Paul Lyons, Craig Simmons, and Mark Apperley,
HyperPascal: A Visual Language to Model Idea Space
Proceedings of the 13th New Zealand Computer Society Conference, August 1993, 492-508

HyperPascal: A Visual Language to Model Idea Space[1]

Paul Lyons, Craig Simmons, and Mark Apperley
Department of Computer Science
Massey University
Palmerston North
P.Lyons@massey.ac.nz

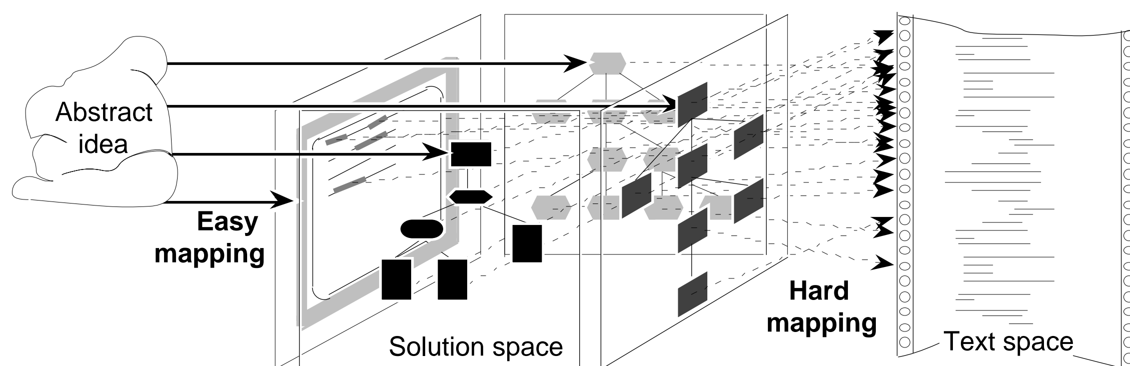
ABSTRACT

Programmers develop problem solutions in an abstract *idea space*, where they can easily visualise different views of the solution. However, conventional programs exist in sequential *text space,* where these different views often become inextricably tangled. We reject the text-based single-sequence structure - which many visual programming languages have heedlessly adopted - in favour of a hyperspatial environment where optimally-structured, disparate, models of a problem solution can be fashioned, where model integration is facilitated, and where hyperspatial navigation is intuitive.

To explore the hyperspace, we invented HyperPascal, a general-purpose visual programming language with the capabilities of Pascal. The language exploits the power of interactive graphic interfaces, automatically generating syntactically correct programs, but constraining the programmer's actions minimally. Later developments will go beyond the procedural paradigm to explore the relationship between higher level system design paradigms (like OOD) and hyperspace.

1. The *idea space* and the *text space*

There are various representations for computer programs, such as high level, assembly, and machine language. It has seemed natural for the representations to be sequential, because machines generally execute code sequentially, and because conventional memory designs are still based on von Neumann's sequential model. The drive towards sequence influences program appearance more than programmers' multidimensional, *gestalt* conceptualisations of problem solutions; much complexity in programing is organising a mapping from mental hyperspace to linear physical medium (see Figure 1).



[1]published in: "Applying the Future Today: Proc. 13th NZ Computer Society Conference, August 1993 492 - 508

Figure 1: We should only have to do the **easy** mapping!

We suggest that problem solutions are better modelled as a collection of partially independent, but interrelated, mappings from, or views of, a single **idea space**. This accords with:

Shu (1988), who suggests the need for tools which provide multiple views of a program;

Raeder (1985), 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 (1985), who develops multiple-view, though low-level, representations of a program, and

Williams and Rasure (1990), who warn against development of visual notations based on a single large graph.

A complete solution to a programming problem can be separated into a number of views, amongst which are a group of actions, a related group of data items, and a group of states (conditions which hold at a particular point in the program's execution). The first two of these are often organised, in structured programs, into trees; the third is generally ignored, but can be similarly tree-structured.

More restricted aspects of the problem solution may also warrant their own explicit views. For example, to obtain an integrated view of the appearance of the input and output of a conventional program of any complexity, it is necessary to extract a sequence of individual read and write statements from amongst all the calculation and flow-of-control specifications and trace this sequence through, building up the format item by item. A Visual Programming environment has the potential to provide a high level, more integrated, approach to visualising the appearance of a program's I/O.

These partially disjoint views of a program's abstract solution are easy to conceptualise in the abstract. It is not so easy to identify the contributions they have made to a text-based program as the physical sequences that they map to are interwoven throughout the program code. In order to achieve the desired structural support, it has been necessary to adopt intrusive constructs such as **begin**s and **end**s, **then**s and **else**s, a variety of loop control constructs, separation of data manipulation specifications (statements) from data storage allocation (declarations) and on and on[1].

Here we describe an approach to programming in which the different views of the idea space can be mapped separately onto a physical representation while the close connections between their related components are easy to discover.

2. Important Views of a Program

2.1 The Action Tree

The various alternative action sequences specified by a conventional procedural program have more influence on its appearance than the other idea space views. The

[1] Why are these structuring constructs intrusive? Because they don't manipulate any data, but, as text, they closely resemble the constructs that do. Thus, at one level, the reader has to filter them out, and at another, remember their significance.

action sequence view can be separated from the others, and organised into a tree of subroutine (procedure and function) nesting, and nesting generated by choice and iteration statements. We shall therefore call this the Action Tree view of a problem solution. In a conventional program, all the other views are made to fit in and around the Action Tree view.

When programmers are instructed in the techniques of structured programming, they are encouraged - or coerced - into developing a two-tiered view of the Action Tree; a top-level tree consisting of the subroutine calls, and a set of lower-level trees consisting of the nested statements in individual subroutines. The latter view is comparatively easy to visualise, though its mechanical skeleton (**begin**s, **end**s etc.) is tedious to assemble and intrusive (see above); the former view, though formally present in the written program code, is masked by the greater space taken up by the statement sequences contained by the subroutines.

We assert that the data manipulation aspects of the Action Tree should be distinguished from its structural skeleton, but that the skeleton should remain on display.

