

The following paper was presented at SoftVis '99. It may be cited as:

Programming in Several Dimensions, Paul Lyons, *Proc. SoftVis '99*, (ed. A Quigley), December 3-4, 1999, 31-39

Programming in several dimensions

P.J. Lyons

*Institute of Information Sciences and Technology
Massey University
Private Bag 11222,
Palmerston North,
New Zealand
Ph. 0-6-35057992472
fax 0-6-3502259
email P.Lyons@massey.ac.nz*

Abstract

Writing a program to solve a problem is a process that can be divided into two phases: first, we invent a *mental model* of the solution; secondly, we map the mental model onto a *physical representation*. The mental model is multi-dimensional and syntax-free; in today's textual programming languages, the physical representation is single-dimensional and syntax-burdened. In fact, it hasn't changed greatly since Algol 60. Mapping from one representation to the other has remained a painstaking and error-prone task, in spite of the ready availability of immensely faster computers, massive amounts of memory, high-resolution graphics displays, and powerful graphic input mechanisms.

The Hyperprogramming paradigm exploits these capabilities. A hyperprogramming language employs different visualisations for different program components - for example one visual syntax is suitable for visualising algorithms and another is suitable for visualising subroutine nesting. Each visualisation is designed for minimal overlap with the others, and where overlap is essential, hyperlinks between the views are automatically provided to allow easy navigation between them, and automatic updating of shared information.

HyperPascal was developed as a vehicle for exploring this idea. In creating a program, a HyperPascal programmer edits information in three main visualisations:

- the *action window* visualisation, which represents algorithms using a visual language based on structure diagrams;
- the *data structure templates* visual component, which represents dynamic data structure algorithms using before-and-after pictures
- the *scope window* visualisation, which represents declarations as a nested visualisation analogous to conventional subroutine nesting

Keywords

Software visualisation, HyperPascal, Data Structure Templates, hyperprogramming, multiple views

1. Background

Textual languages are single-dimensional

Conventional text-based languages are one-dimensional. Although it is a truism to say that this detracts from their usability, we begin by rehearsing some of their drawbacks.

Text is the traditional medium for encoding computer programs. Indeed, for a long time, textual input was the only feasible and affordable way of conveying complex instructions to a computer. A few sophisticated computers used digitisers, but their cost and unwieldiness meant that they were only practical for capturing special-purpose graphical data, such as maps, and plans. Although they are the antecedents of the mouse, they were never treated as a general-purpose graphical input device. As with input, so also with output - textual output was historically far cheaper to produce than graphical output, and even today, with the ready availability of high-resolution graphic output devices, text output is still far simpler to deploy than graphics routines.

Another subtly linearising influence was language

designers' knowledge of how conventional computer architectures are organised. Memory is arranged sequentially, at least at the fine grain level; data buses transfer word-sized amounts of information sequentially; single-accumulator machines can only process complex information when it is presented sequentially. Given that the processor works in so linear a fashion, it's easy to leap to the conclusion that instructions to the computer are most naturally expressed in a linear mode.

Finally, powerful techniques for parsing linear text have long been known, and are easily incorporated into most compilers. Language implementors who find it too tedious even to write the code for these techniques can take advantages of such systems as YACC, which will automatically assemble a parser for a linear language, given only lexical and syntactic specifications.

Programs are multi-dimensional

Computer developments fall into two broad classes; those

that make more efficient use of the computer (implicit assumption: computers are expensive) and those that make the computer more convenient to use (implicit assumption: computers are cheap). All the linearising influences we've looked at belong to the first class; they simplify compilation, so that it will involve less system resources; they fit with simple (and therefore cheap) architectures, and so on. To be fair, there is one strong linearity argument based on user convenience: natural languages are linear, and, by implication, an intuitive programming language will also be linear. However, this argument does not stand up to analysis. It's certainly true that natural languages are restricted to expressing ideas sequentially. But this doesn't make them the ideal way of representing complex ideas. Anyone who has struggled to express a complex idea simply and elegantly will testify to that.

In generating programs, programmers are in precisely this position. They need to divide their attention between "getting the syntax right", designing and keeping track of complex data structures, remembering the variable names used to reference that data, remembering the static nesting of their program's components, or the inheritance structure of its objects, and so on. Keeping simultaneous track of all these activities and, in particular, mapping them onto a single linear medium, is, in large part, why programming demands great skill and concentration.

We might say that a programmer's main function is to generate a multi-dimensional mental model of a program and then map it onto a linear textual space.

