be folded up along refinement lines. When one wants to see detail, one unfolds around the statement, giving detail. To insert detail into the paper, one cuts the diagram and insert a new piece.

All flow chart techniques are topological, link-based structures. In almost all cases, they rely on textual programming language statements for conditions and assignment statements. The classic flow chart usually a fairly sparse graph, with most nodes having either one or two outgoing edges. Back edges in the graph are always from loop statements - a loop is the visual equivalent of a branch instruction to an previous program statement. A variety of different node shapes are used to indicate different forms of computer statement. Each statement in a language, or at least each statement block, has its own box. For the most part, the charts flow from top to bottom.

2.2.7. State Transition Diagrams

State transition diagrams are well known in computer science as originating from the study of finite automata. Transition diagrams are used for modeling a variety of event-based computer science domains, including parsing, user interface design, and circuit design. At the applications level, they are used to represent transaction flows, appliance controls, marketing scripts, etc. Edges represents transitions from state to state that occur as a result of an input symbol being read.

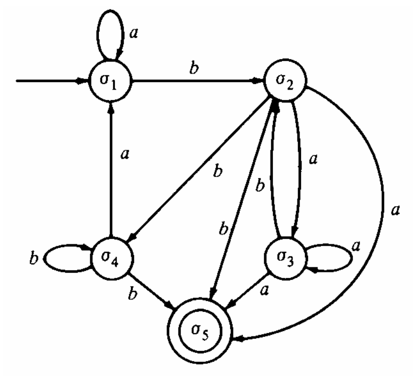


Figure 2.10. Finite State Automata. From Johnsonbaugh (1984).

State transition diagrams tend to look more graph-like than flow charts, meaning that there is usually no predefined axis in the diagram. Also, the degrees of nodes can vary widely, depending on the application. With the exception of special symbols for start nodes and termination nodes, all nodes are the same shape. Nodes and edges are both labeled. Termination nodes are often indicated by two concentric circles - this is to be considered a different type of node, not an instance of containment.

As state diagrams become large, the chances of the diagram being planar decreases. Therefore, large state diagrams are unwieldy, with much attention being required to properly space nodes and edges so that labels can be read unambiguously. Using the convention of H-graphs (see section 2.3.1), some of this complexity can be handled:

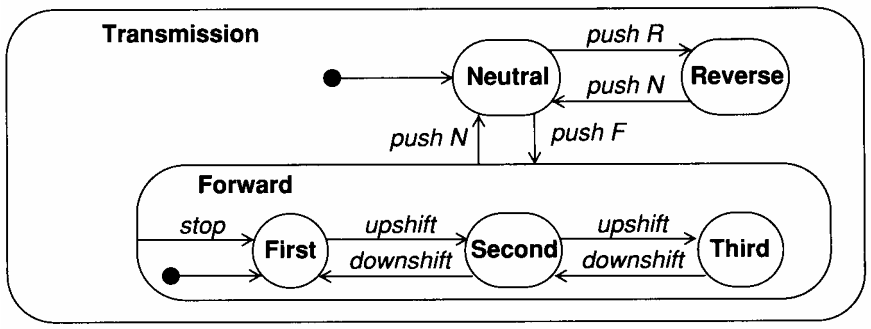


Figure 2.11. State Chart from Rumbaugh (1991).

This diagram uses a convention devised by Harel (1987) called State Charts, in which a state such as *Forward* above, can contain other states, such as *First, Second, Third.*

2.2.8. Nassi-Shneiderman diagrams

In this sort of diagram, hierarchy is shown using the conventions of enclosure and adjacency. The figure below shows the Nassi-Schneiderman representation and the equivalent flow chart.

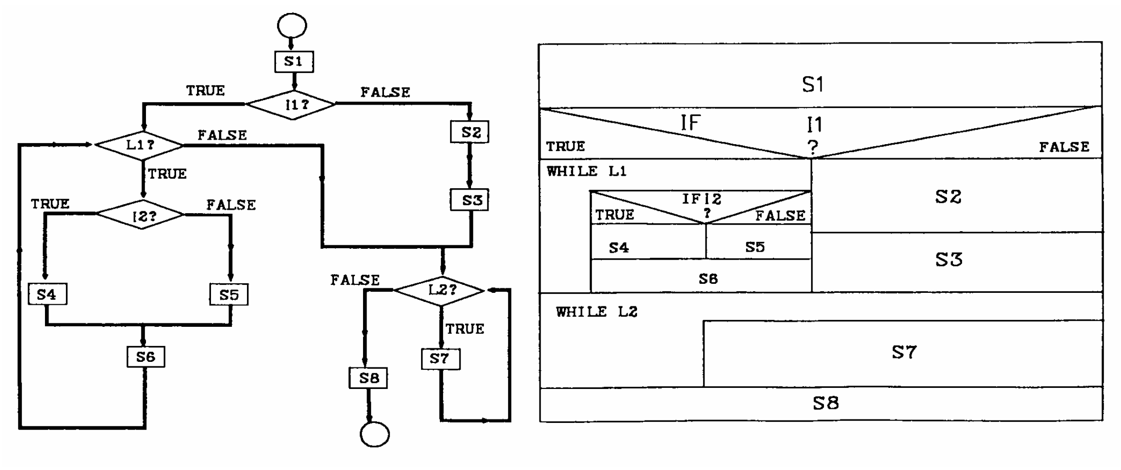


Figure 2.12. Flow Chart vs. Nassi Schneiderman diagram. From Shu (1988).

Decisions are shown by splitting the line into three smaller, parallel boxes. Loops are shown by enclosing a box in a larger box labeled with the loop condition.

As with other adjoinment-based conventions, there is a limit on what they can represent. The early termination of loops and multiple conditionals of some languages can be combined in ways that cannot be represented by these graphs.

2.2.9. Cells and arrows

In a combination of adjoinment and link-based conventions, data structures are often showed as adjacent memory locations linked by pointers:

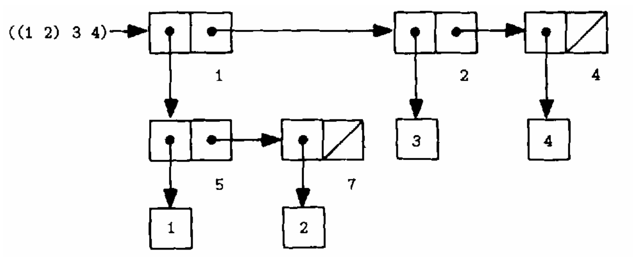


Figure 2.13. Cell and arrow diagram. From Abelson(1985).

Most often this is used for teaching or for program documentation. In programming the manipulation of linked lists, it is customary to think about the manipulation of pointers in a the following manner:

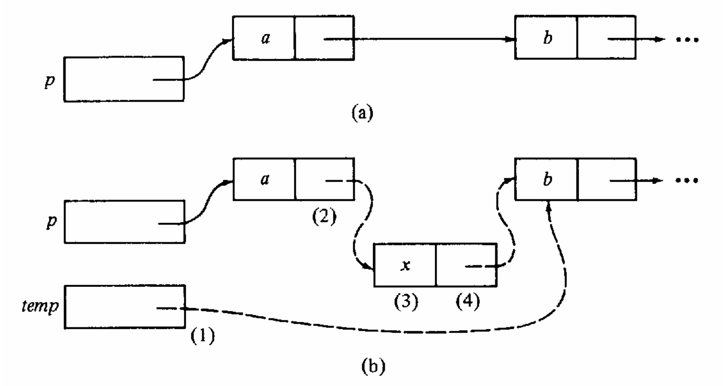


Figure 2.14. Linked list insertion. From Aho (1983).

The accompanying program shows how textual language represents the problem:

```
procedure insert(x:elementtype; p: position);
    var
        temp: position;
    begin
        temp := p^.next;
        new(p^.next);
        p^.next^.element := x;
        P^.next^.next := temp;
    end
```

For the beginning programmer, the program text is confusing without the corresponding diagram, yet the diagram itself does not contain enough information to execute from.

Variations on box-and-pointer constructs are often used to represent the displays and activation records of programming languages:

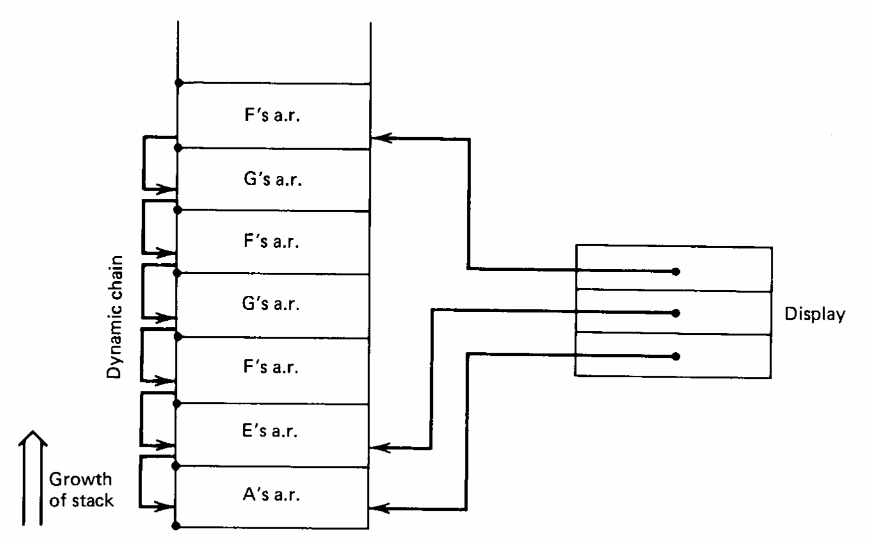


Figure 2.15. Programming language display. From Ghezzi (1982).

2.2.10. Traversal Patterns

In a kind of static animation, diagrams are often used to explain tree and graph traversals:

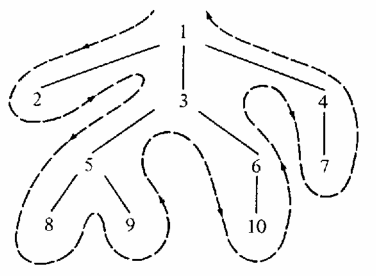


Figure 2.16. Tree Traversal. From Aho (1983).

The arrow that shows the sequence of movement we will refer to as a meta-edge. A meta-edge indicates something about the underlying graph, and is not part of it.

2.2.11. Recursion portrayal

Recursion is not an easy concept to teach or understand. Visual representations are often attempted as part of this effort. For the most part, recursion is represented using containment:

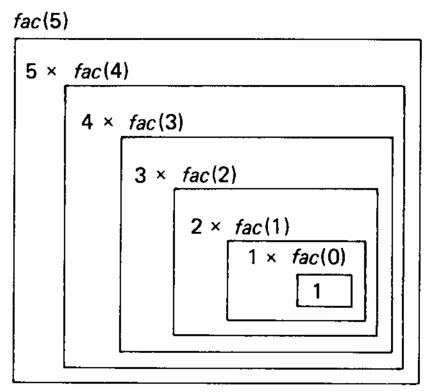


Figure 2.17. Recursion frames. From Bauer (1982).

In some cases, the well-known technique of unwinding recursion through a set of variables or a stack is used so that a recursive problem can be represented with flow charts:

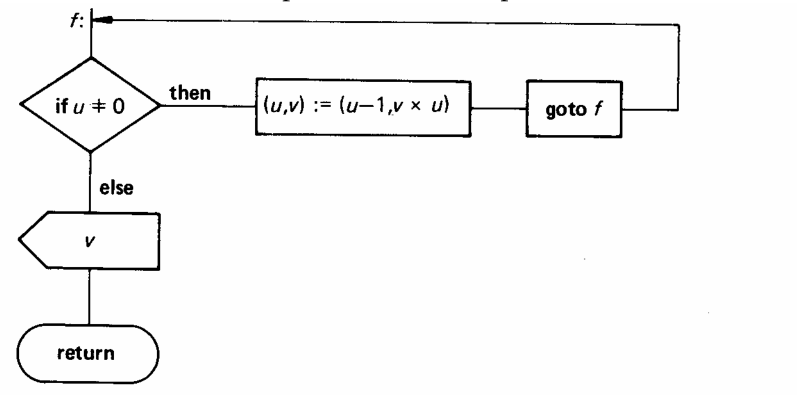
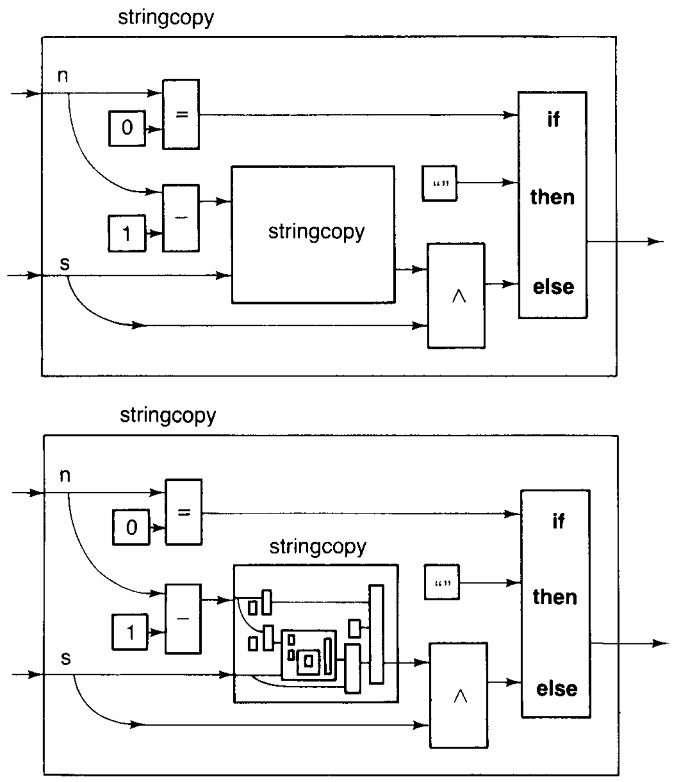


Figure 2.18. Recursion with flow charts. From Bauer (1982).

Without performing this unwinding, it is impossible to represent recursion in a flow chart - recursion is not a visual concept, it is a symbolic one. The following representation suggests the difficulty:



fun stringcopy (n,s) = **if** n = 0 **then** "" **else** stringcopy (n−1,s) ∧ s

Figure 2.19. Stringcopy using enclosure diagrams. From Reade (1989).

In the top diagram, the inner box, stringcopy, must call itself. The bottom diagram shows the only way this can be visually displayed - by using a containment convention, inserting a copy of itself at a smaller scale. This process must continue *n* times - in other words, the physical diagram itself is dependent on one of the input parameters. In order to avoid this, the top diagram is most often used - it makes use of the power of the symbolic realm by naming the inner box the same as the outer box.

This inability of the visual to represent recursion without recourse to the symbolic is a warning sign about the limits of the visual. The visual cannot refer to itself in the same way as the symbolic can.

2.2.12. Object-Oriented Analysis

The representation of data is often accomplished using diagrams. The following diagram shows two different conventions - Entity-Relationship (Chen 1976) and the Object Modeling Technique (Rumbaugh 1991). ER diagrams are extremely simple - there are three sorts of nodes. Entities and Relations form a bipartite graph. Entities and Relations can both have associated Attributes. (A similar convention called conceptual graphs is described in the work of Sowa (1984)).

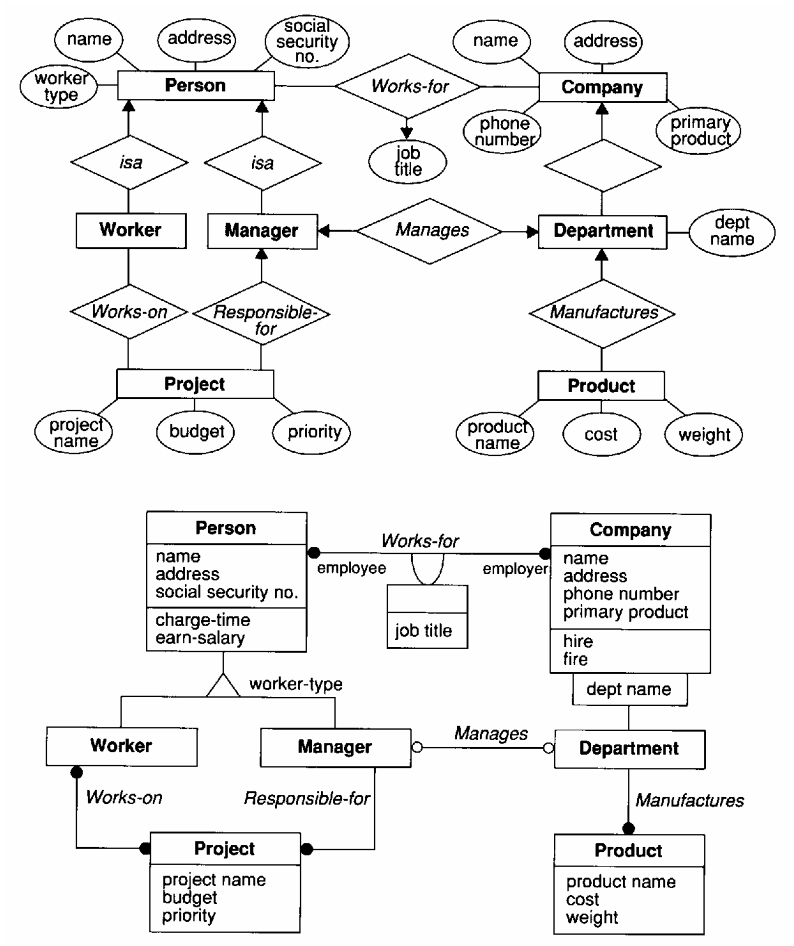


Figure 2.20. Entity-Relationship and OMT representation. From Rumbaugh (1991).