2.2 The Scope Tree

The structure of the Action Tree, down to subroutine level (and lower, in languages which allow declarations to be interspersed amongst statements) is mirrored by the structure of the Scope Tree. Identifier declarations located at a node in the Scope Tree are visible only to actions at or below the matching node in the Action Tree. Although the structure of the Scope Tree mirrors the structure of the Action Tree, and, in programs written in procedural languages, declarations are incorporated with the code for procedures, the Scope Tree's structure is masked by large amounts of action specification (statements). Note that in practical situations, the ability of a program to access implicit subroutines, and externally-defined subroutines requires the host computer's directory structure to be part of the Scope Tree, but any structured view of this part of the program's environment is rarely available.

We assert that the Scope Tree should be revealed clearly, away from the context of the Action Tree.

2.3 Forms Windows

Programs are ultimately written to produce output. Indeed, it would not be too fanciful to view the design of a program's output as the first (top-level) task in a top-down programming exercise. However, programs are rarely designed from their output appearance down. In part, this is because programmers see the design of complex data manipulations as their "real" activity. It is also, we believe, because of the difficulty of designing formats globally when the formatting specifications are interspersed, in small amounts, through the program code.

A program's I/O activity is data-dependent; whether an action occurs or not, and the number of occurrences of an action may depend upon data. Nevertheless it is possible to analyse a program and assemble its I/O actions into sequences that are uninterrupted by branches, that is sequences which must occur uninterrupted. One could design a forms editor for specifying the appearance of the input or output of each of these sequences. The editor would be like a word processor, except that it would include fields for the data values which were to be input or output.

We assert that programmers should be able to specify the appearance of large sections of input and output, using a word-processor-like Forms Window. Like all other hyperspatial navigation, transitions between the Action Tree and the Forms window should be as transparent as possible.

2.4 The State Tree

It is commonly held that one cannot predict the behaviour of a program unless one traces its action through with a particular set of data, and this is generally true. It is also true, however, that, for a program's execution to have reached a particular point in the Action Tree, some of its variables may be restricted to certain values. For example, at point {1} in the statement

```
if a<15 and b>7
  then
    begin
      {1}
      write(a);
      write(b);
    end;
```

the values of a and b must be less than 15, and greater than 7, respectively. Of course if the operator in the Boolean Expression had been an **or** instead of an **and**, our knowledge at point 1 would have been far less precise. It is, however, clear that at any point in a program the control structures within which that point is nested allow us to make assertions about the state of its variables.

Another example may serve to illustrate the point further. In generating code for traversing a complex dynamic data structure, programmers frequently find it difficult to visualise the context of the current action without resorting to diagrams of the data structure in some appropriate configuration, and with appropriate data in its fields. Now, it is generally impossible without the actual data to predict the form of the actual data structure, so it would seem impossible to display its appearance. However, a restricted amount of information must be available, or it would be impossible to write the code for manipulating the structure. In general, the structure of the current node is known (it is implicit in the declaration of the pointer referencing it); some of its fields may be partially or completely specified (by preceding Boolean conditions, by the fact that it has just been created, or by assignments), the name of the pointer which references it may be known. Making a diagram of even this amount of information available as the programmer designs the program will be a powerful guide to correct programming.

Because of the nested structure of the data manipulations which allow access to particular points in the program, this partial state information is another tree mirroring the structure of the current subroutine in the Action Tree. Only information from within the current subroutine is available, as the data dependence of subroutine invocations in general prevents us from predicting conditions more generally.

We assert that the State Tree should be made visible to programmers while they are generating a design.

Having suggested that these three trees and the Forms Window (and possibly also representations of other pertinent information structures) should be available to the programmer, we also assert that navigation within and between the structures, and editing of the structures should be easy and intuitive.

In this paper we concentrate on a proposed syntax for the Action Tree leaving full description of the Scope and State Trees, of the Forms Window, of the editing

environment, and development of representations for higher-level system design concepts, for later publication.

3. Structure Diagrams: Snapshots of the Action Tree

Doran and Tate (1972a and 1972b) invented a notation which used a tree-structured diagram, and control structures represented as easily recognised icons, to show the tree-structured nature of program components. The notation was intended to replace GOTO-oriented flow diagrams as a program design tool, to make it "as easy as possible for us humans to understand and to develop algorithms". In the notation, statements and comments are shown in rectangles, choices are shown in hexagonal boxes, and loops are shown as subtrees of round-ended boxes. Actions are performed in depth-first traversal order. The Structure Diagram in Figure 2 shows an algorithm for finding the maximum and minimum values stored in an array, a[1..100] of integer.

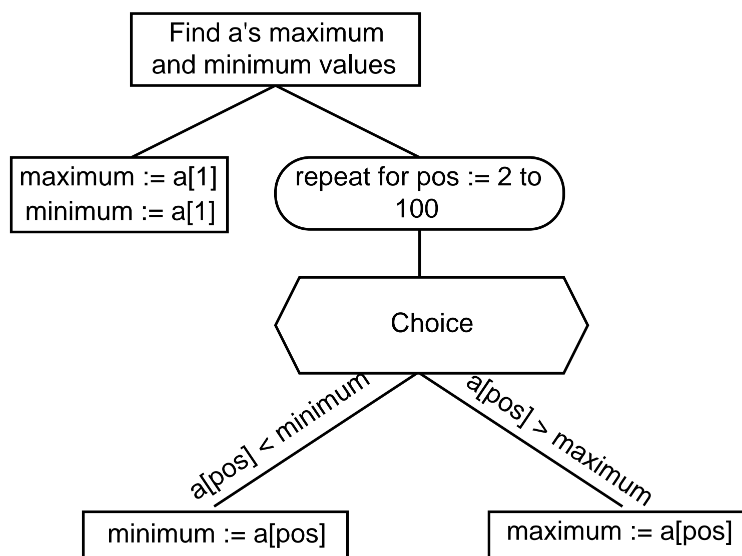


Figure 2: A Structure Diagram

This is a very simple example, but it enables us to see the benefits of the notation; both the actions performed and the conditions which control their execution are portrayed in an unmistakable hierarchy, but they have been separated so that much of the syntactic skeleton required when they are mingled in conventional programming languages - **begin**s, **end**s, **then**s **else**s and so on - is unnecessary. Only later when the diagram is translated into program text must the programmer add the skeleton, and deal with getting its syntax right.