Single-dimensional textual languages provide little support for programmers grappling simultaneously with these requirements. On the contrary, their rigid syntax magnifies the task. All the independent components of the program end up intermingled, and all its commas and semicolons have to be present, all its parentheses have to match and all its reserved words have to be correctly deployed. However, most of these structural language components have nothing to do with program's main purpose, its data processing. They correspond to no instructions in the resulting code. Their only function is to make life easier for the parser, by disambiguating the algorithm's textual representation. Unrelated to data processing though they are, these components can make up as many as 50% of the tokens in a program, and often appear in 100% of the lines in a program.

Of course the multi-dimensional program space is not a formal, mathematical vector space, and there are no formal functions for performing the mapping; if there were, the programming might not be so tedious. However, the metaphor helps us to understand the problem, and what might be done to reduce it.

Consider a program which will ultimately be mapped to Pascal. The programmer must maintain mental representations of at least three different and more-or-less independent aspects of the developing program. First, the data manipulated by the program occupies a tree structure corresponding to the subroutine nesting, the scope tree. Secondly, the algorithms employed to manipulate the data have their own flow-of-control structure which is largely independent of the scoping structure (except for

references to variables used to determine control flow). Generating individual data manipulations certainly involves knowledge of the scope tree, but in most situations, the programmer is only interested in two facts; the type of a variable, and (less frequently), whether it is local or global. The third aspect of the developing program with which the programmer is concerned is essentially invisible in most programming languages. This is summarised by the term *program state*, which can be defined as the set of values of all the program variables at a particular point in the program execution. It is not explicitly represented in programs, except in the Boolean expressions associated with conditional tests or guard conditions. However, the programmer must maintain this information as part of the mental model, for it controls the processing that can be performed at any point in the program.

We might surmise that something as inherently multi-dimensional as a program would not map easily onto a single-dimensional textual representation, and of course, this is the case. In the mapping, some aspects of the program are inevitably distorted out of all recognition, and only one or two aspects of the program maintain any sort of coherency in the text. In general, the structure of algorithms is seen as the most important component of the program, and is the aspect whose integrity is most carefully maintained. Conserving the integrity of the other program components is less important, and they have to be fitted in and around the algorithm wherever is syntactically convenient.

Computers support multi-dimensionality

An "interface budget" that would have seemed profligate 10 years ago is commonplace now. Many computing environments are now based on the model of cheap, high-powered workstations, and even stand-alone systems embody considerable computing power. We've suggested that programmers maintain a mental model of their problem solution which comprises several, quasi-independent views. In the current computing milieu, it is feasible to allocate much more computing resource to maintaining a program representation which is much closer to that multi-dimensional mental conception than is possible with conventional linear programming languages.

A program representation of an underlying multi-dimensional model needs some fairly sophisticated technology to support it. Let's review the requirements:

- high-resolution colour graphics screens, which can represent complex 2-D data directly, and 3-D data with a little work;
- support for multiple windows so that independent information structures can be represented independently;
- mouse-based input, which allows easy creation of complex, interlinked structures;

- considerable raw computing power for generating such pictorial representations, editing them in real time, and animating them;
- real-time support for navigation between separate but related views;
- real-time updating between related views

Modern computing systems provide all of these features, so it is an appropriate time to investigate the use of the multi-dimensional program as a replacement for the linear program.

2. Requirements of a multi-dimensional language

This paper proposes a programming paradigm in which the programmer edits a number of separate “projections” from the multi-dimensional mental space. The vocabularies and syntaxes of the different projections are tailored to the type of information they contain. Graphical vocabularies are used for projections with a heavy emphasis on relationships between items. Textual vocabularies are used where linearity is natural, and where existing textual vocabularies are highly refined. Each projection is a representation of a more-or-less independent aspect of the program, and where the information in different projections overlaps, automatic updating keeps the projections consistent wherever possible. If automatic updating is not possible, the programmer is able to perform the updating without exiting from the current projection. The programming language is integrated into a program development environment with appropriate tools for editing each sort of projection.

Other authors have supported the modelling of problem solutions as a collection of partially independent, but interrelated, mappings from, or views of, a single *idea space*. This accords with:

- Shu (10), who suggests the need for tools which provide multiple views of a program;
- Raeder (8), who asserts that programmers' reasoning about a program is fairly unstructured, and that program representations should provide them with direct, effortless, access to a multitude of notions;
- Reiss (9), who develops multiple-view but low-level, representations of a program, and
- Williams and Rasure (12), who warn against development of visual notations based on a single large graph.