The Object Modeling Technique uses a convention which allows attributes to be shown textually, rather than as nodes. More importantly, different edge types communicate not only the cardinality of relationships, but also the inheritance characteristics. Note that besides graph conventions, adjoinment conventions are used to subdivide information in the nodes. Also, heads and tails of edges are treated as distinct type of nodes with their own labels.

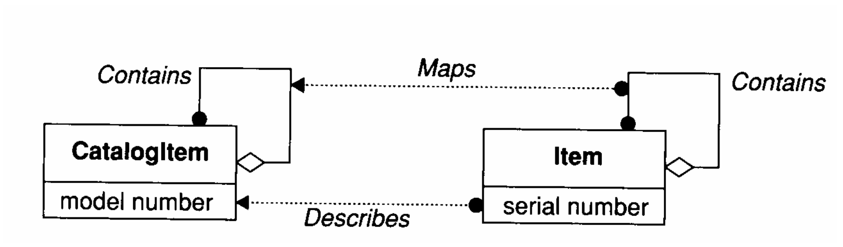
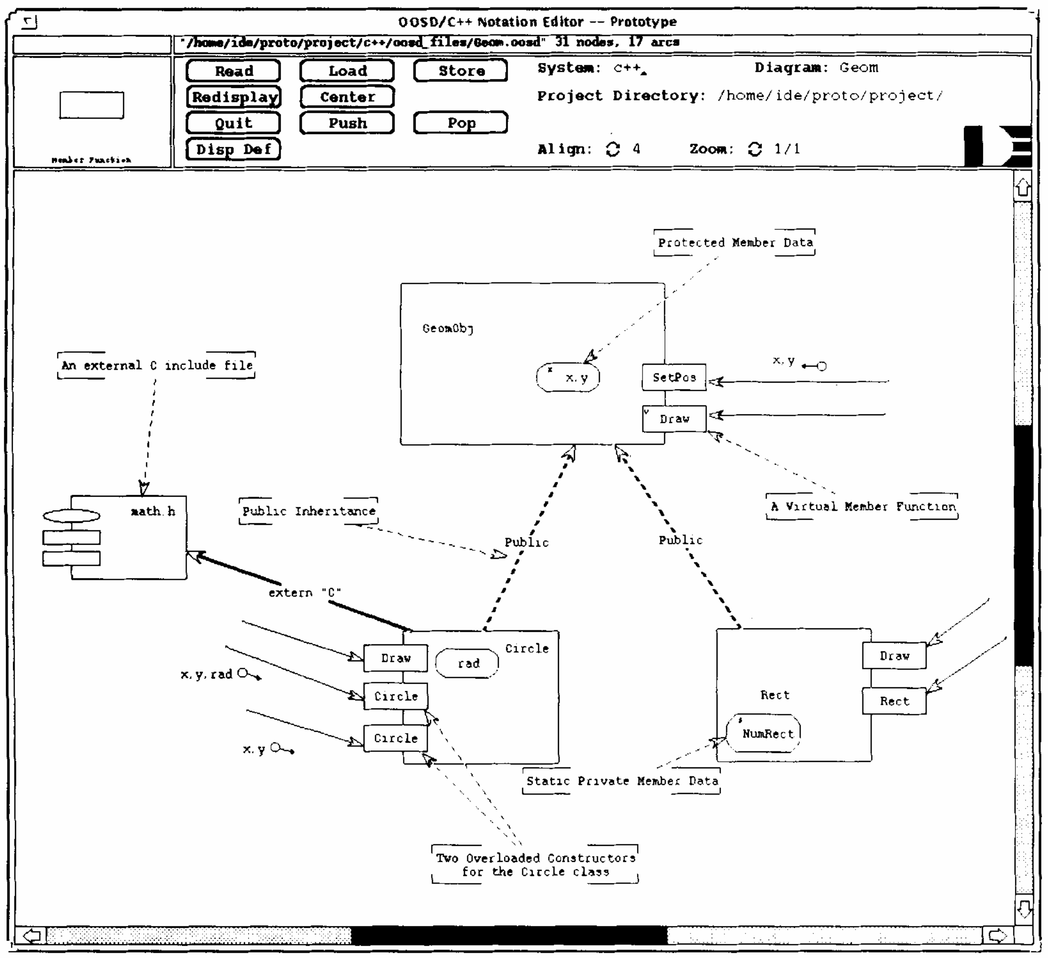


Figure 2.21. Homomorphism in OMT. From Rumbaugh (1991).

The Object Modeling Technique allows for meta-edges, as in the above figure, showing the relationship between two different models of data. The diagram crosses into the symbolic realm through our understanding that this is really two overlaid diagrams showing two different levels of association.


OOSD/C++ Notation Editor -- Prototype

"/home/ide/proto/project/c++/oosd_files/Geom.oosd" 31 nodes, 17 arcs

System: C++

Diagram: Geom

Project Directory: /home/ide/proto/project/

Align: 4

Zoom: 1/1

Figure 2.22. CASE Tool (Software Through Pictures) version of Booch notation. From Fisher (1991).

This diagram from a CASE tool shows a program represented in Booch notation. In contrast to many of the other diagrams shown, the comparatively low resolution of screens (75 dots per inch) to paper (300 - 1200 dots per inch) suggests some of the limits of the amount of information that can be shown in front of a programmer. Booch notation uses all three conventions: linkage, enclosure, and adjoinment.

2.2.13. Petri Nets

Petri nets are closely related to data flow graphs. The main distinction is that the graphs are bipartite, made up of a set of places and transitions:

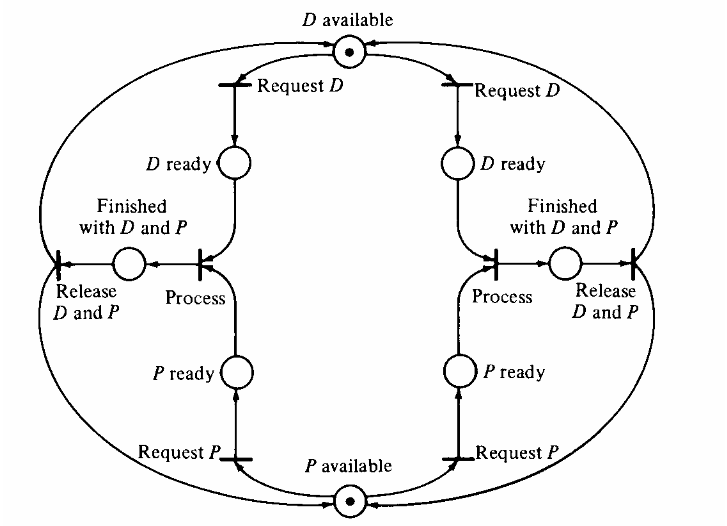


Figure 2.23. Petri net. From Johhsonbaugh (1984).

Each type of node can be further subdivided into subtypes.

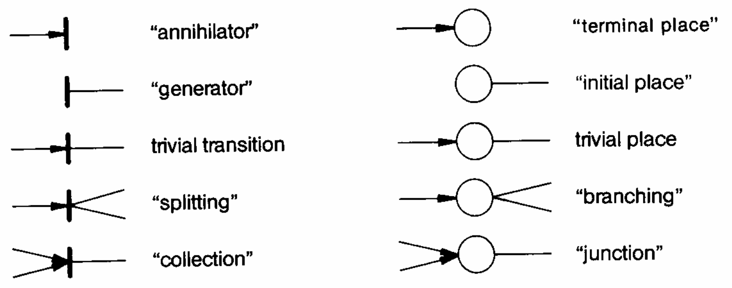


Figure 2.24. Petri Net node types. From Bauer(1982)

2.2.14. Data flow graphs

A data flow graph is a directed graph consisting of edges, which represent data flow, and nodes, which represent operations.

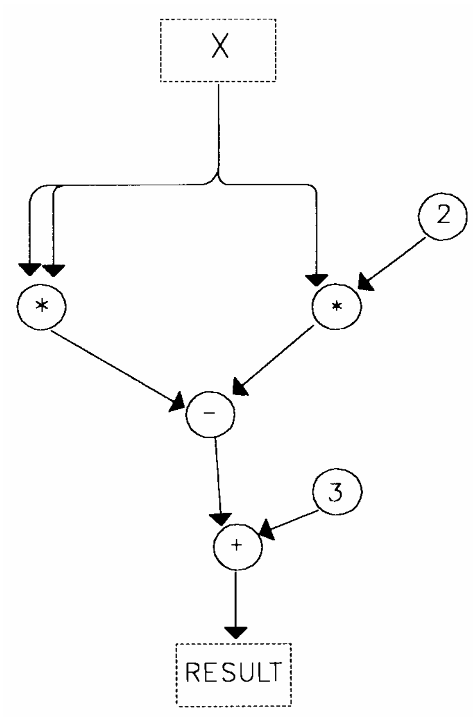


Figure 2.25 $\boxed{x^2-2x+3}$ from Shu (1988).

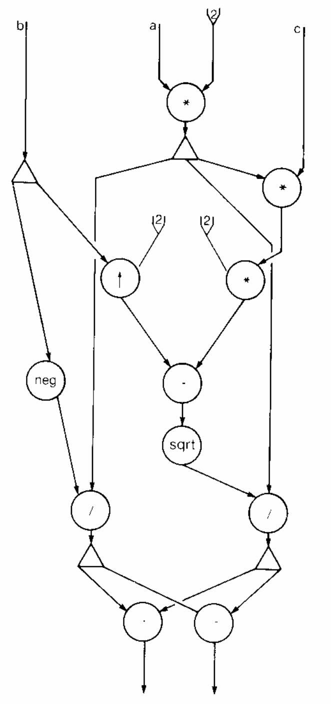


Figure 2.26 Roots of a quadratic equation. From Sharp (1985).

Tokens flow through the graph - when a node has tokens ready on all its incoming edges it will execute. When the node has executed, it puts tokens on its output edges. There is no predetermined sequence to the execution of a data flow graph - the data drives the order of execution.

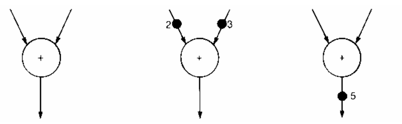


Figure 2.27. Illustration of token firing, from Bauer(1982)

On the graph itself, the nodes can be of 4 types (Perrot 1987) :

- computational (2 in, 1 out, or 1 in, 1 out),
- control (2 in, 2 out),
- merge (2 in, 1 out),
- dup (1 in, 2 out).

Dataflow graphs are often used in conjunction with dataflow machines, computers built to process tokens in parallel.

2.2.15. Dataflow diagrams

Dataflow diagrams are oriented to flow-type operations. Objects of data are shown in relationship to procedures. No decision logic is show; the diagram is most often used to model the flow of data.

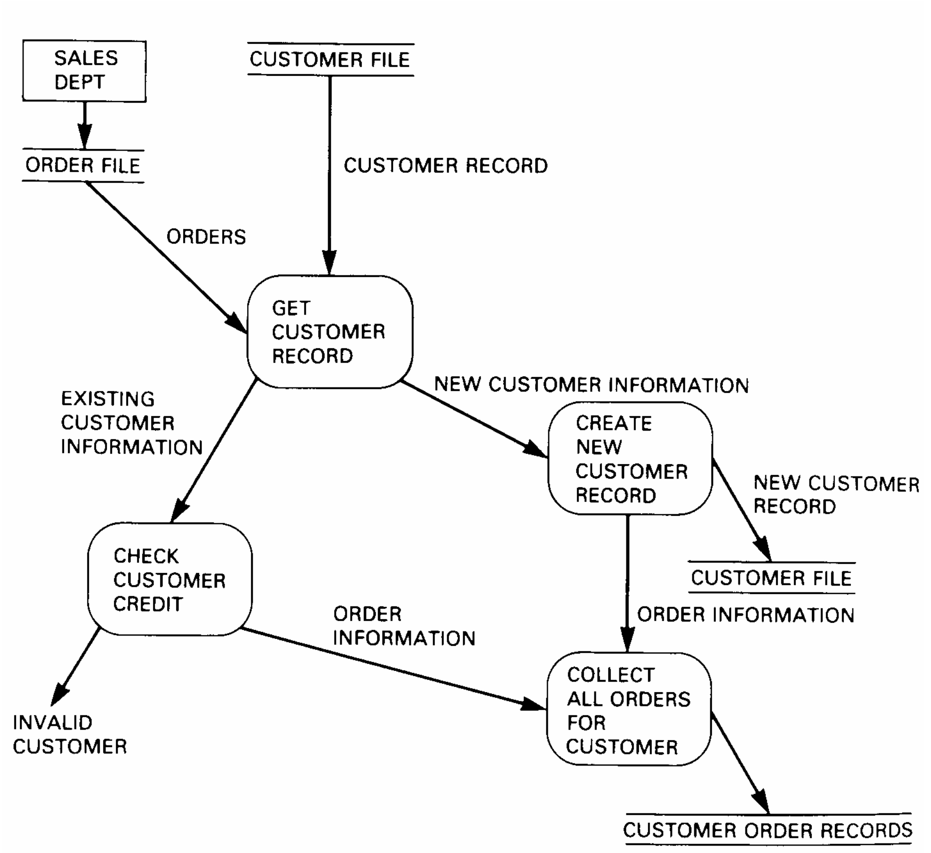


Figure 2.28. Simple data flow diagram. From Martin (1983).

A simple example is shown above, a more complex example is shown below.

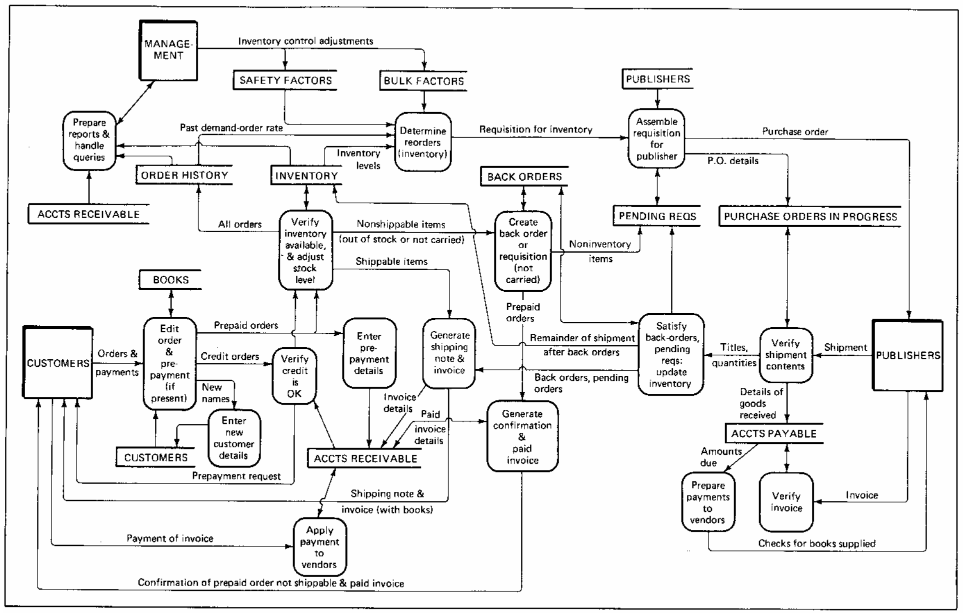

MANAGE-MENT
Inventory control adjustments
SAFETY FACTORS
BULK FACTORS
PUBLISHERS
Prepare reports & handle queries
Past demand-order rate
Determine reorders (inventory)
Requisition for inventory
Assemble requisition for publisher
Purchase order
Inventory levels
ORDER HISTORY
INVENTORY
BACK ORDERS
P.O. details
ACCTS RECEIVABLE
All orders
Verify inventory available, & adjust stock level
Nonshippable items (out of stock or not carried)
Create back order or requisition (not carried)
PENDING REQS
PURCHASE ORDERS IN PROGRESS
Noninventory items
Shippable items
BOOKS
Prepaid orders
Satisfy back-orders, pending reqs: update inventory
Edit order & pre-payment (if present)
Prepaid orders
Enter pre-payment details
Generate shipping note & invoice
Remainder of shipment after back orders
Titles, quantities
Verify shipment contents
Shipment
CUSTOMERS
Orders & payments
Credit orders
Verify credit is OK
PUBLISHERS
New names
Back orders, pending orders
Details of goods received
Invoice details
Paid invoice details
Generate confirmation & paid invoice
Enter new customer details
ACCTS PAYABLE
CUSTOMERS
ACCTS RECEIVABLE
Amounts due
Prepayment request
Prepare payments to vendors
Verify invoice
Invoice
Shipping note & invoice (with books)
Apply payment to vendors
Payment of invoice
Checks for books supplied
Confirmation of prepaid order not shippable & paid invoice

Figure 2.29. Complex data flow diagram. From Gane (1979).