4. HyperPascal: An Active Mapper from Idea Space to Implementation Space

Structure Diagrams provide a good basis for a programming language which incorporates a visual representation of the Action Tree. They are unable to represent declarations, and need augmenting to render them suitable for interactive capture and editing, but suitable methods for accomplishing these tasks have been designed.

HyperPascal is a program-capture-and-compile-environment - the programming language is only part of an integrated package. The name HyperPascal was chosen to reflect the hyperspatial nature of the programs produced within the environment, and

the generally Pascal-like feel of the language; it is strongly typed, and with minor exceptions, it has Pascal's data types and operators, and analogues of Pascal's control-flow constructs. However, the Hyperspatial aspects of the language are universal, and are certainly not restricted to the Pascal capabilities.

In spite of their shared characteristics, Pascal and HyperPascal programs have very little visible similarity; their resemblance is at an abstract level. Conversely, the original appearance of Structure Diagrams is retained in the new language, but their vocabulary has been considerably extended to remedy some omissions from the original notation.

In describing HyperPascal, we shall use conventional GUI terms *open, select, drag,* and so on, without reference to their implementation on any particular platform. Thus, for example, an icon could be opened by mouse-clicks when the cursor was positioned over it, or by selection of an *open* option from a menu.

4.1 A Simple Subprogram in HyperPascal

Figure 3 shows a simple iterative multiplication subprogram in HyperPascal. All the visual components of the notation, such as declarations and sub-icons (described below) have been completely expanded. The root node is the subprogram header. It shows the name of the subprogram, declarations of the formal input parameters, x and y, as a pseudo-record, and the type of the subprogram name (which is being used to return a value). For the sake of efficiency, the algorithm starts by swapping x and y if y is larger. Then it initialises the product to 0, and loops, adding x into the product and subtracting 1 from the multiplier repeatedly while the multiplier is greater than 0. There is no exit code associated with the loop.

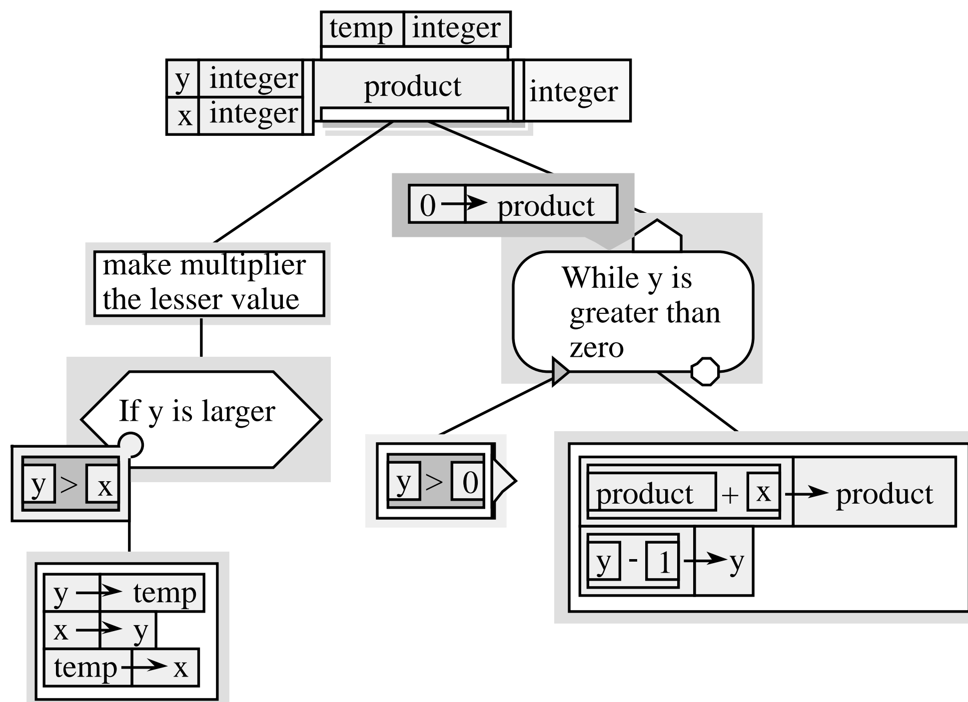


Figure 3: x*y Implemented as an Iterative Functional Subprogram in HyperPascal

Figure 4 shows a compressed view of the Scope Tree for a program which contains subprograms nested three deep. Opening any of the nodes in the tree gives the programmer access to the declarations associated with that node; opening the persistent

node gives access to declarations of items which continue to exist after the program has finished executing.

The programmer is able to transit between the Scope Tree and the Action Tree *via* a menu, but will usually not have to perform such an explicit mode-switch, being free to declare variables *en passant* when they are first used. Further, program name and procedure name icons act as a window from Scope Tree space to Action Tree space. We shall therefore postpone further discussion of the Scope Tree until we have dealt with the Action Tree.

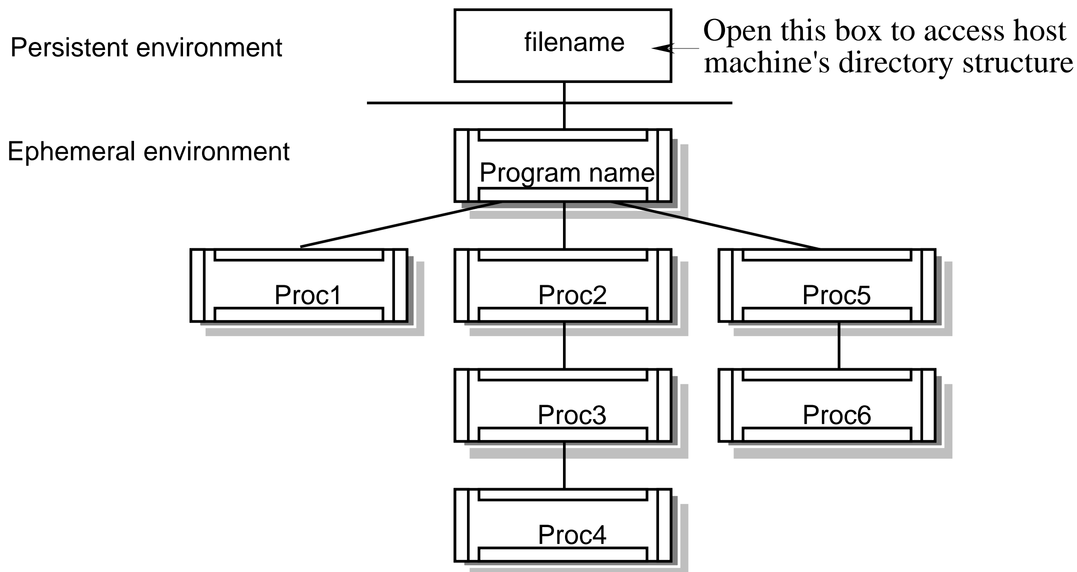
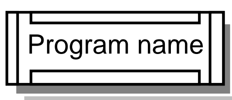