3. HyperPascal

HyperPascal has been designed in order to provide a vehicle for experimenting with these precepts. The implementation was intended to help determine whether hyperprogramming is practicable (can a system be implemented with a reasonable amount of effort, and will it make moderate processing demands?) whether it can be

complete (is it possible to use hypertechniques to provide a practical representation for the whole of a general purpose programming language?) and whether it can be consistent (does the hyperprogramming paradigm lend itself to a small number of elegant, general constructs, or is characterised by lots of special cases and a huge vocabulary?).

The language's functionality is based closely on Pascal's, to make it easy to draw conclusions about hyperprogramming, rather than the functional capabilities of some new language developed for this experiment. Accordingly, it is called HyperPascal and it uses Booleans, real, integers and strings (not chars), procedural and functional subprograms, records, var parameters, and so on. The author's concern is with the difficulty of expressing conventional algorithms in conventional textual program code. Object-oriented languages have added another complete layer of complexity to programming languages, but the constructs used in object-orientation are largely orthogonal to the concerns addressed in the current work. Further, OO design paradigms incorporate much more graphics than traditional procedural languages (5). In view of this, it is appropriate to concentrate on the lower-level aspects of programming that have been incorporated into OO languages without updating, before attempting to work on visualisations of OO programs.

HyperPascal is a visual programming language. Its visual aspects are concentrated on the representation of relationships between program components, and not on the low-level components themselves. In particular, this means that programmers in HyperPascal do not create icons to represent data processing actions, and low-level data processing is not preformed visually. As Graf (4) points out, designers of visual languages should avoid developing a visual metaphor for every possible language component. For example, most programmers are not especially artistic, and can develop a large vocabulary of mnemonic identifiers more easily than a large set of instantly recognisable, meaningful icons. Further, the development of sophisticated notations for representing mathematical relations in particular has been taking place for several centuries. It seemed inadvisable to replace these notations *in toto* with visual analogues, and preferable merely to support and only augment them, where typographical constraints restrict their power.

Views

HyperPascal's representation of a program is divided into three types of view. The data processing and control structures of algorithms are represented in Action Tree views, each equivalent to a Pascal procedure or function. Each Action Tree is a node in the second view, the Scope Tree, which also houses declarations of all the other identifiers used in a program. The Forms Windows are the third visualisation, and they provide a WYSIWYG editing environment in which programmers specify the appearance of their program's input and output.

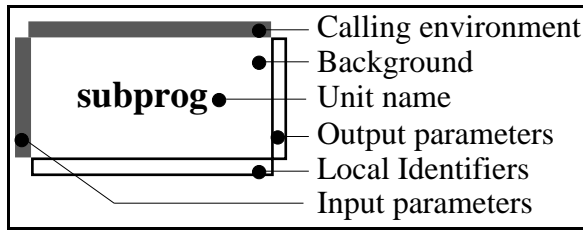


Figure 2: Components of a single program unit icon

Hyperlinks

Two of the three types of hyperlink currently supported in HyperPascal programs are related to identifiers. Opening an instance of identifier in the Action Tree (double-clicking on it, in a Macintosh system) leads the user to its declaration in the Scope Tree. This mechanism operates for all identifiers, including subprogram invocations, so opening these leads the user to their Action Tree representation. This is possible, of course, because their Action Trees are conceptually entries in the Scope Tree, though they are represented in independent windows. When a new (i.e. currently undeclared) identifier is used in the Action Tree, the program development environment automatically links the user into the Scope Tree, to add its declaration there, without requiring the user to move between the windows.

Next, the program development environment maintains, in the Action Tree and Scope Tree views of each subprogram, a list of the subprograms which it is invoked by, and which it invokes. Coupled with the Scope Tree, this gives programmers the ability to check on each subprogram's static and dynamic environments. To expert programmers, this may not seem a significant feature, but in a study dealing with the differences between expert and novice programmers, Fix, Wiedenbeck and Scholtz (3) identified "matching procedure names to the procedures they call" as one of the most significant differences between these two groups.

Finally, there are hyperlinks between I/O actions in the Action Tree and the associated Forms Window. The representation of I/O actions is minimised in the Action Tree; it is only necessary to state there that particular variables are being read or written, and the formatting information associated with the action is specified in the Forms Window. This allows the formatting of a group of I/O actions to be treated as an integrated whole, free of the distraction of the data processing actions that intervene between the I/O actions. The hyperlinks between these two views allow the programmer to move from the Action Tree to the associated Forms Window to format I/O screens, and back again to find the action environment of each component of an input or output form. There are also links between instances of identifiers in the Forms Window and the Scope Tree, which allow programmers to introduce and declare new identifiers while editing the Forms Window, again without the need to move between windows.