As these diagrams are often complex, an H-graph convention (see section 2.3.1) is often used in which any particular node can expand into subnodes:

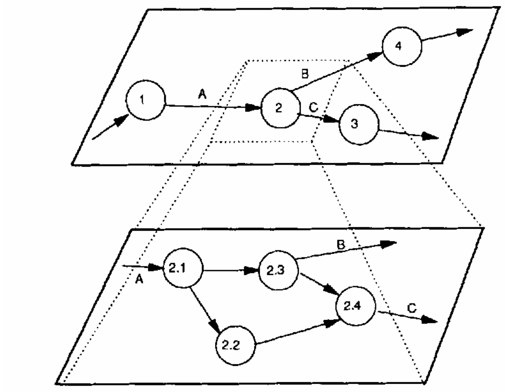


Figure 2.30. Nested dataflow diagrams. From Fisher (1991).

2.2.16. Signal Processing Graphs

The conventions of signal processing are often used to describe streams of computations:

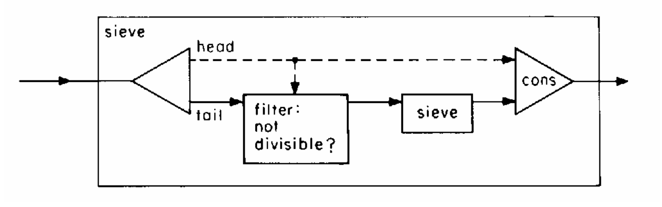


Figure 2.31. Sieve example. From Abelson (1985).

In this example from Abelson, a dotted line indicates a singular element, while a solid line indicates a stream. Both link and containment conventions are used. Also, note the use of symbolic recursion; sieve calls itself.

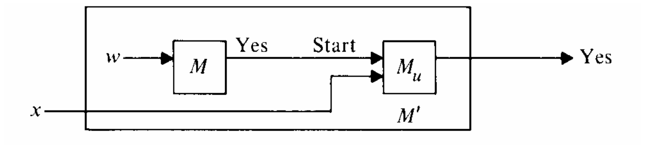


Figure 2.32. Machine notation from Hopcroft (1979).

In this example from Hopcroft, the signal processing convention is used to communicate strategies for combining abstract machines. Again, both containment and link conventions are used.

2.3. VISUAL LANGUAGE FORMALISMS

Diagrams such as flowcharts can represent programs. Programming languages can be represented by context free grammars. But it is not obvious if diagrams can be generated from formal languages. This sections outlines the formal methods used to describe diagrams.

2.3.1. H-graphs

H-graphs are graphs whose nodes can be other graphs. This construction implicitly underlies most visual programming languages. H-graphs are discussed in Pratt (1971b, 1973).

The components of an H-graph are atoms, nodes, and graphs. We use A and N to signify the universe of possible atoms and nodes.

> An extended directed graph (or graph) G over N and A is a triple
> G=(M, E, s) where
> Nodeset $M \subseteq N$,
> Edgeset E: $M \times A \rightarrow M$,
> Initial Node s, $\in M$.
> E is partial, finite, and M is finite, nonempty.
> E(m, b) = n means there is an edge from node *m* to node *n* labeled *b*.
>
> An H-graph is a pair H = (M, V) where Nodeset $M \subseteq N$, and is finite, nonempty.
> Value function V: $M \rightarrow A \cup \{G \mid G \text{ is a graph over } M \text{ and } A\}$

In other words, the value of a node is an atom or another graph. This allows for a hierarchy of graphs.

Pratt also defines sub and rooted H-graphs, and defines a selector and arc traversal function. Rooted H-graphs are used to model arrays, records, sets, simple variables, lists, etc. Nodes represent storage locations, and the root is the point of access to the whole structure. The value function is the accessing mechanism. Atoms represent primitive values such as numbers and characters.

Pratt (1971a) outlines a method for converting programs back and forth into flowcharts:

```
procedure Absmax (a, n, m, y, i, k);
            array a; integer n, m, i, k; real y;
            begin integer p, q;
               y :=  0;
            for p :=  1 step 1 until n do
                   for q :=  1 step 1 until m do
                     if abs(a[p, q]) > y then
                               begin y := abs(a[p, q]); i := p; k := q end
                     else y :=  y
end
```

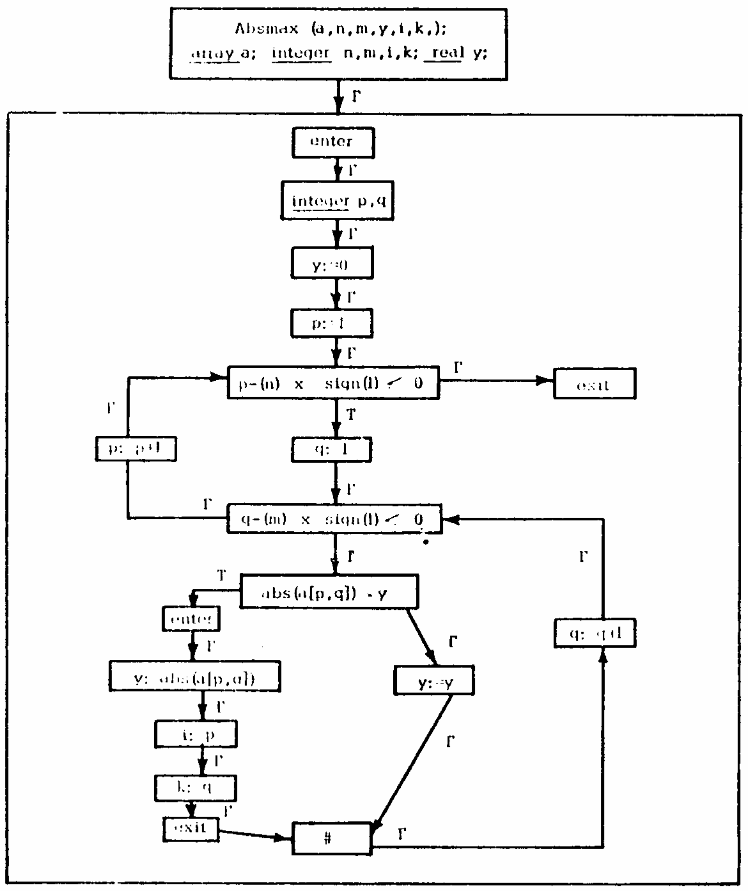


Figure 2.33. Flow chart and language equivalence. From Pratt (1971a).

He points out that compilers and programming language semantics both rely on building a representation more structured than strings. He notes the similarity between H-graphs and Web Grammars (Pfaltz 1969).

Pratt accomplishes the conversion between programs and flowcharts by manipulating graph languages. A graph language is a set of directed graphs with labeled nodes and arcs. A graph grammar generates a language of terminal graphs from a single non-terminal node. Each rewriting rule specifies a way of rewriting a node with a nonterminal value. A node can be replaced by a graph. This is done by replacing the incoming edge with an edge leading to the replacing graph, and replacing the outgoing edge of the node with the outgoing edge of the graph. This restricts the grammar rule to having a single input and output node. Stotts (1988) has extended the H-graph model to a visual parallel programming language.

2.3.2. Picture Layout grammars

Golin (1989) points out that programs are usually developed free-form in text editors. He recommends that visual programs should be developed free form in graphics editors. In order to do this, we need visual language grammars and compilers. He calls the grammars *picture layout grammars*. Golin bases his structure on a grammar model of his own invention he calls an *attributed multiset grammar*. The right side of a multiset grammar is seen as being an unordered collection rather than a sequence. Each grammar symbol has associated with it a rectangular extent or two endpoints. Production operators include

> *over, left_of, tiling, contains, group_of, adjacent_to, touches,*
> *points_to, labels, follow, join, fork, parallel.*

A parser, given a picture created with a graphic editor, can recover the underlying structure of the picture using the picture layout grammar.

2.3.3. Generating Trees and Graphs

Syntactic pattern recognition is a field of study that tries to recognize patterns in images by using formal language techniques. In most cases, these techniques are meant to be applied to data from scanned images, rather than to the already computer-generated images of visual programming. However, the work is often cited in the visual programming literature, and is useful in discussing the difference between textual and visual representation.

Context-free grammars can be used to generate images if some terminals are assumed to be line segments, and operators are assumed to be ways of joining the line segments. Then a string such as ((a + b) * c) can define three segments joined in a certain way, say into a triangle. This is the technique used in Shaw's picture description language (1969).

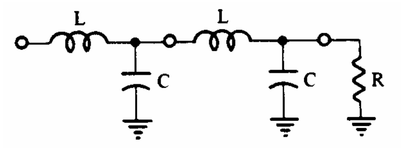


Figure 2.34. Circuit diagram from a grammar. In Gonzalez (1978).

The above figure can be generated from the string *aab*, if an *a* generates a capacitor and a *b* generates a resistor.

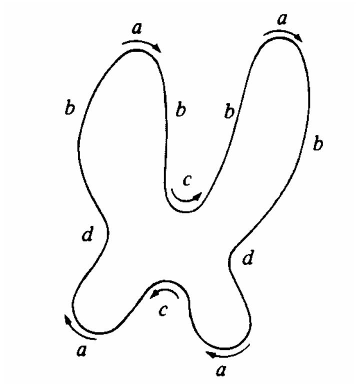
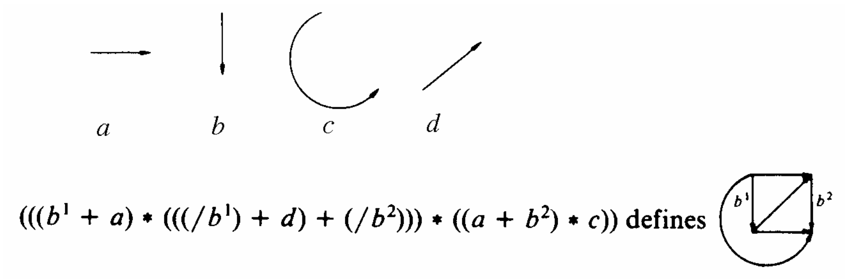


Figure 2.35. Chromosome from a grammar. In Gonzalez (1978).

A string can be used to generate curvilinear figures, as in the above chromosome, from the string *babcbabdacad.* But the generation of a real graph is much more difficult with this method:



$(((b^1 + a) * (((/b^1) + d) + (/b^2))) * ((a + b^2) * c))$ defines

Figure 2.36. Complete 4 node graph. In Gonzalez (1978).

This provokes the thought that textual representation works well with trees, but badly with graphs. We will return to this question in chapter 7.

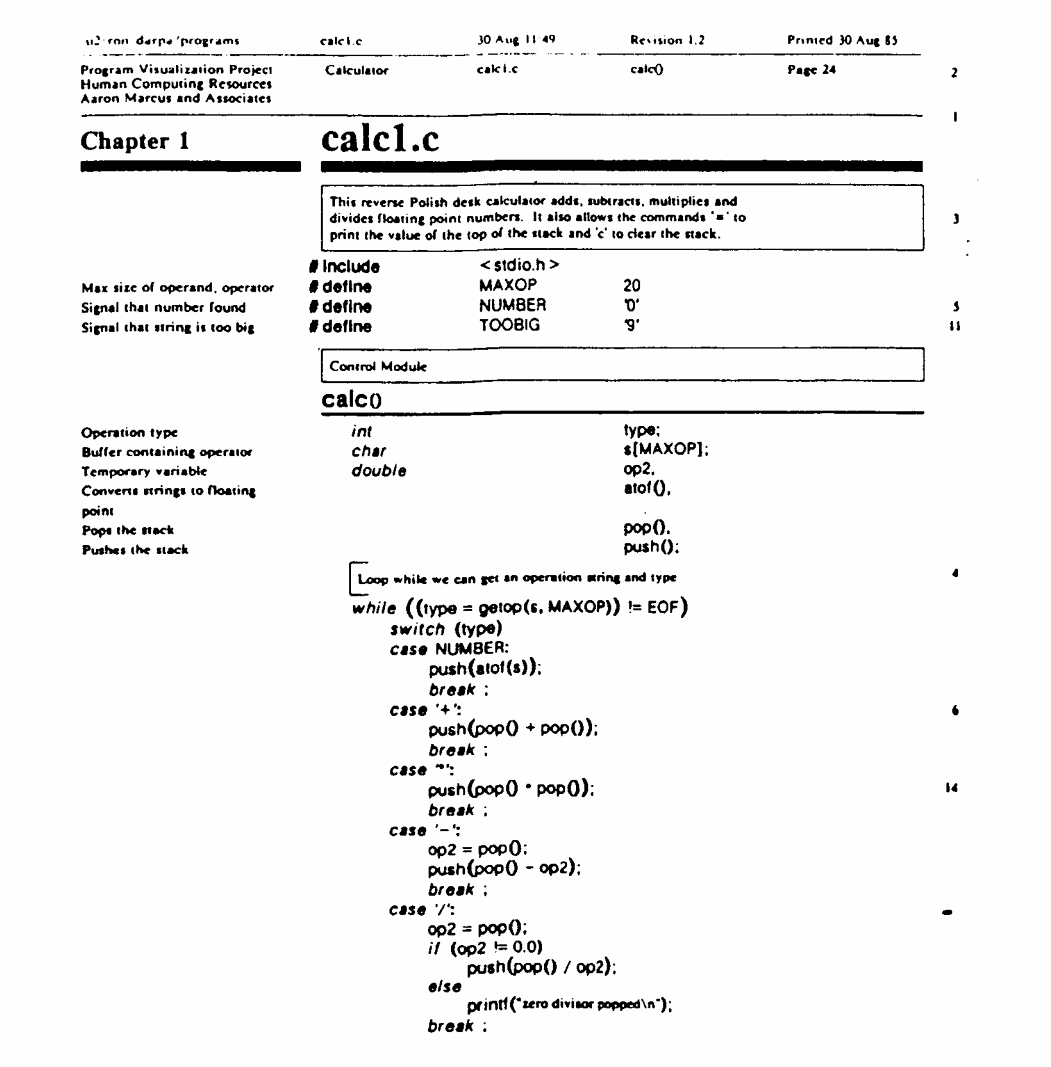
All of the above grammars are described in Gonzalez (1978); web grammars were first defined in Pfaltz (1970). The work of Fu and his students has dealt with many different ways of representing images, including the grammars described above (Fu 1984).

2.4. Program Visualization Systems

Programs are visualized by portraying their algorithms or data structures. Whereas the software diagramming techniques discussed above are often performed in the initial analysis and design stages of a project, program visualization is intended for use once an application is already built.