Figure 4: **A Scope Tree**

4.2 An Informal Tour of the Components of the Action Tree
The Action Tree is logically a single structure, but is split into a number of subtrees, one for the main program, and one for each of the subprograms. Each of these may be edited in a separate window. More than one window may be visible at a time.



The root icon of each Action subtree contains the name of the program or subprogram, two grey shadows, which are links to the corresponding part of the Scope Tree and the State Tree, and four wide *collapse bars* which can be opened (see Figure 5) to reveal declarations of the program's or subprogram's:

input files (left bar of program subtree) or
input parameters (left bar of subprogram subtree),
identifiers (top bar), and
output files (right bar of program subtree) or
output parameters (right bar of subprogram subtree)
invocations (bottom bar).

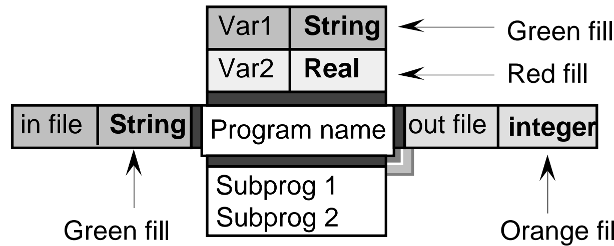


Figure 5:[2] The program icon with its file declaration, identifier declaration and subprogram invocation bars opened

This example illustrates several general features of the notation:

wide *collapse bars* (vertical or horizontal) can be opened to reveal information relating to the item they are attached to, and possibly to allow it to be edited;

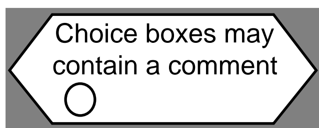
all predefined types are associated with a colour. The type colour always appears in any field which returns a value of that type;

collapse bars (and the other sub-icons which we will encounter later) can be opened. Even in their collapsed state, they provide some information about their contents; they are white when they are empty, they are the type-colour of the item they contain when this has a predefined type, they are grey when they contain a user-defined type; otherwise, they are the colour of the background of the box to which they will expand;

input and output parameters of a subprogram, and input and output files of a program are shown as pseudo-records. That is, they are listed vertically with a grey bar running down the side of the grouping, and associating them with the pseudo-record name (the subprogram or program name). This is consistent with the representation of ordinary records, which have a similar grey bar allowing them to be associated with their record name (see below)

opening a grey shadow behind a program component reveals an alternative view of the current icon, showing the Scope Tree or the State Tree, depending on the shadow chosen.

Comment boxes contain an editable comment space and act as junction nodes in the Action Tree. They specify no actions, and exercise no authority over the program's flow of control.

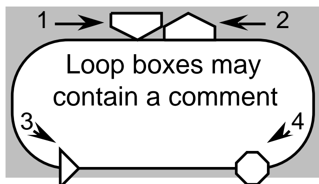


Choice boxes are used to specify which of two or more subtrees in the Action Tree to execute. They initially contain a single disc-shaped sub-icon. Opening this reveals an edit box containing a Boolean condition which, if true, allows execution of the code in the subtree attached to the disc.

The single sub-icon in the choice box can be duplicated repeatedly (*via* standard menu or "power-key" operations), and each of the resulting sub-icons can be opened to give access to an editable Boolean Expression field (else is a valid Boolean Expression here,

[2] In this monochrome diagram, as in others in this paper, different densities of grey fill are used to indicate different item types.

and is automatically inserted - but editable - as the Boolean Expression in the right-most of the added sub-icons), thus allowing the choice box to act as an IF-THEN, an IF-THEN-ELSE, or a case statement with Boolean expressions instead of conventional case labels, depending on the number of choice discs.



Loops are controlled by the code contained in the loop icon's sub-icons. The programmer can open these to insert code to be executed (1) prior to entry into the loop, (2) on exit from the loop, and to determine whether the loop is (3) to continue executing or (4) to terminate. The subtrees attached to the loop icon, which are not restricted in number, are executed in left-to-right order, and the stop and go sub-icons may be dragged to a different position in the sequence of subtrees, so that the test they perform can occur before or after some relevant processing.

For example, the icon in Figure 6 specifies a loop in which the code in three subtrees is executed each time the loop is executed, except in the iteration when the stop sign's Boolean expression, tested after the first two subtrees have been executed, becomes true. In that case, the loop is terminated immediately, and the third subtree is not executed. Of course, if, in a particular iteration, the Boolean expression in the go sign becomes false, none of the sub-trees' code is executed during that iteration.

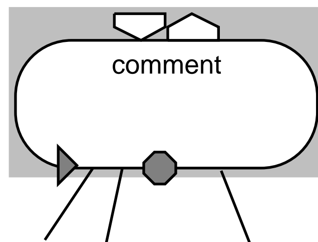
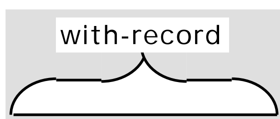
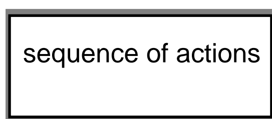


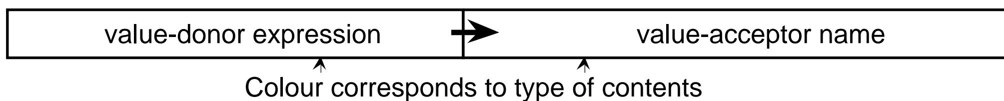
Figure 6: The stop condition may prevent execution of the third subtree in this loop



A With icon contains a reference to a record structure in the field at its top. It has the same function as a standard Pascal with statement: to allow a programmer to reference the fields of a record without specifying the parent record name. All the subtrees attached to the with-record icon are affected. Similar rules to those used in Pascal about resolution of ambiguity apply.