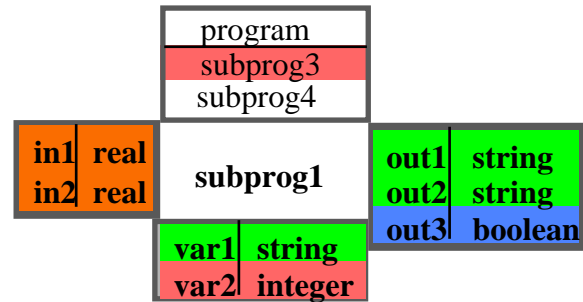


Figure 1: Expanded collapse bars for a program unit icon

The Scope Tree Visualisation

The identifiers in a HyperPascal program conform to the same scoping rules as identifiers in Pascal. That is, each subprogram is a node in a tree, the children of a node are the subprograms declared local to it, and the root node is the mainline of the program. Identifiers declared in a subprogram are in-scope within that subprogram and within its children, unless a name is redeclared at a lower level.

Figure 2 shows the program unit icon, the only type of icon used in this visualisation. Each program unit icon owns four *collapse bars*, which contain information associated with the program unit. They are white when empty, and grey when non-empty. They can be opened to reveal the information they contain (there are several types of sub-icons in HyperPascal, and the colouring and opening conventions are common to them all). The top collapse bar contains lists of the program units which invoke, and which are invoked by, this program unit. Opening an item from these lists creates a window to the Action Tree view of the corresponding program unit. The left and right collapse bars contain lists of the formal input and output¹ parameters, respectively of the program unit, if it is a subprogram, and the input and output files if it is the mainline (in HyperPascal information conventionally flows from left to right). The bottom collapse bar contains the declarations of all identifiers local to the program unit, (except other program units, which have their own icons).

Figure 1 shows the appearance of a program unit icon with all its collapse bars expanded, showing its input and output parameters, its dynamic calling environment (calling units listed above the bar, called units below), and its local variables.

Figure 3 shows a small Scope Tree containing several program unit icons. The programmer creates the Scope Tree using a simple graphic editor in which an icon is instantiated by a mouse click in the position where it is to appear. Dragging the cursor from an existing icon to an empty part of the window creates a new icon, and an automatic connection between it and the original icon.

¹Output parameters replace Pascal's **var** parameters. If appropriate, a parameter can appear in both the input and output lists.

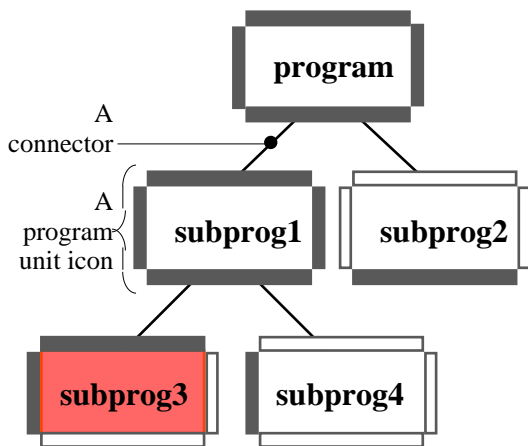


Figure 3: Four program unit icons assembled into a Scope Tree

Alternatively, dragging the cursor between two existing, but unlinked, icons, creates a connection between them. HyperPascal uses colour-coding to denote types, and to indicate type-compatibility (for colour-blind users, this is also indicated in other ways). One of the program unit icons in Figure 3 has a red background, to indicate that it is a real-valued subprogram (i.e. a function).

The Action Tree Visualisation

Action Trees are a little more complex than the Scope Tree. The notation is based on Doran and Tate's (1, 2) Structure Diagrams. Each Action Tree represents the algorithm for a program unit in the Scope Tree. It shows the data processing actions performed by the program and the flow of control through the actions.

The icons used for the program unit in the Scope Tree and the root of the program unit's Action Tree are identical. The data-processing actions specified in the Action Tree are traversed in left-to-right, depth-first order, except where choice and loop flow of control icons appear. The subtrees of loop icons (round-cornered boxes) execute repeatedly in left-to-right order, until the loop's exit-conditions become true, and while its continue-conditions remain true. Only one of the subtrees of a choice icon (hexagons) - the left-most subtree whose control condition evaluates to true - is executed.