2.4.1. Readable Source Code

One typographic convention used in writing code is the indentation of nested code blocks. Programmers tend to indent differently. Utilities have been created that produce a standard indentation model, clarifying or at least standardizing the structure of the source code. Baecker and Marcus(1986) takes this a step further with a system called SEE, which typesets the source:



u2:ron darpa/programs	calc1.c	30 Aug 11:49	Revision 1.2	Printed 30 Aug 85
Program Visualization Project Human Computing Resources Aaron Marcus and Associates	Calculator	calc1.c	calc()	Page 24

2

1

Chapter 1 **calc1.c**

This reverse Polish desk calculator adds, subtracts, multiplies and divides floating point numbers. It also allows the commands '=' to print the value of the top of the stack and 'c' to clear the stack. 3

```
                                   # include   < stdio.h >
Max size of operand, operator      # define    MAXOP       20
Signal that number found           # define    NUMBER      '0'          5
Signal that string is too big      # define    TOOBIG      '9'          11
```

Control Module

calc()

```
Operation type                        int               type;
Buffer containing operator            char              s[MAXOP];
Temporary variable                    double            op2,
Converts strings to floating                            atof(),
point
Pops the stack                                          pop(),
Pushes the stack                                        push();

       Loop while we can get an operation string and type              4
       while ((type = getop(s, MAXOP)) != EOF)
            switch (type)
            case NUMBER:
                 push(atof(s));
                 break ;
            case '+':                                                   6
                 push(pop() + pop());
                 break ;
            case '*':
                 push(pop() * pop());                                   14
                 break ;
            case '-':
                 op2 = pop();
                 push(pop() - op2);
                 break ;
            case '/':
                 op2 = pop();
                 if (op2 != 0.0)
                      push(pop() / op2);
                 else
                      printf("zero divisor popped\n");
                 break ;
```

Figure 2.37. Typeset source code. From Baecker (1986).

The system uses typographic rules to separate portions of the code, and uses multiple columns to differentiate comments from program statements. It also makes use of graphic symbols to call attention to anomalies such as early returns from functions. We classify this system as textual verging on the diagrammatic.

2.4.2. TFPDRAW

TFPDRAW (Matsumura 1986) is more programmatic than a typesetting system, but is not quite a programming language by itself.

Figure 2.38. TFPDRAW. In Matsumura (1986).

It can be seen from the diagram that program text is still written. However, the text is highly structured, with sequential, repetitive, and branching represented with lines ending in rectangles, semi-circles, and triangles, respectively. Each module is required to be less than 50 statements, so all modules appear on a separate screen or page.

The modules themselves are displayed in a module-relation diagram that used the same format as the design diagram. Modules can be of three types - process, data, or package, and these types are indicated by shape. The limitation of 50 lines per module is logically artificial, but psychologically sensible, as it allows the programmer to get an immediate understanding of the module in one glance.

2.4.3. Incense

Incense (Myers 1983) is a system working on top of Mesa. It provides a way for a programmer to define how each data structure will be displayed on the screen. For example, the system displays box-and-pointer diagrams of pointer structures rather than their numeric addresses.

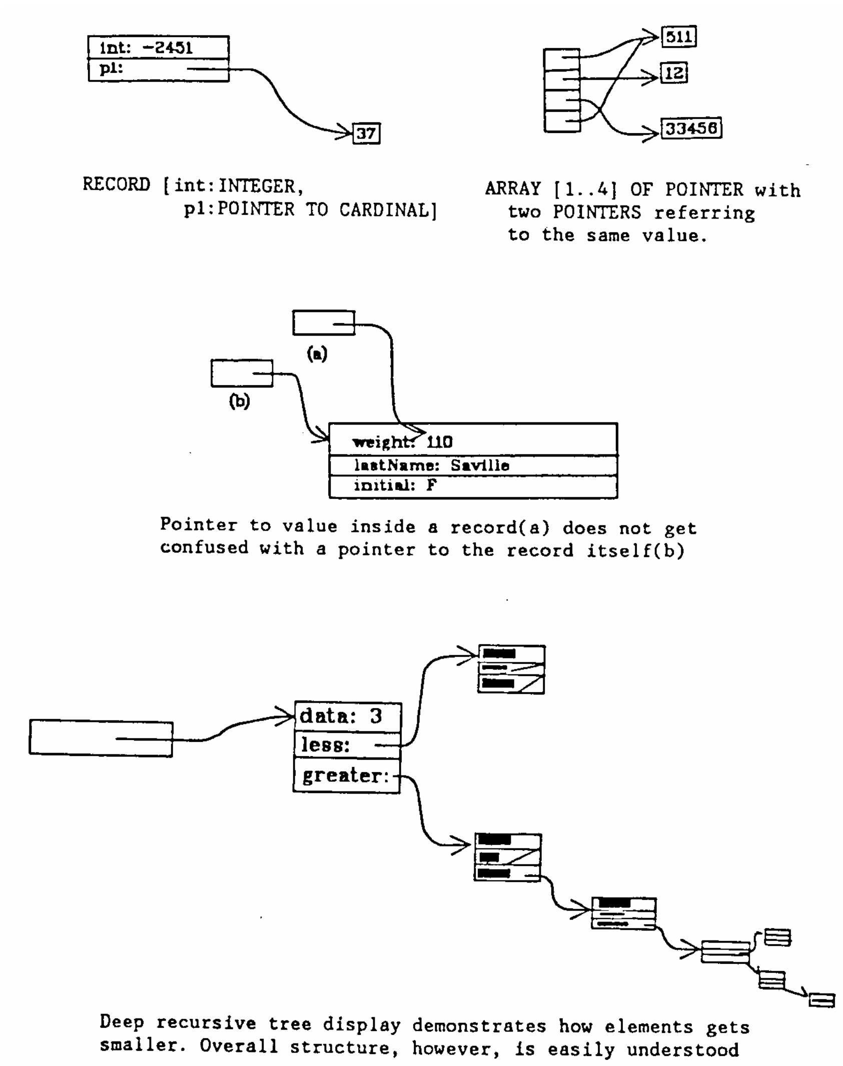
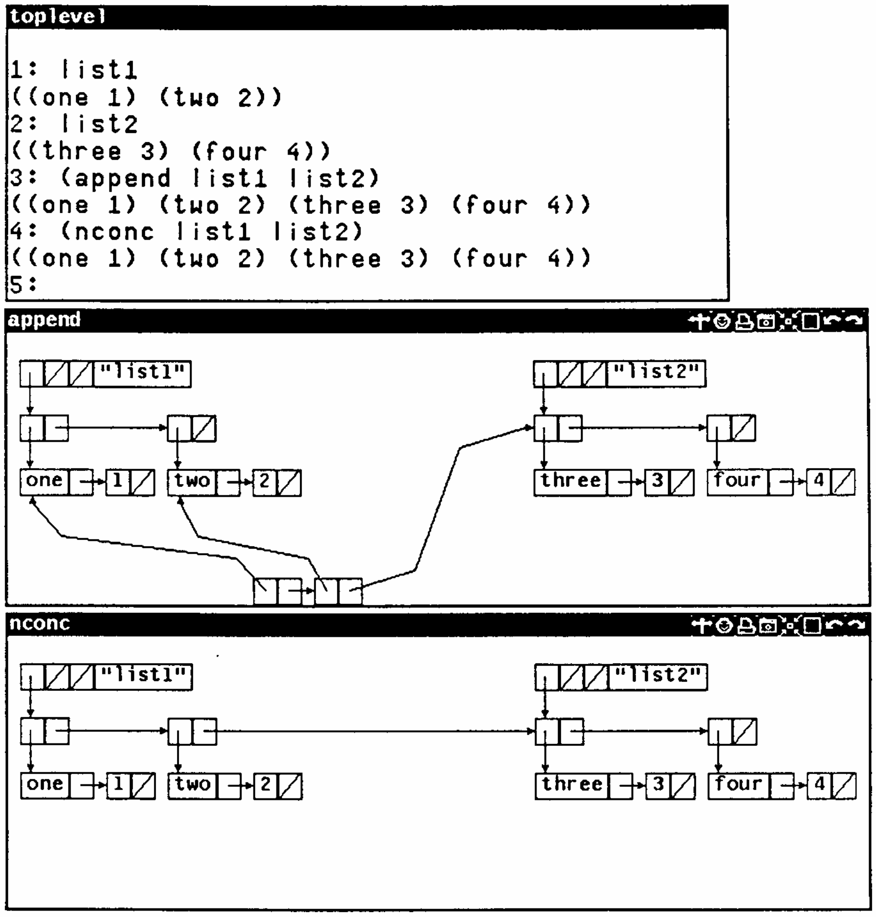

int: -2451
p1:
37
RECORD [int:INTEGER,
p1:POINTER TO CARDINAL]
511
12
33456
ARRAY [1..4] OF POINTER with
two POINTERS referring
to the same value.
(a)
(b)
weight: 110
lastName: Saville
initial: F
Pointer to value inside a record(a) does not get
confused with a pointer to the record itself(b)
data: 3
less:
greater:
Deep recursive tree display demonstrates how elements gets
smaller. Overall structure, however, is easily understood

Figure 2.39. Incense. From Meyers (1983).

Incense handles some of the problems of fitting arbitrary structures onto the screen, and proposes methods for editing of the structures. Arrays, matrices, and any variety of pointer structures are all displayable. Size is used as a sort of logical proximity indicator - the farther down the list, the smaller the elements get. This use of size is a metric rather than a topological convention.

2.4.4. Kaestle, FooScape

Kaestle (Boecker 1986) is in many ways similar to Incense. It is built on top of LISP instead of Mesa. Since the LISP cell structure is uniform, the system can automatically generate a visual representation for any data structure.



```
1: list1
((one 1) (two 2))
2: list2
((three 3) (four 4))
3: (append list1 list2)
((one 1) (two 2) (three 3) (four 4))
4: (nconc list1 list2)
((one 1) (two 2) (three 3) (four 4))
5:
```

Figure 2.40. Castle. From Boecker (1986).

Fooscape, also mentioned in Boecker 1986, operates at the function level. It is described as a landscape of functions:

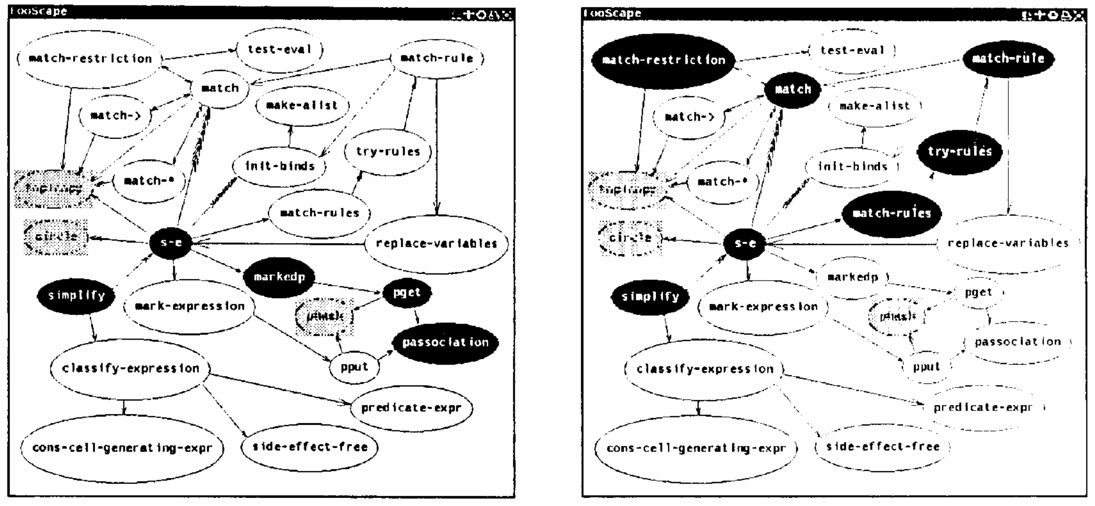


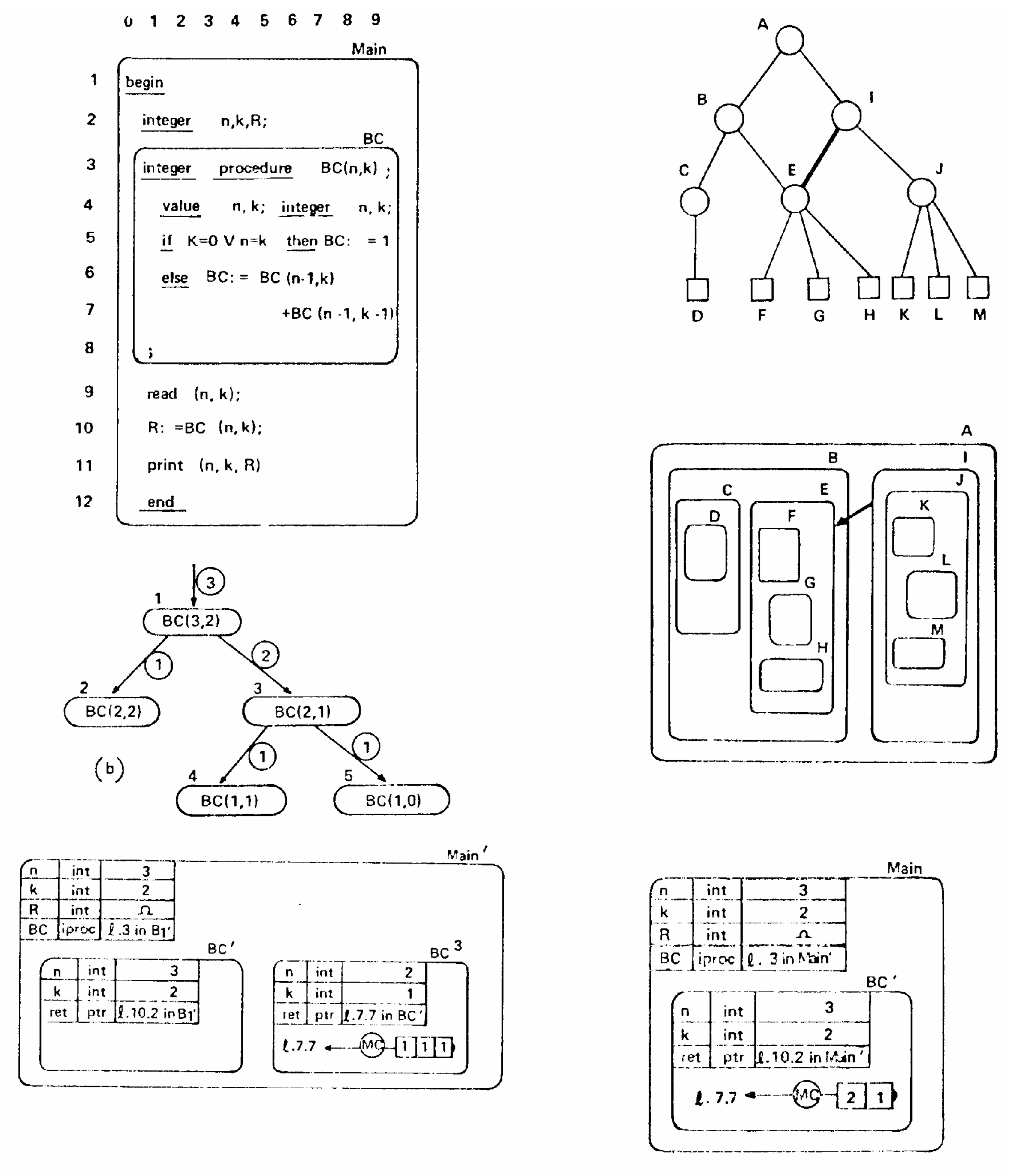
Figure 2.41. Fooscape. From Boecker (1986).

Its main purpose is debugging. A diagram of functions is displayed. As each function in a program is fired off, the oval containing the function changes color. Some of the functions can be marked off as not of interest (without this feature, a string-conversion function might flicker hundreds of times for each invocation of a higher-level function). It is also evident from the above diagrams that a limited number of functions can be represented on the screen at any time.

Boecker has added an audio dimension to the system by giving a unique entrance and exit tone to each function. He claims that abnormal behavior in a program both looks and sounds different from normal behavior.

2.4.5. Contour diagrams

Organik (1974) discusses the display of contour diagram snapshots of executing programs as a teaching tool. The programmer invokes a procedure call at any point in the program, and a diagram for the current state is generated.



```
     0 1 2 3 4 5 6 7 8 9
                        Main
 1   begin
 2     integer   n,k,R;
                        BC
 3     integer  procedure   BC(n,k) ;
 4       value   n, k;  integer   n, k;
 5       if  K=0 V n=k   then BC:  = 1
 6       else  BC: =  BC (n-1,k)
 7                      +BC (n -1, k -1)
 8     ;
 9     read  (n, k);
10     R: =BC  (n, k);
11     print  (n, k, R)
12     end
```

n	int	3
k	int	2
R	int	Ω
BC	iproc	ℓ.3 in B1'

n	int	3
k	int	2
ret	ptr	ℓ.10.2 in B1'

n	int	2
k	int	1
ret	ptr	ℓ.7.7 in BC'

n	int	3
k	int	2
R	int	Ω
BC	iproc	ℓ. 3 in Main'

n	int	3
k	int	2
ret	ptr	ℓ.10.2 in Main'

Figure 2.42. Contour diagrams for a code fragment. From Organick (1974).