It will surprise no-one to discover that Action Sequences specify sequences of actions. There is only one type of action; assigning values (usually only one) to a storage location (which may include a file). Figure 7 shows an assignment icon's two fields. The right field identifies the destination of the value to be assigned; the left contains an expression capable of generating that value[3].



---

[3] In HyperPascal, we adopt the view that assignments are analogous to information flow - which, for English speakers at least, seems most naturally represented by a left-to-right notation - rather than to mathematical equations, the ancestors of conventional assignment statements like a :=b.

Figure 7: General format of an assignment icon

If the donor and acceptor fields are not type-matched, their type-colours retract from the field boundary. When they are type-matched, the type-colours extend up to the boundary, and an assignment arrow appears, pointing from the left to the right field to indicate the direction of movement of the result of the expression.

4.3 In-line Declarations

When an identifier is entered, it is compared with contents of the Symbol Table, and if it is not found, a type menu is displayed. The user then declares it by selecting a type from the menu. There may then be sub-dialogues for specification of upper and lower bounds (for an array - see Figure 8) or fields (for a record) and so on.

Note that the user may choose to continue specifying actions, leaving the declaration till later. The menu will then disappear, but, until the identifier is declared, it will be shown with a spectral background wherever it appears. Thus undeclared variables may occur in the language, but they are rendered conspicuous.

Simultaneously with the type menu, the Scope Tree is displayed, and clicking on a location in the Scope Tree associates the variable with that location. Identifiers may also be declared when the solution space is viewed through the Scope Tree window.

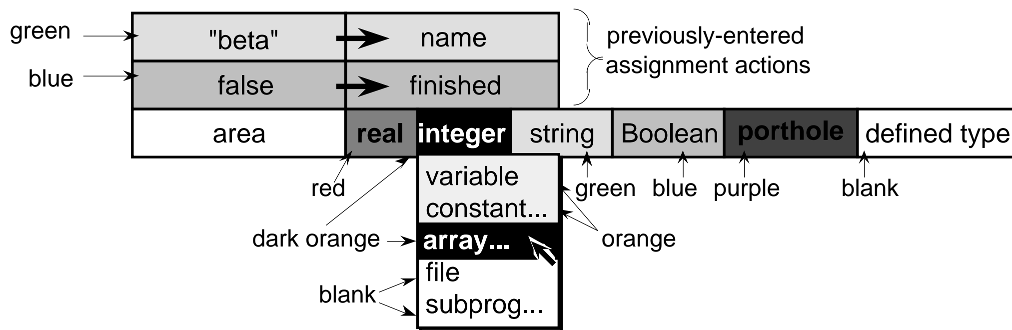


Figure 8: Typing an array variable when it first occurs in an action sequence

The variable/constant/array/file/subprogram selection is available for all appropriate types; defined types can be selected (see Figure 9) from sets, records, enumerated lists and user-defined types.

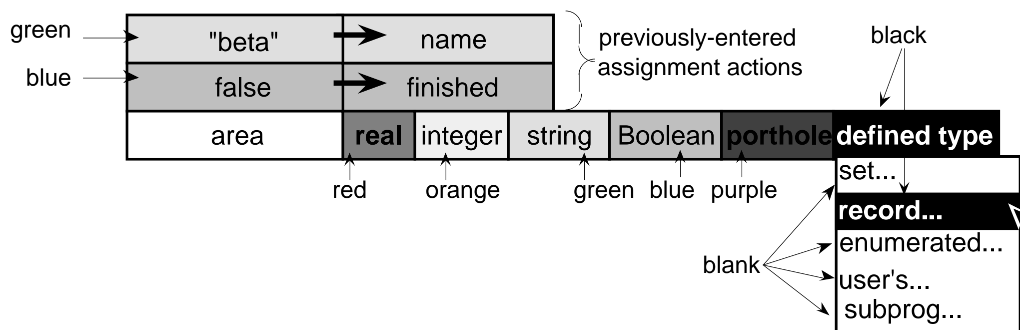


Figure 9: Typing a complex identifier

4.4 Complex Data Structures

When a record name or a pointer name is used, it is shown with, respectively, a collapse bar along the bottom of its field or a porthole in its field. Double-clicking on this sub-

icon opens it to reveal the fields of the associated record (see Figure 10). Note that the bar associated with an open record or porthole is situated at the right if the item is a value donor, and at the left if it is a value acceptor. When the item is a value donor, the bar is not just a mnemonic icon, but acts as a **with** group, enclosing by default a set of fields corresponding to the item's fields, but capable of being expanded to allow the effect of the **with** to apply to more assignment actions.

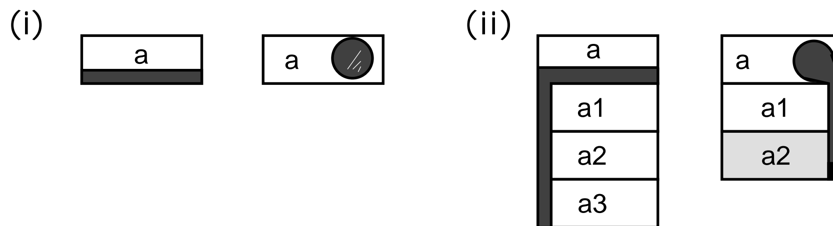
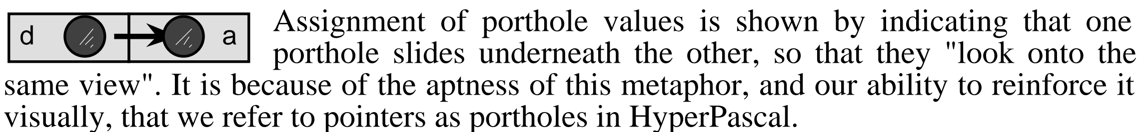


Figure 10: Record and porthole icons (i) unopened and (ii) opened.