Rectangular parent nodes, other than the root, only contain comments. Rectangular leaf nodes specify sequences of actions to be executed in top-to-bottom order.

Editing Action Trees is similar to editing the Scope Tree, except that as there is a choice of icons, a pop-up menu appears whenever an icon is being created to allow the programmer to select the appropriate icon.

Figure 4 shows an algorithm for finding the maximum of a set of positive integers. It repeatedly tests the value of the latest value input, exiting the loop when zero is input, and otherwise replacing the value of the variable max-so-far with the greater of itself and the input value. Item *a* is the program unit icon, with its output

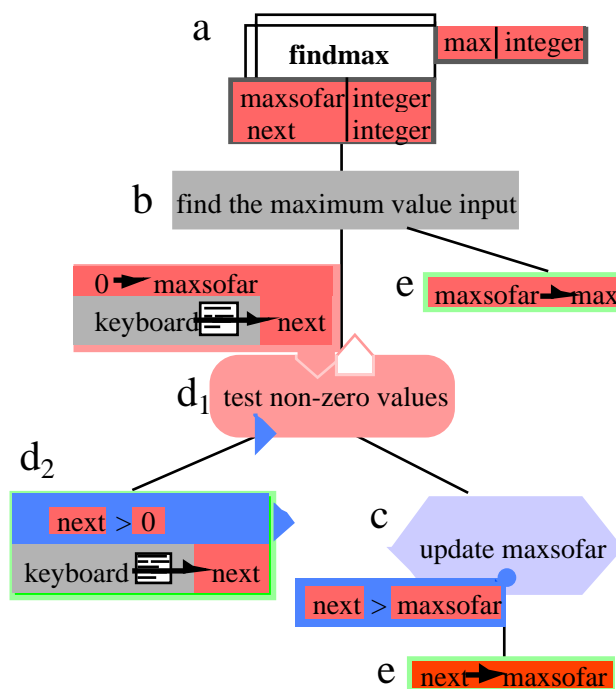


Figure 4: An Action Tree representation of a simple algorithm

parameter and local variable collapse bars expanded. Item *b* is a comment (rectangular non-leaf nodes hold comments; rectangular leaf nodes contain sequences of data processing actions). Opening the comment will produce an editor for a larger comment, with the short comment as its heading. Item *c* is a choice icon; it contains a comment, and the condition governing execution of its subtree is contained in a circular sub-icon, here shown expanded form to reveal the condition. More condition sub-icons can be added to the choice icon to provide more choices. By default, the last condition sub-icon contains an *else* condition. Item *d1* is a loop icon. It also contains comment text. There are also three sub-icons shown. The left (down-) arrow box on the top of the icon can incorporate code to be executed before the loop starts. This maintains a close "physical" association between the loop and its initialisation code, so that if, for example, the loop icon is moved to another position in the Action Tree, the sub-icon moves too. The adjacent up-arrow has a similar function; it can contain code to be executed after the loop terminates. The triangular, right-pointing *continue condition* sub-icon contains a condition sequence, expanded in item *d2*, and comprising a condition which must evaluate to true if the loop is to continue, and an action which is executed if the continue condition returns true. Octagonal *stop conditions* can also be placed on the bottom of the loop icon. Child subtrees of the loop are repeated while any *continue conditions* are true, and until any *stop conditions* become true. Either or these kinds of condition can be placed anywhere in the left-to-right sequence of execution of the loop's subtrees, and will be tested at the corresponding

time in the sequence. Finally items e contain the data processing actions performed by the algorithm. They both contain single assignments, though multiple assignments are possible. Only assignments and subprogram invocations (which closely resemble assignments) can occur in an action sequence. I/O operations are a special case of assignment.

The Structure Templates sub-visualisation

Within the Action Tree, structure manipulations are represented by a special-purpose visual syntax - the Data Structure Template (7).

The syntax was developed as a reaction against the conventional, arrow-and-dot notation (e.g., $node^{next} := nextnode$) for manipulating data structures. The notation is a very obscure abstraction for a difficult concept, and it causes programmers great difficulties. The Structure Templates visualisation replaces the dot-and-arrow notation by diagrams of data structures before and after they have been changed.