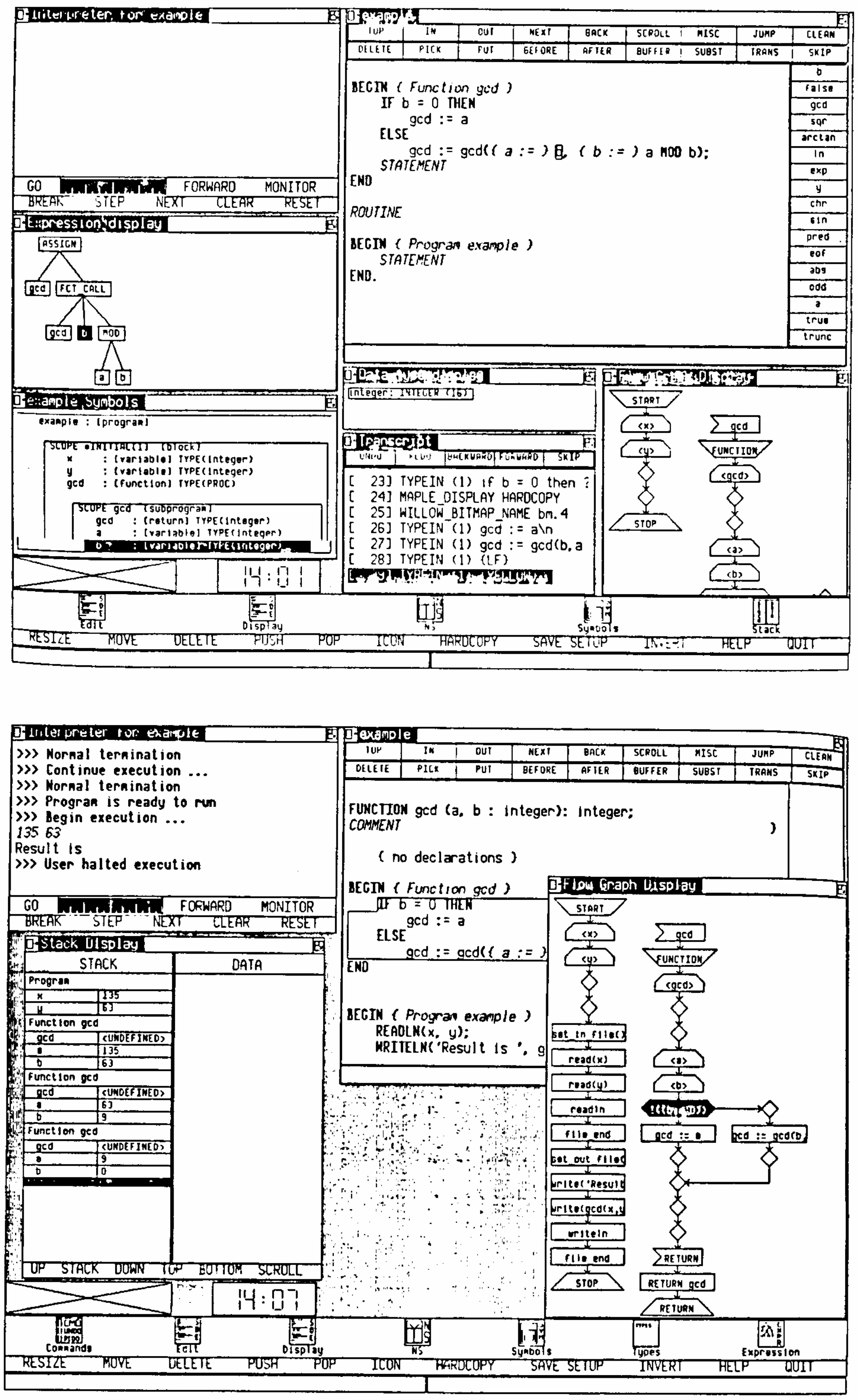
This system uses all three sets of conventions - tree structures are represented both as graphs and as containment diagrams, and memory is represented with adjoinment conventions.

2.4.6. PECAN

Pecan (Reiss 1985) is built on top of Pascal. The programmer sees multiple views of the program. These include

- source
- expression display as a tree
- flow graph display
- NS diagrams
- symbol table
- datatype definitions
- stack
- interpreter output for debugging

Both the flowgraph and the source highlight as the program is stepped through. The stack is displayed as the activation record of the particular part of the program that is executing.



```
BEGIN ( Function gcd )
    IF b = 0 THEN
        gcd := a
    ELSE
        gcd := gcd(( a := ) b, ( b := ) a MOD b);
    STATEMENT
END

ROUTINE

BEGIN ( Program example )
    STATEMENT
END.
```

```
example : [program]

SCOPE @INITIAL(1)  [block]
   x     : [variable] TYPE(integer)
   y     : [variable] TYPE(integer)
   gcd   : [function] TYPE(PROC)

  SCOPE gcd  [subprogram]
     gcd   : [return] TYPE(integer)
     a     : [variable] TYPE(integer)
     b ?   : [variable] TYPE(integer)
```

```
[  23] TYPEIN (1) if b = 0 then ?
[  24] MAPLE_DISPLAY HARDCOPY
[  25] WILLOW_BITMAP_NAME bm.4
[  26] TYPEIN (1) gcd := a\n
[  27] TYPEIN (1) gcd := gcd(b,a
[  28] TYPEIN (1) (LF)
```

```
>>> Normal termination
>>> Continue execution ...
>>> Normal termination
>>> Program is ready to run
>>> Begin execution ...
135 63
Result is
>>> User halted execution
```

```
FUNCTION gcd (a, b : integer): integer;
COMMENT                                          )

    ( no declarations )

BEGIN ( Function gcd )
    IF b = 0 THEN
        gcd := a
    ELSE
        gcd := gcd(( a := )
END

BEGIN ( Program example )
    READLN(x, y);
    WRITELN('Result is ', g
```

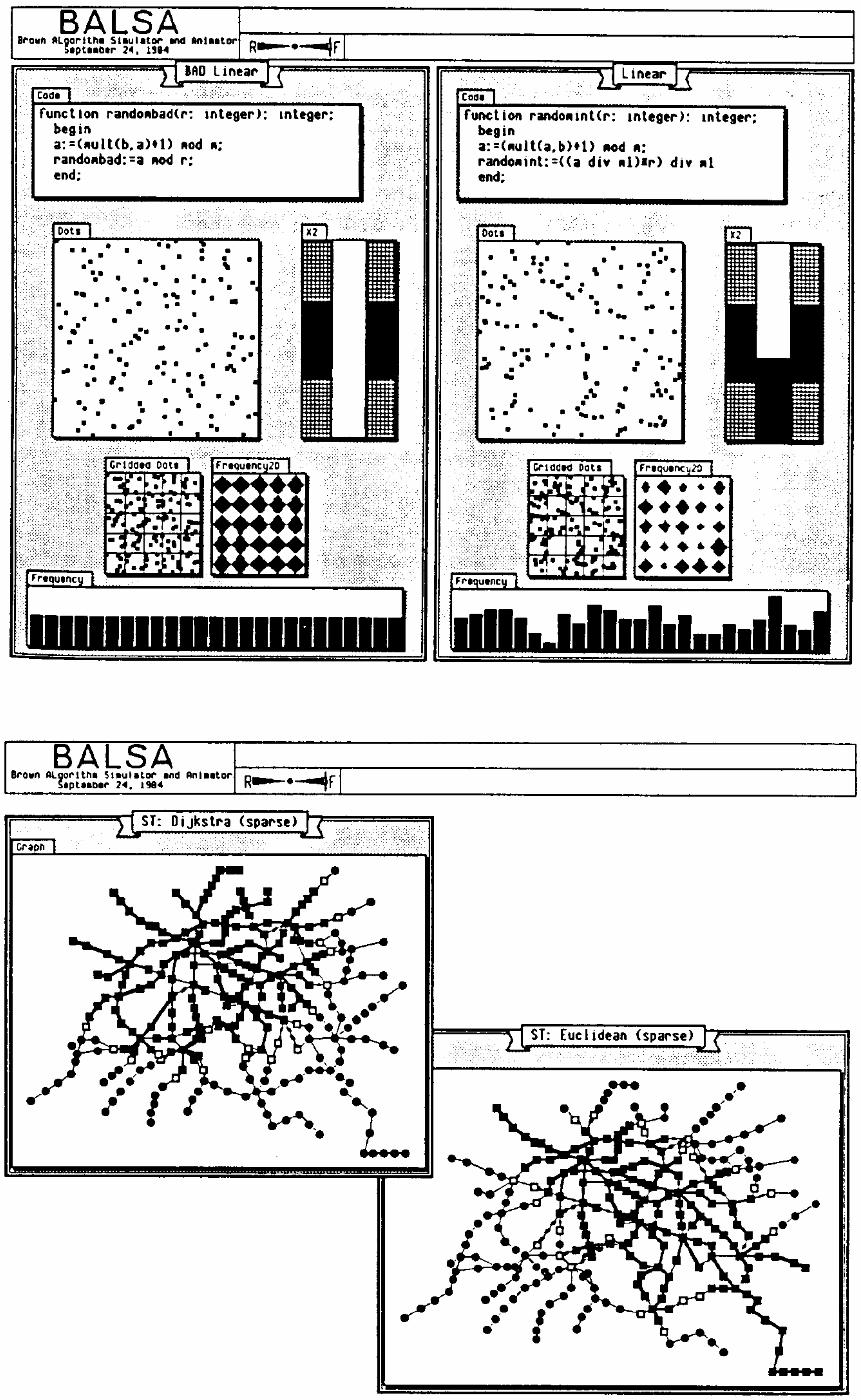
```
STACK                     DATA
Program
  x        135
  y        63
Function gcd
  gcd      <UNDEFINED>
  a        135
  b        63
Function gcd
  gcd      <UNDEFINED>
  a        63
  b        9
Function gcd
  gcd      <UNDEFINED>
  a        9
  b        0
```

Figure 2.43. Pecan. From Reiss (1985).

The contribution of this work is in the strong implementation of multiple views, where a program can be simultaneously seen in a number of different representations. The conventions used are text, graph-based, and containment based. In a certain sense, all windows-based system make use of the containment metaphor, as a window is a rectangle that is perceived as containing a viewpoint and state distinct from all other windows.

2.4.7. Balsa

The Balsa system (Brown 1985, 1987, 1988) was constructed as a teaching tool. It elaborately animates algorithms.



```
function randombad(r: integer): integer;
  begin
  a:=(mult(b,a)+1) mod m;
  randombad:=a mod r;
  end;
```

```
function randomint(r: integer): integer;
  begin
  a:=(mult(a,b)+1) mod m;
  randomint:=((a div m1)*r) div m1
  end;
```

Figure 2.44. Balsa. From Brown (1987).

An impressive piece of development, it uses both topological and metric conventions in representing program activity. Many of the outputs of the system are histogram-like in nature, intending to demonstrate the differences in effectiveness between alternate algorithms. Brown claims the system has helped computer scientists discover flaws in widely used algorithms.

In order to accomplish the animations, subroutine calls are injected into the program that control the animation. Brown refers to these injection points as *interesting events*. Freezes in the animation can be programmed in, so that the viewer is prompted in order to continue viewing the animation. The tool requires work from the programmer to decide what parts of the algorithm should be visualized and how the algorithm should be seen.

Perhaps the biggest contribution of the work is its demonstration of the power of animation in teaching how programs work:

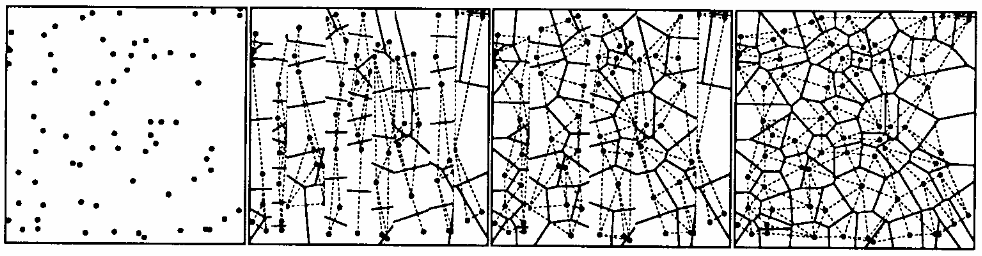


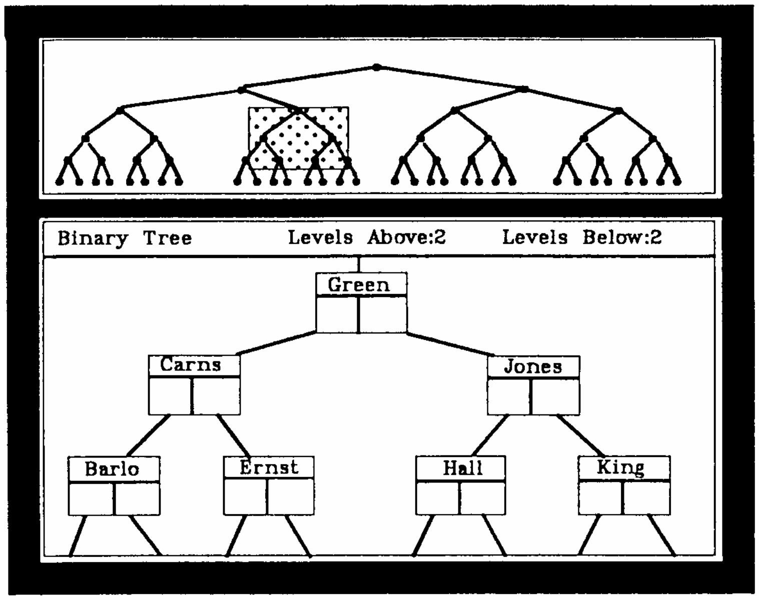
Figure 2.45. Balsa animation example of a Vornoi diagram algorithm. From Brown (1987).

2.5. Programming in the Large

The term *programming in the large* refers to the high-level design of subsystems and modules. Techniques and concerns of high-level design are different from the concerns of algorithm design, and it has been argued that high-level design should be done in a language distinct from that used to create data structures and algorithms. In most real-world environments, the high-level design is done ad hoc or on paper. The following systems not only provide computer-assisted ways of doing high-level design, but also provide formal output from the process.

2.5.1. PV: Software design visualization

G. P. Brown (1985) describes a system called PV built to visualize the software design process.



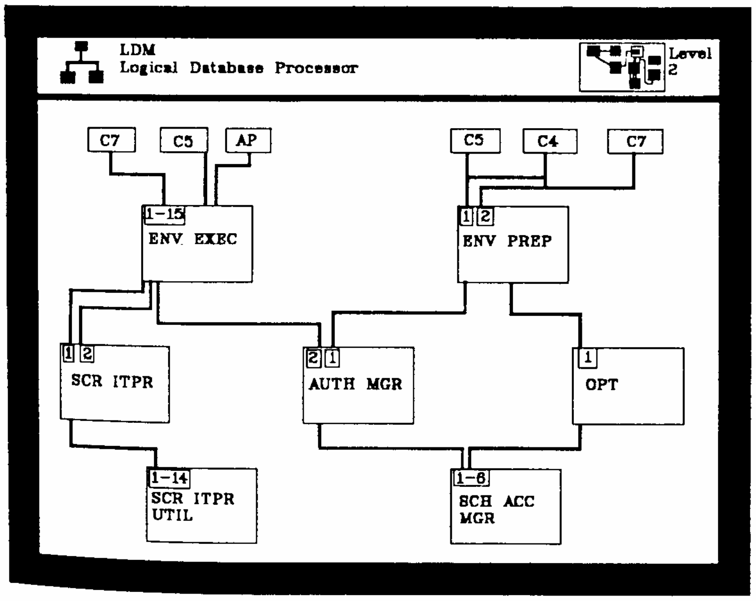


Figure 2.46. Software design visualization. From Brown(1985)

The system calls for linking together a whole range of diagram types:

- system requirements documents
- program function diagrams

- program structure diagrams
- communication protocol diagrams
- program text
- program commentary
- flow control
- structured data
- persistent data
- program in relationship to host environment

Graphic conventions are used to situate the programmer in the hierarchy of diagrams being edited. A smaller overall diagram is always shown; this diagram has a region highlighted that corresponds to the detail diagram being edited. The system seeks to handle everything from data flow diagrams to algorithm animation. The approach is more encompassing and less rigorous than the Pegasys system described below.