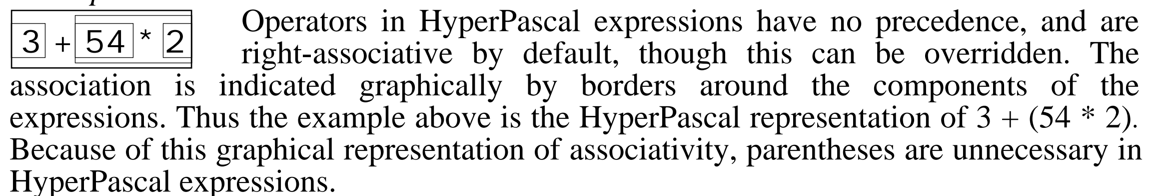


Assignment of porthole values is shown by indicating that one porthole slides underneath the other, so that they "look onto the same view". It is because of the aptness of this metaphor, and our ability to reinforce it visually, that we refer to pointers as portholes in HyperPascal.

4.5 Variations on the Assignment Action

Figure 11 shows a variety of different assignments which have been formed by various open and drag edit actions, all designed to organise a donor item, d, and an identically-typed acceptor item, a, into the adjacent left and right fields of an assignment action. When two adjacent items have different types, the type-colours in their fields retract so that type mismatches are readily apparent (even to colour-blind programmers). In addition, when a correctly-typed pair is produced, an assignment arrow appears between them.

4.6 Expressions



Operators in HyperPascal expressions have no precedence, and are right-associative by default, though this can be overridden. The association is indicated graphically by borders around the components of the expressions. Thus the example above is the HyperPascal representation of 3 + (54 * 2). Because of this graphical representation of associativity, parentheses are unnecessary in HyperPascal expressions.

4.7 Subprograms

HyperPascal has generic subprograms rather than separate functions and procedures. The subprograms can have side-effects (i.e. they can modify global variables), and they can transform input parameters into output parameters. Their names can behave as simple variables, returning a single value (like a function), or as the name of a pseudo-record of output variables, returning a set of values (like a procedure). A subprogram's input parameters are always regarded as a pseudo-record associated with its name. Depending on whether it has 0, 1, or more input and output parameters, and whether it is being used functionally or procedurally, a subprogram invocation takes one of several forms.

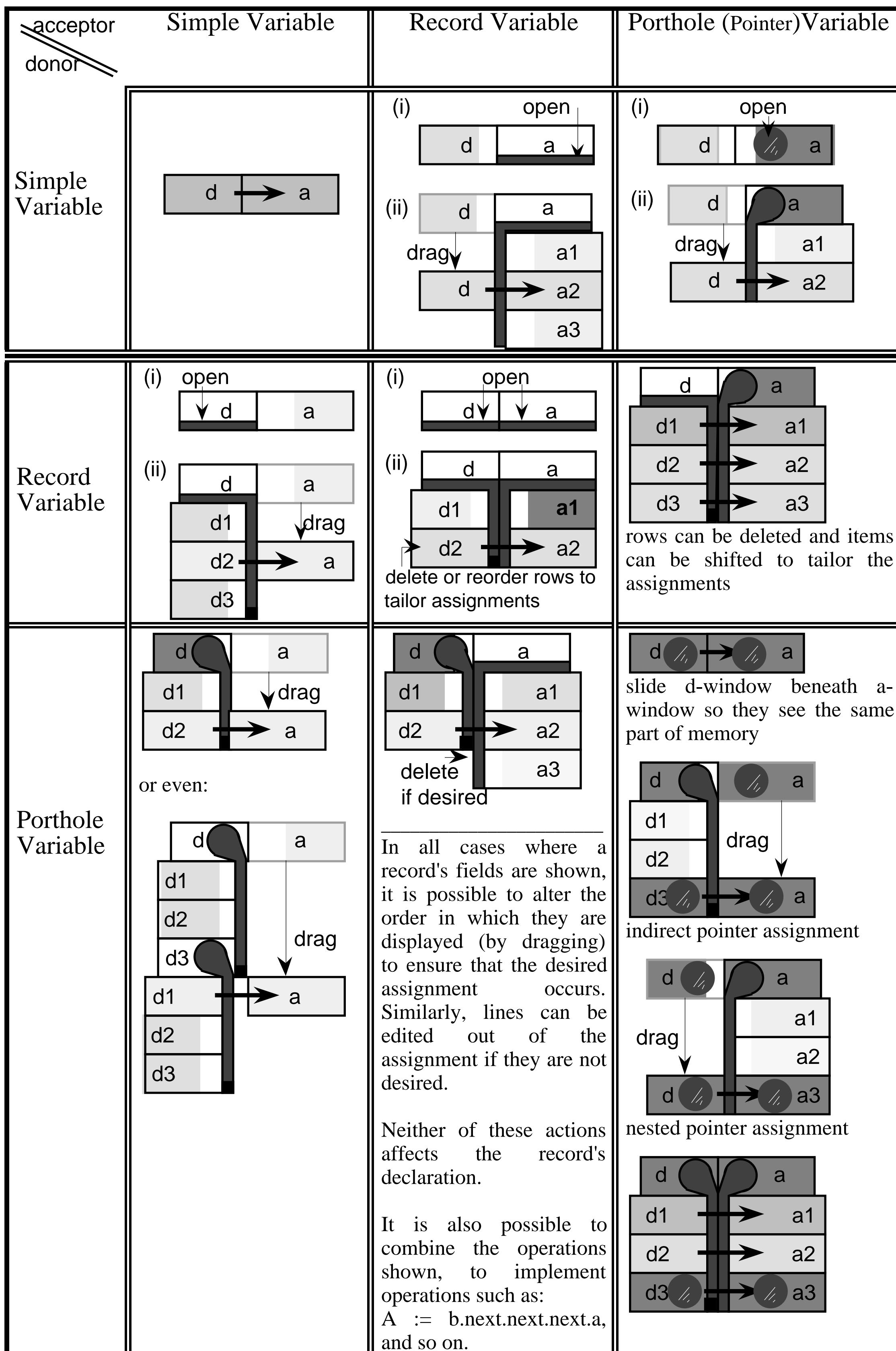


Figure 11: A wide variety of assignments is possible in HyperPascal

A subprogram invocation (see Figure 12(i)) has an abbreviated form in which the subprogram's name is preceded and followed by collapse bars, which can be opened to reveal the subprogram's input and output parameter associations. These are modelled on standard HyperPascal assignments, and show both the actual and the formal parameters. It is also possible to open the horizontal collapse bars at the top and bottom of the invocation to show the subprogram's local variable declarations and any subprogram invocations which it performs.