First, it's necessary to tell the system how to draw the nodes in the data structure. To this end, the HyperPascal development environment incorporates a simple drawing editor. The programmer declares a record by drawing a picture of it, using the drawing editor, and specifying the types of its fields (recursively if necessary). Then, when the programmer uses a pointer that points to a record of that type (or a variable with the type of the record) the system knows how to draw the record. Furthermore, if the record contains a pointer field, the tail of the pointer is shown in the field, and the user can drag it out from the record. The system responds by drawing, first an arrow head with a symbol representing null, and then - if the user keeps dragging the pointer - a node of the type the pointer references. The node may be of the same type as the parent node, as in the case of a linked list, or it may be of a different type. The programmer can insert values into the fields of the nodes in the growing data structure diagram.

The picture that the system draws, under the programmer's control is a Before-picture. In essence, it is a complex conditional, specifying that *if* the data structure looks like *this*, then...

Then what? ..then the programmer specifies an After-picture which is the new appearance of the data structure. In fact, After-pictures are usually very similar to Before-pictures, so the HyperPascal system draws an After-picture that is identical to the Before-picture, and the programmer edits this to reflect the changes that will occur in the data structure.

This is not programming by demonstration; the programmer is not giving an example of what to do to a data structure and expecting the system to generalise from that example. Instead, the programmer produces a series of Before-pictures that specify the complete algorithm. In the current version of the system, this involves drawing the succession of diagrams - though the drawing is a low-stress activity, because the system does all the hard work. However, recall that the before-picture is a complex

condition. This can be expressed as a series of nested *If*-statements. Now, in an *if-then-else* statement, we don't normally write down the conditional expression that governs the execution of the *else*-clause, but we can work out what it is. In this *if-then-else* statement:

```
if a < 3 then
  if { a < 3 and } b > 5 then <statement1>
    else { a < 3 and b >= 5 } <statement2>
else { a >= 3 } <statement3>
```

the *else* conditions have been written out explicitly.

Similarly, in the data structure manipulation, once the programmer has drawn one Before-picture, it is possible for HyperPascal to derive a series of *else* clauses that match the nested conditions in the Before picture. HyperPascal can then draw a Before picture corresponding to each of these conditions. And since when HyperPascal draws a Before-picture, it also automatically generates a corresponding After-picture, which the programmer edits to describe the operations on the data structure, the state information that is readily available in the semantic tree is not only driving the development of the algorithm, but also doing a lot of the tedious drawing that would otherwise be involved in representing pictorial data structure algorithms. Figure 5 illustrates an insertion in to an ordered linked list. The Before-Picture (on the left is a complex conditional, corresponding to the conventional Pascal code:

```
if L <> nil then
  if L^.data > x then { create After-Picture }
```

It is possible to generate *else*-clauses matching these conditions:

```
if L <> nil then
  if L^.data > x then { implement After-Picture1 }
{1} else { L <> nil and L^.data >= x } {AP2}
else { L = nil } {AP3}
```

and also to draw the corresponding Before-Pictures automatically. The programmer then needs only to edit the automatically-drawn prototypes of the After-Pictures in order to specify the actions to be taken under each circumstance. Note that it is sometimes necessary for the programmer to edit one of the automatically-generated before pictures. for example, the programmer will probably want to distinguish between the cases when the current node in the linked list has a value less than x and when it has a value greater than x . Editing the conditional in an automatically-generated Before-Picture will result in the automatic generation of another Before-Picture, containing the condition that has been removed from the edited picture.

That is, if the programmer changed the Before-Picture corresponding to {1}, above, to {1'}. below} then the system would automatically generate {2} and a corresponding Before-Picture.

```
if L <> nil then
  if L^.data > x then { implement After-Picture1 }
{1'} else if L <> nil and L^.data < x then {AP2}
{2} else { if L <> nil and L^.data = x } then {AP3}
else { L = nil } {AP4}
```

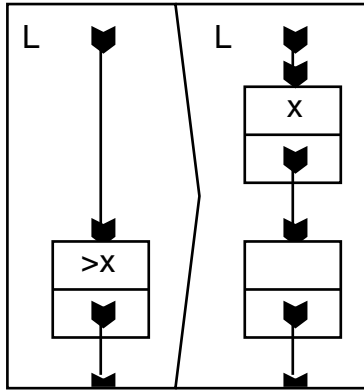


Figure 5: A Before-After-Picture-Pair showing an insertion into an ordered linked list

Hyperlinks revisited

HyperPascal automatically supplies hyperlinks between related items of information in the various views. All identifiers, wherever they occur in the Action Tree or the Forms Window, are linked to their declarations. The programmer can therefore check on the scope of simple variables (their type is colour-coded), and the scope and type of complex variables just by double-clicking on them.