2.5.2. Pegasys

Pegasys (Moriconi 1985) links design diagram to underlying logic. The system uses a form calculus. As an example, when a network diagram is created, and an arrow is drawn between a shape labeled *host* and a shape labeled *line*, predicates such as the following would be generated:

- process(Host)
- module(Line)
- type(Packet)
- write(Host, Line, Packet)

Active entities in this system are:

- subprogram
- process
- module

Pictures are drawn through an interactive editor that enforces the form calculus. The drawing can be refined at different levels, meaning that a program is defined through a hierarchy of pictures.

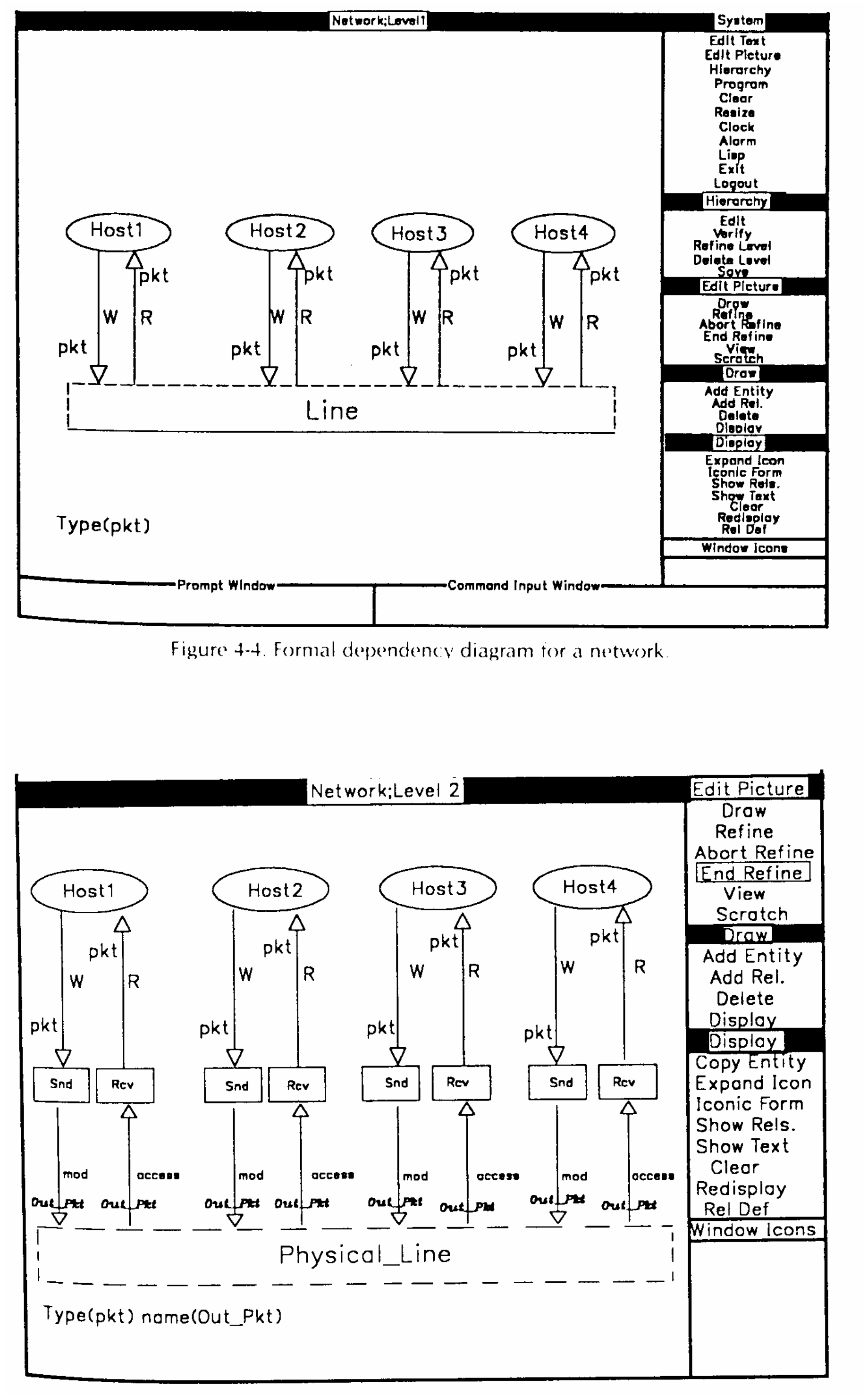


Figure 2.47. Pegasys network diagram. From Moriconi (1985)

Every atomic entity, that is, one that is not refined, must be associated with program units. Passive entities are associated with data objects. The system is built to be tied to Ada, so the active entities are associated with subprograms, packages, tasks and generics. The diagrams produced using the system correspond to a set of predicates, and since the editor enforces a set of rules the predicates are internally consistent. Pegasys can then, in an automated way, prove that the design hierarchy is consistent with the code it describes.

2.5.3. Goguen

Goguen (1985) suggests that semantics be used on large systems by attaching theories to program units. He suggests that this process is a visual, diagrammatic one, and should be made explicit.

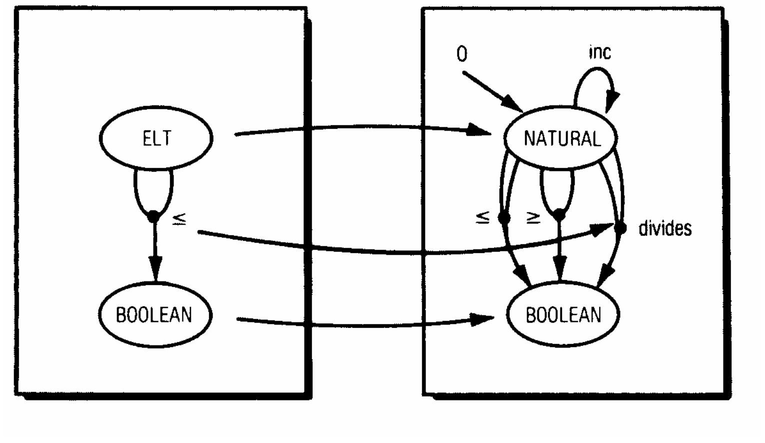


Figure 2.48. Mapping graphs with meta-edges. From Goguen (1985)

The diagrams he uses are of two sorts. Above, a graph with meta edges shows the mapping between graphs. Below, a puzzle-piece convention, an instance of the adjoinment convention, is used to illustrate how software modules can be fit together.

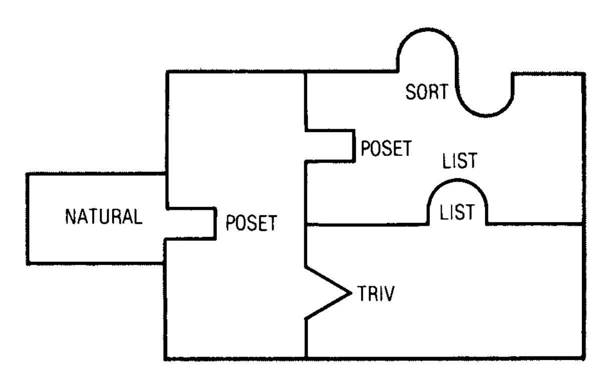


Figure 2.49. Puzzle piece convention. From Goguen (1985)

2.6. VISUAL PROGRAMMING LANGUAGES

We now consider visual languages that can create code. In looking at a broad set of languages, we find that some concepts are especially difficult to represent visually. Repetition, whether through iteration or recursion, can be hard to communicate. Parameter passing is also difficult; the textual substitution of a parameter does not lend itself to simple visual representation. The representations of repetition or parameters differ according to the underlying language model. We distinguish between imperative and functional languages; we look at one functional language, PROGRAPH, in depth.

2.6.1. Imperative Visual Languages

Many of the visualization tools we discussed in the previous sections are based around imperative languages. And all flowchart techniques are diagrammatic descriptions of imperative languages.

PICT

PICT is a system developed by Glinert in his Ph.D. dissertation (1985). He strives for a system in which no text will be necessary.

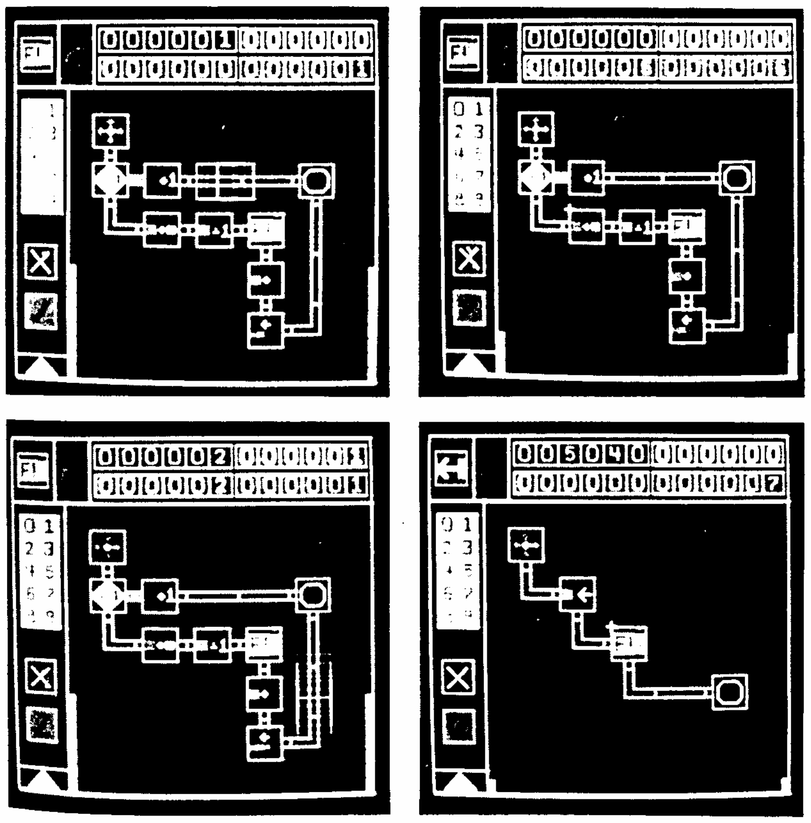


Figure 2.50. PICT factorial example. From Glinert (1985).

The prototype is very restricted; only four 6-digit nonnegative decimal integers are allowed in each module. This essentially solves the parameter-passing problem. Each of the variables is assigned a color. The whole language is defined as

<language primitive>	::=	<sys control | <declarative op> | <boolean op>
<sys control>	::=	'start(entry)' | 'stop/return'
<declarative op>	::=	+ | - | + 1 | - 1 | set to 1 | set to 0 | assign a copy | input from joystick
<boolean op>	::=	> | = | < | = 0 | = 1

Glinert writes:

> PICT/D user programs look like flowcharts; although some professionals have questioned the usefulness of these diagrams, we find much to recommend the flowchart when it *is* the program itself rather than merely an aid to documenting it (page 53).

Using flowchart-like symbols, *if* and *while* statements can be constructed.

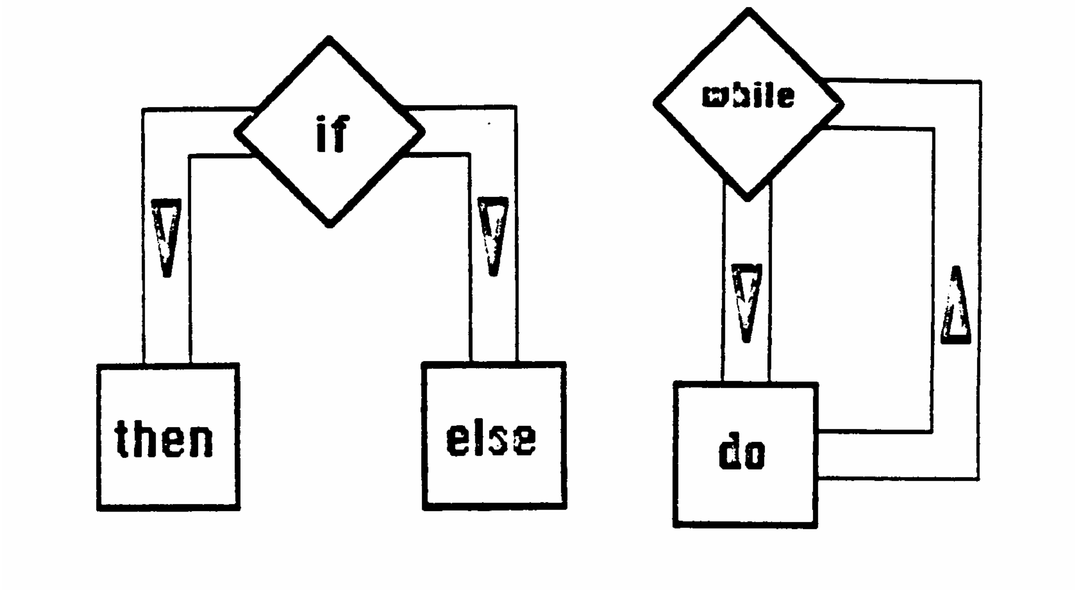


Figure 2.51. PICT conditionals and loops. From Glinert (1985).

The physical construction of the program is done with a joystick, so the loops are drawn by steering a line around the screen.

Blox

Blox (Glinert 1986) is a visual programming language made up of puzzle-like pieces that fit together.

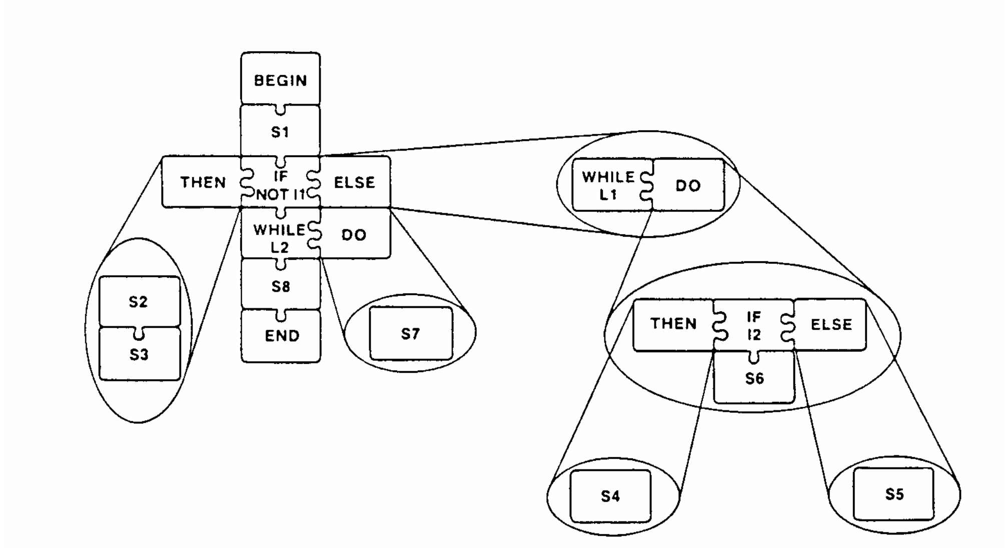


Figure 2.52. Blox. From Glinert (1986).

Protrusions are fit into sockets on the pieces, which consist of program language statements such as *while* and *if* statements. Each piece can hide lower level constructs that are encapsulated within. (This is the H-graph component of the language.) The individual statements such as assignments, are typed in at the keyboard. Notable in the language is the way *then* and *else* clauses move off the main vertical line of the control.

2.6.2. Functional Visual Languages

Functional languages seem by their nature to lend themselves to visual representation. Most often, they have simple, uniform data structures. They do not have parameter passing, and they don't allow multiple assignment. This means that the data flow graphs previously described can be used for creating low level expressions. At the higher level, functional languages provide a set of functional forms, which allow for the combining of already defined functions. The function becomes an object that can be manipulated to form a higher-level function.

Once a value is determined in a functional system, that value flows through the rest of the system unchanged. This is called the principle of single assignment. In practice, this is very restrictive, and many languages relax this restriction in order to handle iteration in a nonrecursive manner. In the textual data flow language Val, one can increment a designated loop variable. In textual Sisal, one also has mechanism for incrementing a loop variable. We see these ideas carry over into visual languages discussed below.

PROGRAPH

PROGRAPH (Matwin 1985) is a data flow-like language. Matwin writes:

> ... graphical formulation of programs in PROGRAPH is based on a premise that non-linear concepts can be better expressed in a two-dimensional pictorial form rather than in sequential verbal script.