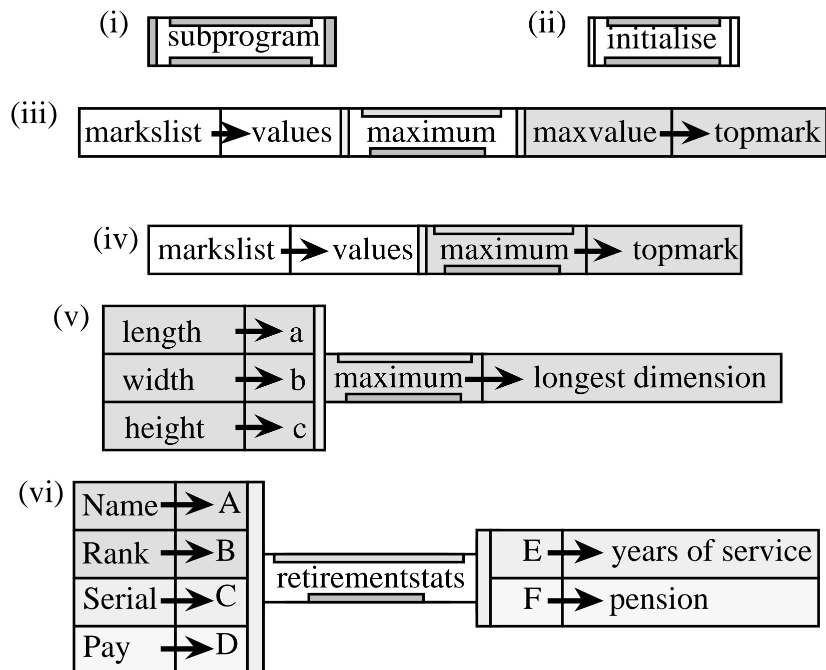


Figure 12: Subprogram invocations in procedural mode

"Initialise" subprograms, which assign initial values to a program's global variables, are frequently called with neither input nor output parameters; in such a case the collapse bars are shown empty (see Figure 12(ii)). The subprogram invocation in Figure 12(iii) has a single input parameter and a single output parameter. Its actual input parameter is shown being assigned to its formal input parameter, and its formal output parameter is shown being assigned to its actual output parameter. It is also permissible for the subprogram name to be used to return a value (see Figure 12(iv)), if there are no other output parameters. Figure 12(v) again shows a subprogram name being used to return a value. This time, however, it is shown with three formal input parameters, which are represented as a pseudo-record being assigned the values of the actual input parameters. When a subprogram accepts multiple input parameters, and produces multiple output parameters, both its input and output lists appear in the Action Sequence as pseudo-records (see Figure 12(vi)).

Thus far, our discussion of subprograms has shown them as the sources of values for simple assignments. They can also produce values to be used in expressions. As the input parameters are represented in the same way in expressions as in simple assignments, we shall not discuss them further here, but will concentrate on the output parameters.

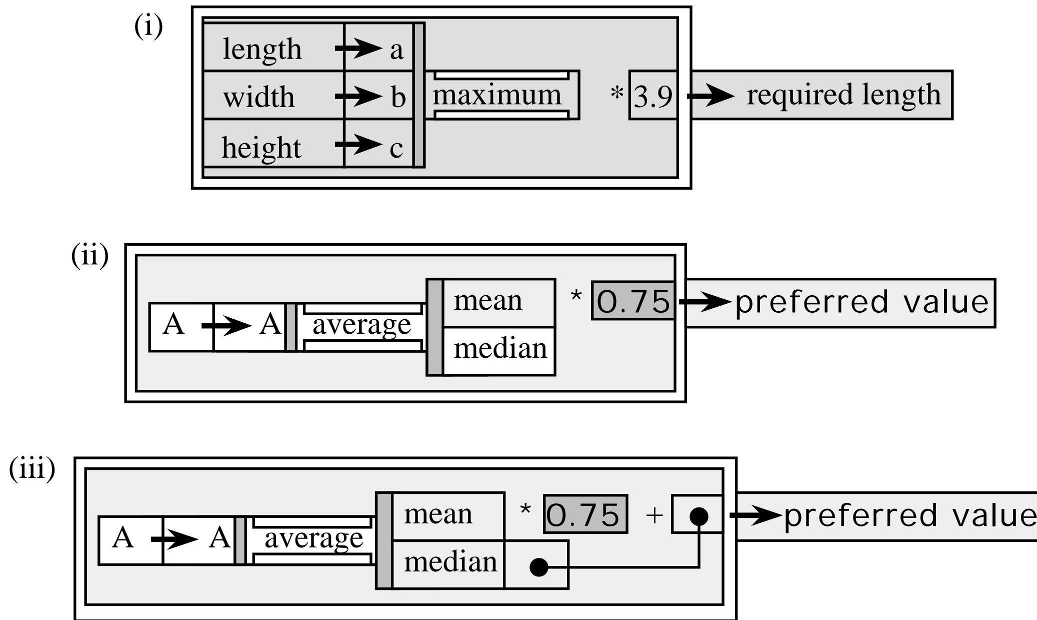


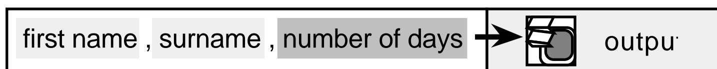
Figure 13: Subprogram invocations in functional mode

In the assignment shown in Figure 13(i), **maximum** returns a value in its name, which is then further processed (multiplied by 3.9) before the desired result is assigned to the variable **required length**.