Implementation

A preliminary implementation of HyperPascal was written in Snart, an object-oriented version of Prolog, and the inter-view updating is supported by another Snart application, MViews. This application maintains a single *base view* for a set of interrelated *display views*. When any information contained in a display view is altered, MViews updates the base view is updated, and adds an update record to a queue of updates for other affected display views. It is possible for the other views to be updated in real-time, or for the updates to remain queued until the other window is selected. MViews and Snart were produced by Grundy and Hosking (6). Because of the number of layers in this system, it was impossibly slow, although it could be used to produce workable code.

Currently, a Delphi interpreter is being implemented. Figure 6 shows how this version of the system is structured into three levels. The bottom level, layer 3, contains the semantic representation of the program, essentially a parse tree. Layer 2, the middle level, is under the control of the 1:1 Graphic handler, which manages an undisplayed graphic representation of the complete image of a particular visualisation (Action Tree or Scope Tree). Layer 1 contains concrete views. At this level, the image maintained by the 1:1 Graphic handler is mapped onto a variety of distorted coordinate systems - each concrete view has its own coordinate system. The concrete views could be produced by general distortions of an image, or could be produced by a deeper analysis of the structure of the program graph, such as Storey, Fracchia, and Muller have outlined (11)

The user interacts with HyperPascal at the level of the

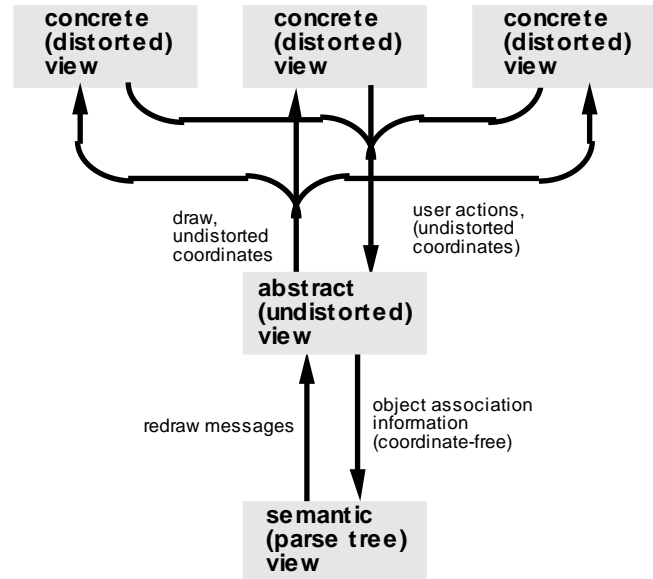


Figure 6: The three-layer architecture used in the current implementation of HyperPascal

concrete view. User actions are converted by layer 1 entities from their individual coordinate systems to the undistorted coordinates of the 1:1 Graphic handler, which translates the graphical input into semantic information, and sends the information down to the semantic layer.

The semantic layer updates the program's parse tree, and sends graphic update messages back to the 1:1 Graphic Handler. This updates its invisible image appropriately, and also sends update messages to the layer 1 entities, which translate the Graphic Handler's undistorted coordinates to their own individual coordinate systems. Because knowledge of the distorted coordinate systems is isolated to the concrete view entities, this is a low-impact way of experimenting with a number of distortion-oriented displays, in order to address the problem of inefficient use of screen real estate that Visual Programming Languages commonly suffer from.

4. Conclusions

The author has been concerned with problems inherent in the conventional approach of using text to represent procedural programs. In particular, conventional textual languages are complicated by the necessity to render a number of related, but different, concepts into a single textual stream. Object-oriented languages have reduced the scale of this somewhat, (though they could be said to replace static complexity with dynamic complexity) but the underlying problem remains: programmers' mental models of algorithms are only poorly represented using text. This project has been an attempt to design a complete system that uses the capabilities of current, commonly available, computing systems, to provide a better match between the mental model and its instantiation in program code. The purpose of the project has been to determine whether such a system can be practicable, complete and consistent. At the current stage of the project, what conclusions can we draw about

hyperprogramming in general and about HyperPascal in particular?

Hyperprogramming systems are practicable. The implementation of this first system proved to be a significant, but not overwhelming programming task. This is mainly because the areas which seemed naturally separate in the programmer's mental model were found to be equally independent in the implementation - confirmation, though a little indirect, of the original theoretical partitioning. Further, and again consistent with our original hypothesis, it was possible to implement only a small number of types of hyperlink to connect the overlapping parts of the different views. In any particular program, of course, the number of actual hyperlinks will probably be quite large, as every identifier, for example, is hyperlinked to its declaration. However, the behaviour of all these hyperlinks is identical, so the cognitive load on the programmer is not proportional to their number.