Basics

Here is a simple PROGRAPH program:

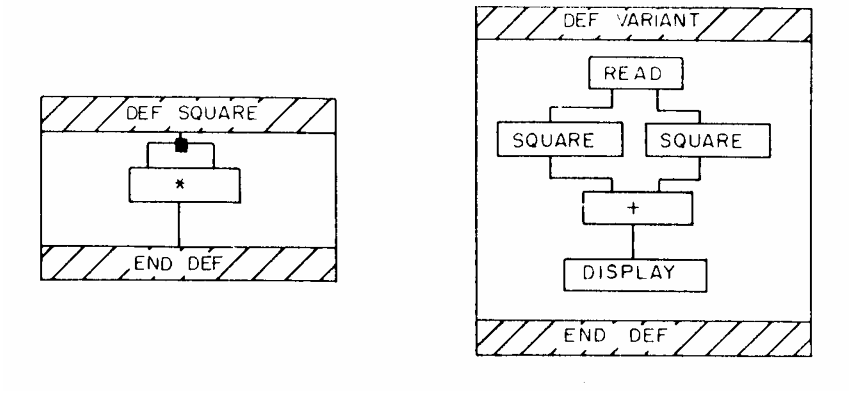


Figure 2.53. PROGRAPH square program. From Matwin (1985).

The line coming into the top of the box DEF SQUARE is the function's input. The line coming out of the box is the result. The branching of the input line feeds two identical values into the binary multiply function. SQUARE is used in the procedure VARIANT, which reads two numbers in, computes their squares, and adds the results together for display. Subroutine calls assume that the calling routine has the same number of input and output wires as the called routine.

Conditionals

Here are two examples of conditionals.

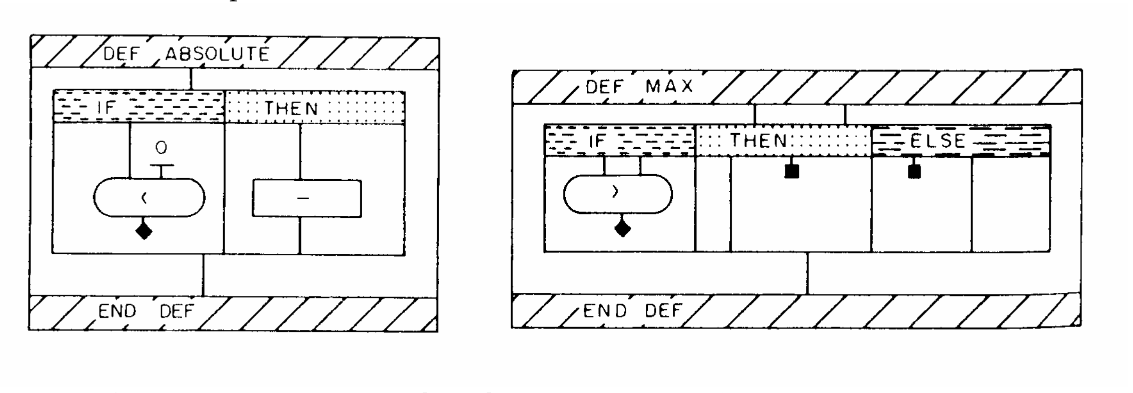


Figure 2.54. PROGRAPH conditionals. From Matwin (1985).

In all cases there is a boolean condition box. If the box tests positive, the THEN box is executed. In the case of the function ABSOLUTE, a number flows into the function; if the number is less than 0, the number goes through a unary minus function, converting the number into a positive number. This number flows out the bottom of the function. If the number is > 0, the IF condition fails, and the number that flowed into the box flows out unchanged.

In the function MAX, we see the use of another PROGRAPH wiring convention. An input can be grounded by terminating it with a small, filled square. If the first input is greater than the second input, the first input flows through, the second is terminated. Otherwise, the first is terminated, and the second flows through. Note that even though PROGRAPH does not have named parameters, the order of the wires performs the same function as parameter labeling does in textual languages.

Recursion

Here is a common example from list processing, REVERSE.

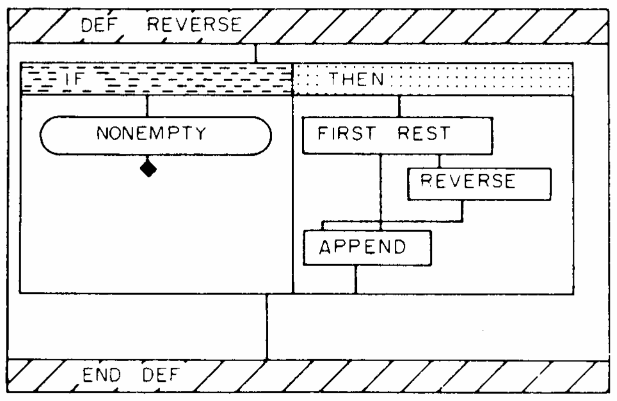


Figure 2.55. PROGRAPH reverse. From Matwin (1985).

FIRST and REST correspond to LISP CAR and CDR, but APPEND is slightly different - it concatenates an element, not a list, on to the back of an input list. The recursion is accomplished by labeling a box inside the function with the name REVERSE. The inside of this interior box could be drawn as a smaller copy of the surrounding box. In such a case we would get a common visual metaphor for recursion, a picture within a picture, regressing all the way down to the boundary condition. In textual languages, the function translates to:

```
reverse(x)
   if (nonempty(x) is TRUE)
        return(append(reverse(rest(x)), first(x)))
   else return(NIL)
```

Both the visual and the textual versions are fairly simple; however the visual version, through the crossing lines in the THEN box, draws attention to the basic idea of the algorithm.

Iteration

Unlike most visual functional languages, PROGRAPH supplies an iteration mechanism. A logical compartment sits on top of a transformation compartment. In a flow chart, the lines would be drawn explicitly. In PROGRAPH, the lines are implied. When the condition in the logical compartment is filled, data is passed to the transformation compartment. Results are then transferred back to the logical compartment. Even though lines are not drawn back up, the effect is very much like that of a flow chart, where a line is drawn explicitly back to the condition box.

Constant initialization is introduced in this figure:

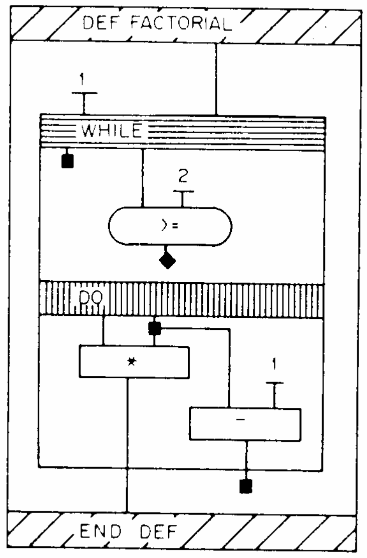


Figure 2.56. PROGRAPH factorial. From Matwin (1985).

The number 1 over the WHILE box is in effect only on the first test; from then on, the line has the value of the leftmost output of the DO loop. This line in effect accumulates the result. On the right side of the DO loop, the input number is decrement by one. This decremented number is output from the DO loop, then grounded. This means the number is fed back into the input line of the WHILE check, but stays internal to the

FACTORIAL box.

Psuedocode for this function reads

```
factorial (x)
   result = 1
   while (x >= 2)
      result = result * x
      x = x - 1
   return(result)
```

Again, both the textual and the visual functions are fairly simple. The visual function needs no temporary variables. However, the introduction of an initialization of a line, and the implicit looping of the construct, make the representation difficult to decipher in a glance. A more complicated example is shown in this figure, a function for merging sorted lists.

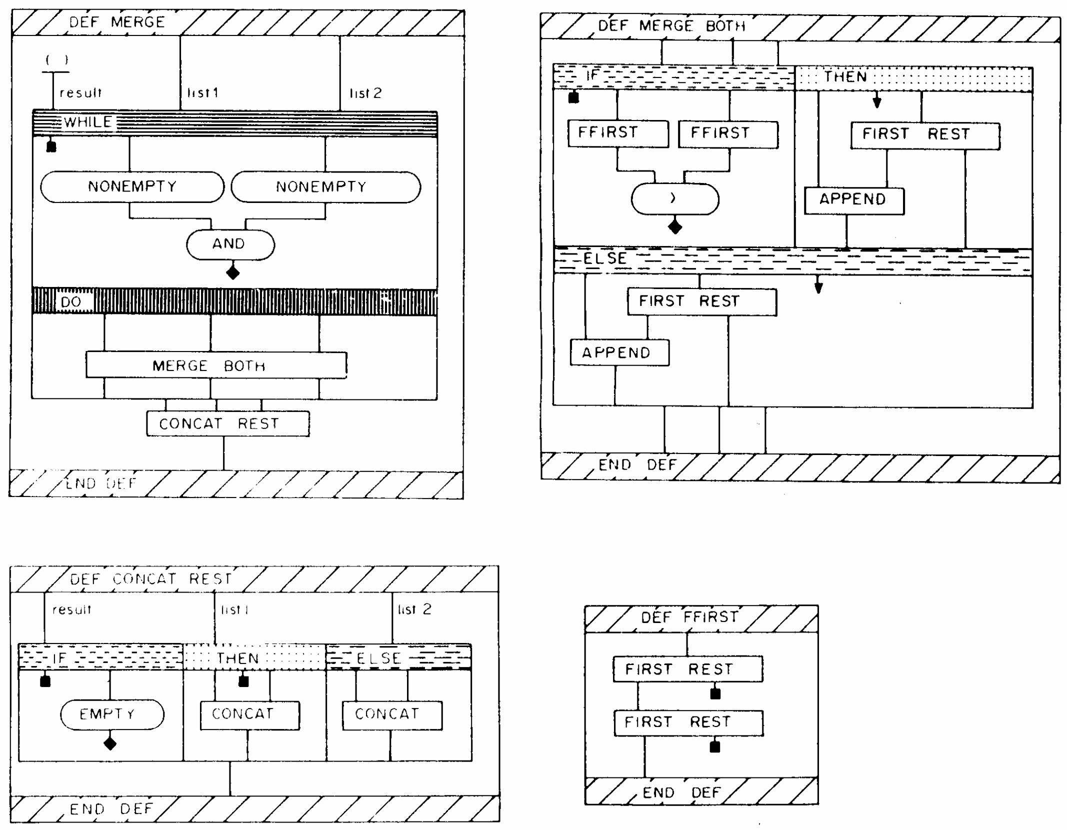


Figure 2.57. PROGRAPH merge. From Matwin (1985).

The function uses three sub-functions, relying at the bottom level on CONCAT, the equivalent of LISP APPEND. Another new convention is introduced; an arrow on an in wire indicates the wire extends to the bottom. Notice this convention is necessary as a result of two other conventions - first, the order the lines exit a function is significant, and second, lines touching boxes is significant. The THEN box, without the arrow convention, would have needed to snake the second line around the boxes, and then out the bottom of the function.

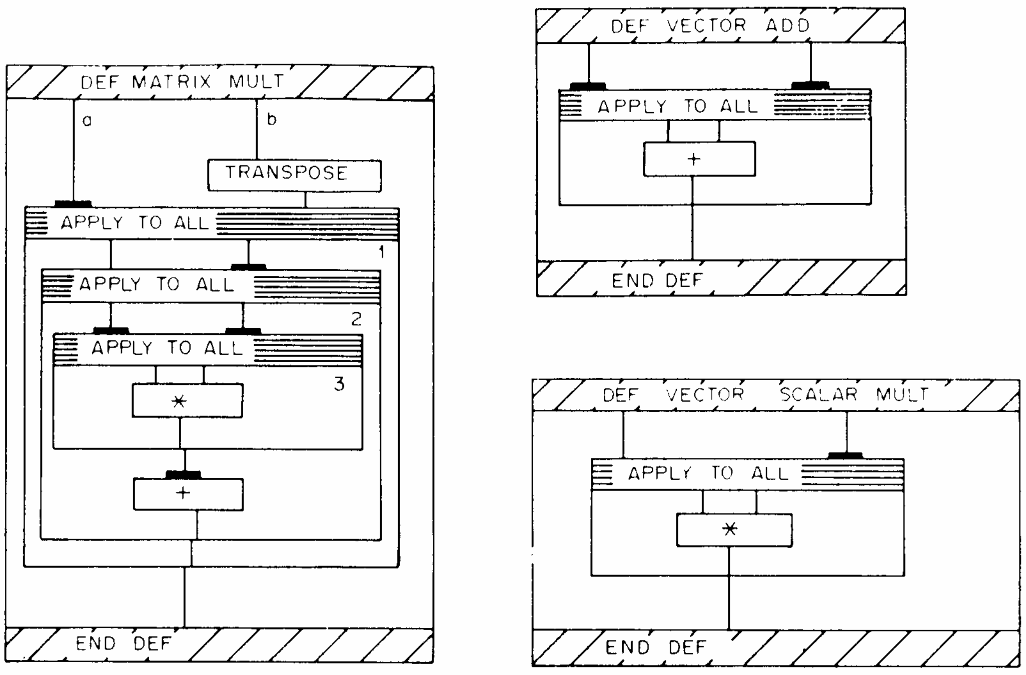
The following is a textual algorithm for merging, in the SCHEME dialect of LISP, from Abelson (1985). It is recursive, not iterative:

```
(define (merge s1 s2)
(cond (( empty-stream? s1) s2)
       ( empty-stream? s2) s1)
       else
       let ((h1 (head s1))
            (h2 (head s2)))
       cond ((< h1 h2) (cons-stream h1 ( merge (tail s1) s2)))
            ((> h1 h2) (cons-stream h2 ( merge s1 (tail s2))))
            (else
            (cons-stream h1
                         (merge (tail s1) (tail s2)))))))
```

The typography is visually aligned, making it is easy to see the minor difference in the way the (< h1 h2) and (>h1 h2) conditions are handled. As it is, the textual version seems easier to understand than the visual PROGRAPH version. The problem may be in the way the IF, THEN and ELSE clauses are stacked in the PROGRAPH version. If they were all drawn side-by-side, and the THEN clause was drawn the same size as the ELSE clause, the visual version would be somewhat clearer.

Parallel operations

PROGRAPH includes a convention for handling operations on lists in parallel. If an input line has a horizontal thick bar attached, then the input is considered multiple. This convention is used in conjunction with the APPLY TO ALL function.



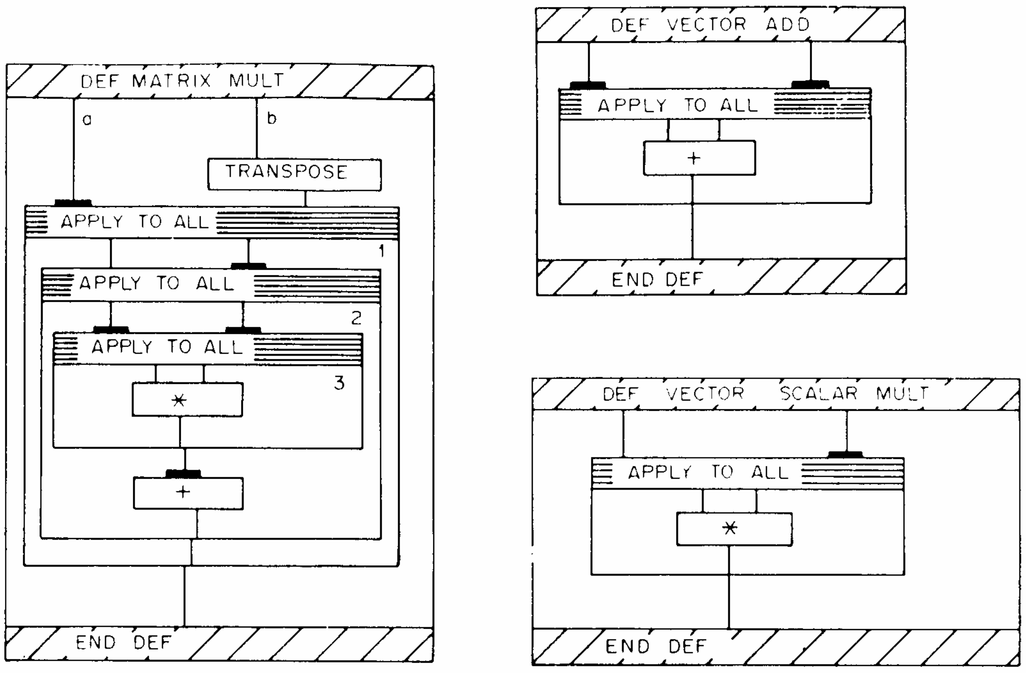


Figure 2.58. PROGRAPH matrix multiply. From Matwin (1985).

Two vectors are fed in as multiples to the APPLY TO ALL function. The corresponding elements of each vector are added together, and the resulting list output. In the right-hand diagram, one list is fed in normally, the other as a multiple. Each sublist of the multiple input will be multiplied by the single list fed in on the left input line.

Above, matrix multiplication is defined. At the bottom of the second box, a multiple is fed into a PLUS operation. When single multiples are fed into PLUS, TIMES, AND, OR, the result is a reduction. So in this example, the numbers in the list are added together, producing a single number.

Matrices in PROGRAPH as represented as a list of lists. In following the logic of the matrix multiplication, attention must be paid to which *in* wires are multiples. Initially, the left matrix is treated as a multiple; this means that a row at a time is processed against the entire transposed right matrix. For every one of the left rows, we treat the right matrix as a multiple. This means that we multiply row 1 by column 1, then row 1 by column 2, etc. The third box performs this multiply, and the bottom of the second box performs the addition. By the sequence of multiples, we end up constructing an output of lists of lists in the correct order.

This does not seem to be an extremely intuitive way of writing a matrix-multiplication function. However, it does work without named parameters, subscripts, or temporary variables.

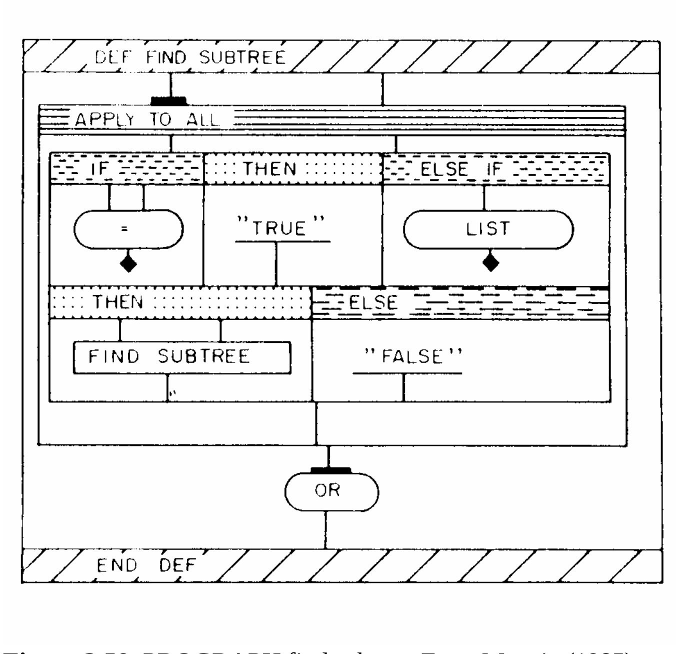


Figure 2.59. PROGRAPH find subtree. From Matwin (1985).

In this figure a recursive algorithm for finding a subtree is shown.

The tree itself is treated as a multiple (it is fed in as list of nested lists). The tree to be matched is also fed in. The APPLY TO ALL function serves to supply the set of subtrees of the list to the recursive call in the THEN clause. A multiple OR at the end catches any successful find. Matwin notes that the OR can be implemented using lazy evaluation.

SHOW-AND-TELL

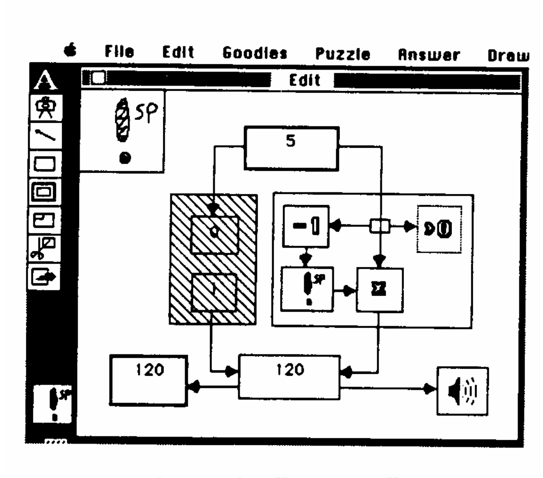


Figure 2.60. Show and Tell. From Gillet (1986).

Show-and-Tell (Gillet 1986) solves the iteration problem by using the conventions of a textual data flow language, Lucid. In Lucid, a sequence of values is kept for each variable. This historical sequence can be referenced the same way an array can be referenced. Ambler (1989) calls this a *temporally-dependent iteration construct.*

Show-and-Tell spatializes this idea by unfolding the multiple instances of a variable, accordion style. Show-and-Tell contains another interesting characteristic. If contiguous boxes have conflicting values, they are considered inconsistent. However, the inconsistency is isolated to the containing box.

PROGRAMMING IN PICTURES (PIP)

Raeder, in his Ph.D. dissertation (1984) describes his system, which uses FP as its main model. Raeder writes:

> ... the strict sequentiality of the imperative model is not compatible with pictures, where there is usually no sequential order imposed on various picture elements visible in a two-dimensional plane (p 104).

A function is defined by drawing a picture of the functions input data, the functions output data and the actions on the data. The underlying convention here is the data flow graph. The higher level of abstraction will be an iconic picture of the function. Then FP-style functional forms are used to combine functions.

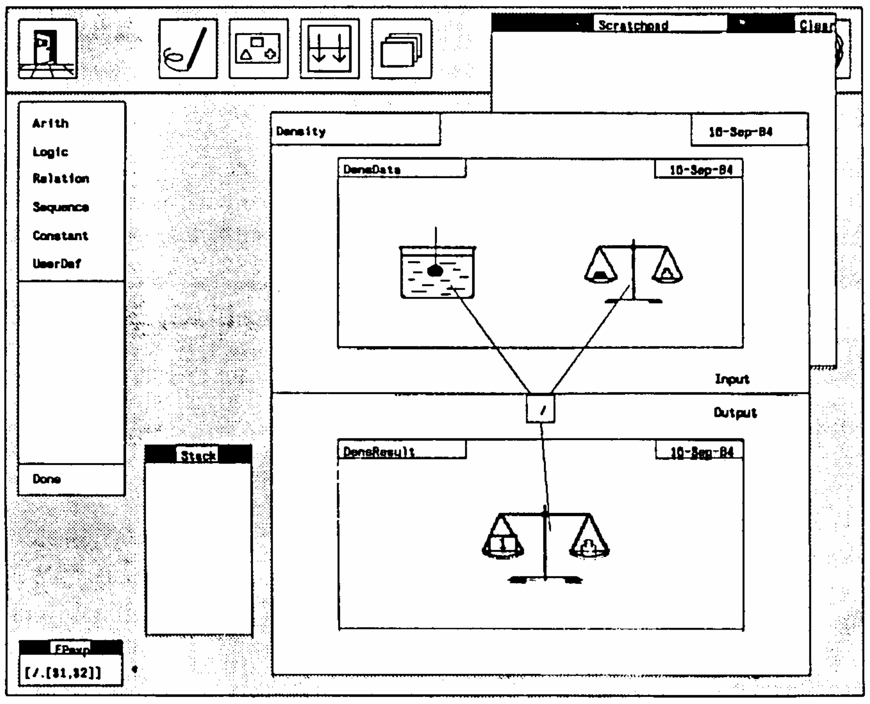


Figure 2.61. Programming in Pictures. From Raeder (1984).

A typing mechanism is added; this is outside of the FP model.

VISUAL FP

Pagan (1977) describes a visual version of FP.

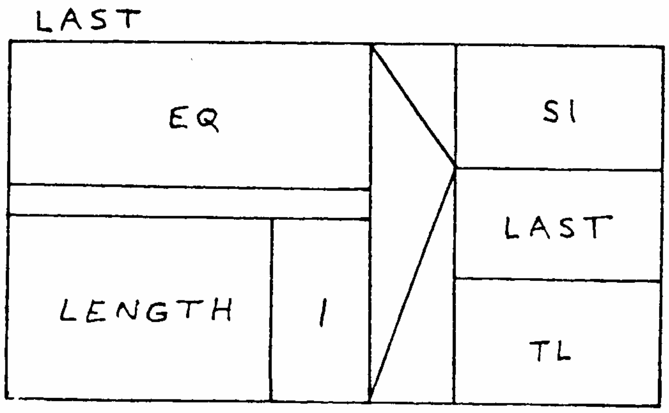


Figure 2.62. Visual FP. Pagan (1977)

It is different in appearance from the other functional languages we have just looked at. Instead of using data flow graphs as the visual conventions, it uses a modification of Nassi-Shneiderman charts. The box-shapes in Nassi-Shneiderman diagrams represent control structures; in graphical FP they represent different functional combining forms. The six combining forms are:

constant -	rectangular box
construction -	two side by side, slab above
composition -	one above the other
insert -	box within a box, 2 corners connect
apply to all -	box in box, criss-cross
condition -	arrow.

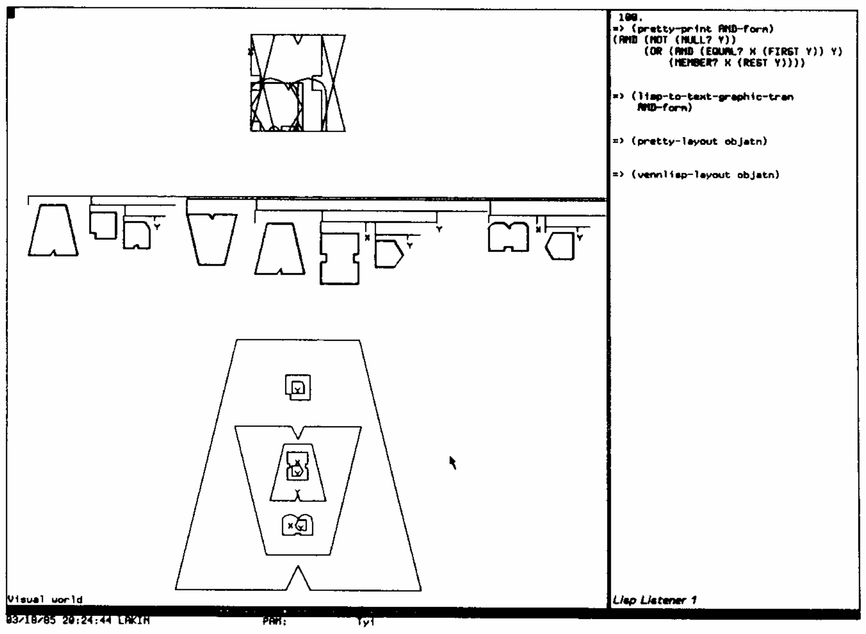
The system is very simple. As with other Nassi-Schneiderman based systems, text still predominates much of the representation. However, some of the linear terseness of the functional forms of FP seems to have been relieved by the 2-dimensional representation.

2.6.3. Miscellaneous Visual Languages

Our analysis has not been exhaustive; there are many other systems that are documented and discussed in the literature. We mention briefly some of the systems that often are cited.

VENNLISP

Lakin (1986) describes a LISP system in which expressions have been replaced by graphics using the convention of enclosure. It is a variation on a parse tree. The functions themselves are encoded as different enclosing shapes.

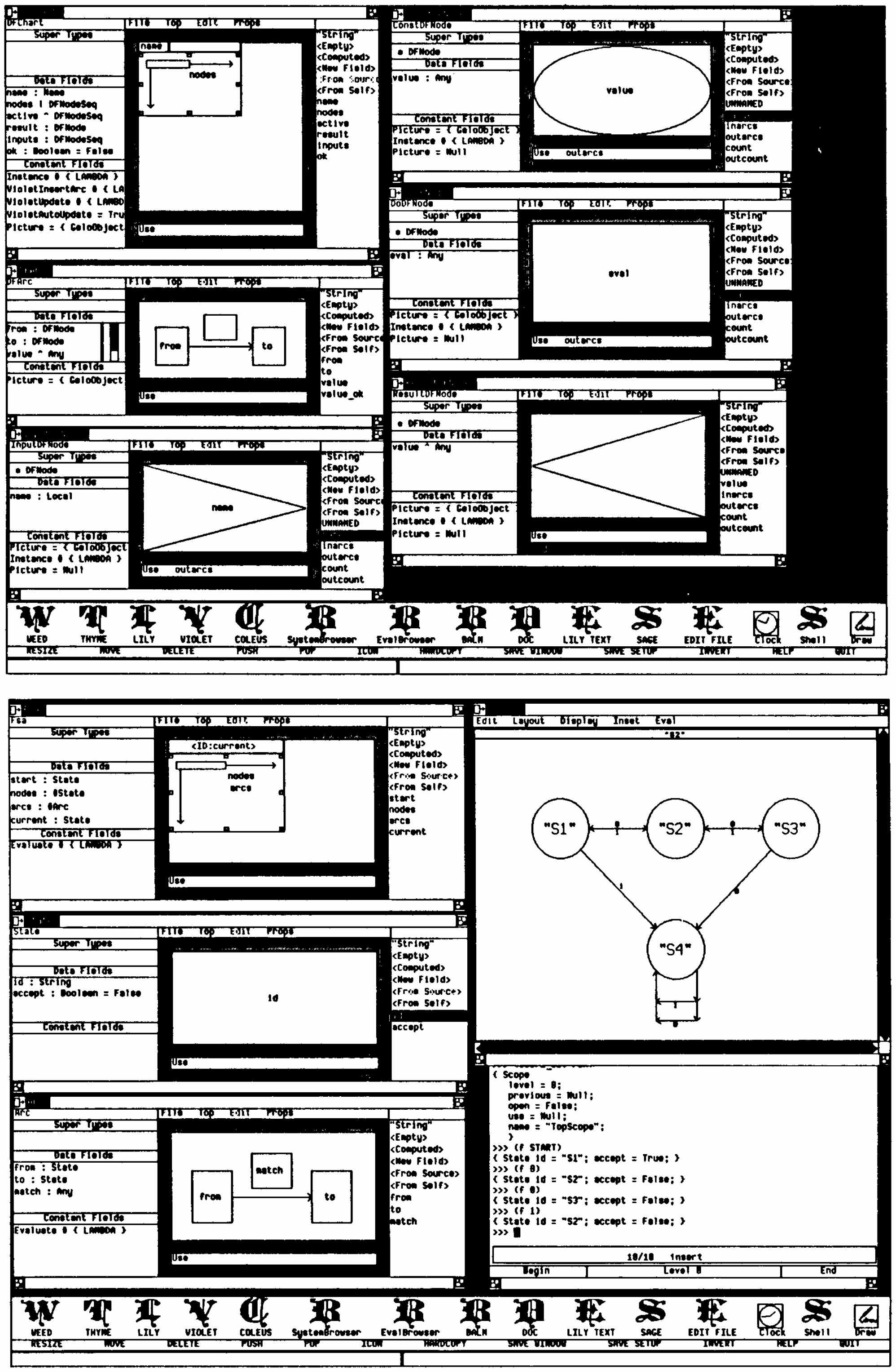


```
 100.
=> (pretty-print AND-form)
(AND (NOT (NULL? Y))
     (OR (AND (EQUAL? X (FIRST Y)) Y)
         (MEMBER? X (REST Y))))

=> (lisp-to-text-graphic-tran
    AND-form)

=> (pretty-layout objatn)

=> (vennlisp-layout objatn)
```

Figure 2.63. An enclosure convention for lisp. From Lakin (1986).

GARDEN

Reiss (1987) describes a system for building visual languages. In a sense it is a meta-visual programming language. The system is shown building an FSA language, a flowchart language, and a data flow language. The underlying language is LISP. The pictures can be executed.



```
{ Scope
   level = 0;
   previous = Null;
   open = False;
   use = Null;
   name = "TopScope";
   }
>>> (f START)
{ State id = "S1"; accept = True; }
>>> (f 0)
{ State id = "S2"; accept = False; }
>>> (f 0)
{ State id = "S3"; accept = False; }
>>> (f 1)
{ State id = "S2"; accept = False; }
>>>
```

Figure 2.64. Garden. From Reiss (1987).

2.7. SUMMARY

In the course of this chapter identified a way of speaking about diagrams in terms of metric, topological, and symbolic realms. Within the topological realm we identified three main conventions, those of adjoinment, linking, and containment, that when combined can be used to describe all topological diagrams. Using these distinctions, we covered the field of software representation, drawing examples from textbooks and journals on computer science. We identified and described flow charts, data flow diagrams, signal processing diagrams, layer charts, and a variety of other diagrams in formal visual terms.

Also discussed were a variety of systems built to develop and visualize computer programs. Each of these systems made use of the diagrammatic conventions we identified. Most of the systems were based on functional languages, which are more suited to visualization because of their simpler parameter schemes, their concept of single-assignment, and their data driven nature. It is from the functional side that we begin in creating some alternate visual languages.

We noted throughout the discussion some inherent limits in the visual realm. We discussed the representation of recursion, and pointed out that the visual cannot express this symbolic concept, and that this is a warning sign about the limits of the visual. We will return to these limits in Chapter 7.