Hyperprogramming systems can be complete. An important aspect of this project was to demonstrate that the hyperprogramming concepts could provide a practical representation for the whole of a general purpose programming language. One obvious way to verify this was to design a complete new hyperprogramming language. However, designing a totally new language has disadvantages: it is difficult to ensure that essential but obscure features aren't missing from a new and untested language; creating such a design would take an inordinate amount of time; it would be impossible to evaluate hyperprogramming except in the context of the new language. This last point is the most important, as the hyperprogramming approach is not intended to be specific to a particular language. A simple way of avoiding these problems was to apply the hyperprogramming ideas to an existing language. Pascal is a small, but general purpose programming language, and was ideally suited for this purpose. It has been possible to incorporate all of Pascal's functionality into HyperPascal, and in fact, the greater power of the graphical environment to represent relationships has allowed some Pascal constructs to be grouped into a smaller number of simpler and more

general constructs in HyperPascal.

Hyperprogramming systems can be consistent. Reference has been made above to the reduction in the number of control constructs required in HyperPascal when compared with Pascal. The visual environment, rather than hyperprogramming *per se* is the main reason why this was possible. It is possible to indicate relationships between components of a diagram by their positioning, and, in such an environment, it is immediately obvious that constructs such as while and repeat loops are special cases of a single, general loop construct, in which termination and continuation conditions can be positioned arbitrarily. Hyperprogramming allows us to pursue this type of advantage further; just as the single window visual environment allows us to separate unrelated programs and bring together related ones, so a multiple window environment allows us to design separate visual vocabularies and syntaxes to represent unrelated components of a program. Of course, textual languages do provide some support for this sort of syntactic separation - declarations have a different sort of syntax from assignments, for example - but the syntax of a textual language is to some extent overwhelmed by the need to distinguish between the different language components. A multiple-window environment does away with this. Different components of the language occupy different types of view, and can each have their own, tailored syntax. There is no ambiguity if the syntaxes are similar, and indeed, similar editor functions can be applied to them if they are. More importantly, the syntax can be kept consistent with the object being modelled when that object occupies a type of window particular to it.

The importance of HyperPascal is not just that it is a visual language. Equally important is the power which hyperlinks give to the programmer to check, with minimum disruption to his or her flow of thought, on the relationships between program components, to create new structures without having to leave the present location in the program, to maintain an accurate mental model of related but different parts of a program.

References

- [1] An Approach to Structured Programming, Part I, Doran, B., and Tate, G., 1972, Massey University Department of Computer Science Publication no 6.
- [2] An Approach to Structured Programming, Part II, Doran, B., and Tate, G., 1972, Massey University Department of Computer Science Publication no 9.
- [3] Mental Representations of Programs by Novices and Experts, Fix, V., Wiedenbeck, S., and Scholtz, J., 1993, *INTERCHI '93*.
- [4] *UML Distilled: Applying the Standard Object Modelling Language*, Martin Fowler with Kendall Scott, Addison Wesley Object Technology Series, 1997.
- [5] Visual Programming and Visual Languages: Lessons learned in the Trenches, Graf, M., 1990, in *Visual Programming Environments: Applications and Issues*, ed. Ephraim P. Glinert, 452-455
- [6] Integrated Object-oriented Software Development in SPE. Grundy, J.C., and Hosking, J.G., 1993, *Proceedings of the 13th New Zealand Computer Society Conference*, II, 465-478
- [7] Active Templates: Manipulating Pointers with Pictures. P.J. Lyons, M.D. Apperley, A.G. Bishop, 1994, *OZCHI '94 Proceedings*, November 1994, pps

- [8] A Survey of Current Graphical Programming Techniques, Raeder, G., *Computer*, **18**, 8, 1985, 11-25.
- [9] PECAN: Program Development Systems that Support Multiple Views, Reiss, S.P., 1985, *IEEE TRANS Softw. Eng*, **SE-11**, 3, 1985, 276-285
- [10] *Visual Programming* Shu, N.C., 1988, p48, Van Nostrand Reinhold Co. Inc., New York.
- [11] Customizing a Fisheye View Algorithm to Preserve the Mental Map, Storey, M.A., Fracchia, F.D. and Muller, H.A., *Journal of Visual Languages and Computing* **10**, 1999, 245-267,
- [12] A Visual Language for Image Processing, Williams, C.C., and Rasure, J.R., 1990 *1990 IEEE Computer Society Workshop on Visual Languages*.