Figure 13(ii) shows a subprogram which returns two values, but only one is used in the expression in which the subprogram appears. The programmer makes the choice by dragging the invocation to align the field to be used with the operator which is using it. An output parameter field's type colour appears when it aligns with an operator for which it is the correct type, so subprograms can be used in this fashion on the right hand side of operators as well as on the left.

In Figure 13(iii), the value **mean**, returned by the **average** subprogram is used immediately in the expression, whereas the value **median** is connected by a line to the spot where it is used later in the same expression.

4.8 I/O



HyperPascal incorporates I/O operations into its standard assignment action. Input is effected by listing a filename in the donor field. The filename is preceded by a format sub-icon which can be opened to reveal the format specification of the input data. Output is effected by listing a filename in the acceptor field. Again, the format sub-icon can be opened to show the format of the file.

5 Conclusions

5.1 Support for Novices

As HyperPascal has yet to be tested, it is too soon to speculate about its ultimate success. However, some insight into whether it addresses the right problems is provided in Fix, Wiedenbeck, and Scholtz's (1993) paper "Mental Representations of programs by Novices and Experts". They identify a number of features of users' representations of programs characteristic of "mature mental representations in programming". These are

areas in which novices and experts differ significantly in their understanding of programs. The importance of their classification to our work is that they identify problems which developers of visual languages should address in order to provide an environment which will assist users to develop their model of the program. The areas in which the most significant differences exist between novices and expert programmers should be supported by information presented appropriately by the program capture environment. In Figure 15, we list the tasks which Fix *et al*'s subjects attempted. The tasks are ordered so that those for which the difference in achievement between expert and novice was most significant are first. The right column shows the type of support provided by HyperPascal in each of the task areas.

Tasks at which experts perform significantly better than novices	**HyperPascal support mechanism**
Match procedure names to the procedures they call.	Action and Scope Trees allow programmers to check this.
List names used for same data objects in different program units.	Subprogram invocations shows actual/formal parameter correspondences.
Fill in names of program units with a skeleton outline of the program.	Scope Tree shows outline at subroutine level; Action Tree allows abbreviation or expansion of action subtrees, to represent a skeleton outline.
Match variable name to the procedures in which they occur.	Scope Tree search can locate any identifier, and subprogram invocations shows actual/formal parameter correspondences.
Write descriptions of goals of selected procedures.	Explicit provision for comments at all levels in the Action Tree.
Label Complex code segments with plan label (i.e. recognise stereotypical situations).	HyperPascal's proposed dynamic structure traversal icon extends the language beyond the standard traversal paradigms in a powerful but general way.

Figure 15: HyperPascal support for areas where novices have trouble

5.2 A Comparison of HyperPascal with other Visual Languages

Shu (1986) and Chang (1987) have analysed Visual Languages in a three-dimensional framework. They say that, to determine the applicability of a language to a certain type of task, it should be measured in terms of three independent parameters:

its visibility (its usage of visual metaphors);

its language level (its algorithmic "high-level-ness");

its language scope (the extent of its universe of discourse).

It is tempting to infer that it is desirable for a language to achieve a high ranking on all of these scales. However, as Graf (1990) 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 seems inadvisable to replace these notations *in toto* with visual analogues,

and preferable merely to support and augment them, particularly where their power has been constrained by typographical restrictions.

Notwithstanding these *caveat*s we present, in Figure 16, a Shu Chart classifying HyperPascal in the context of some Visual Programming languages which Shu has previously classified and of some which have not previously been classified on such a system (Pearson, Apperley and Lyons (1991) and Ngan (1991)). Our, necessarily subjective, ranking of HyperPascal places it comparatively high on the scales representing adequacy in representing processes and objects, and at a medium height on the visibility scale.

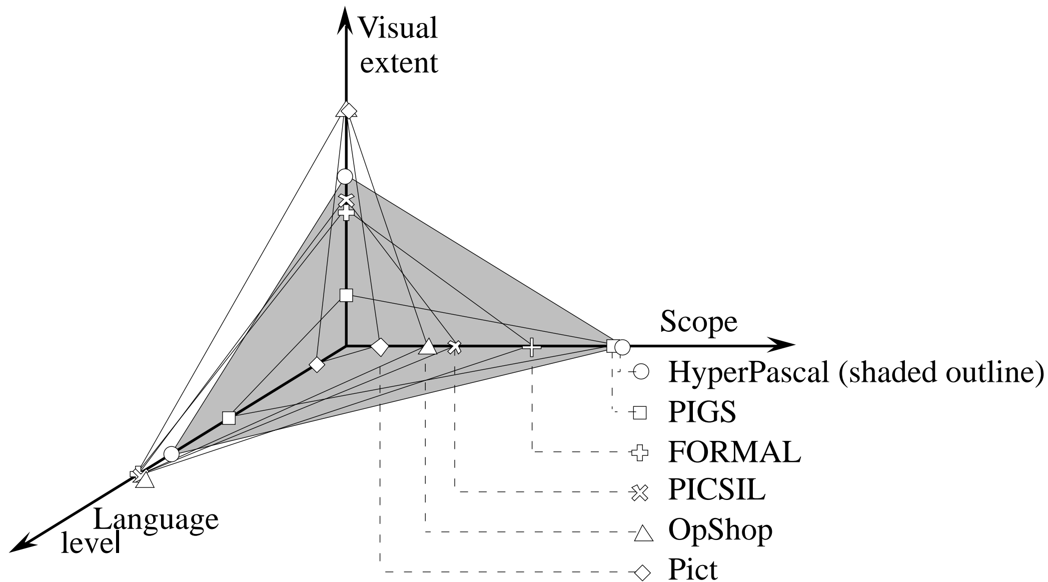


Figure 16: Shu Chart characterising HyperPascal and other visual languages in terms of their Scope, Language Level, and Visual Extent

5.3 Space Required for HyperPascal

When we started developing HyperPascal we were concerned that programs in the notation might occupy very large amounts of screen space. However, we found that common prettyprinting conventions consume many lines of listing in matching **begin**s and **end**s, **then**s and **else**s etc. Further, alternative views of the program, such as declarations and I/O formatting, hidden in HyperPascal's Scope Tree, the Format Window, and so on, occupy space in a sequential listing at all times. Consequently the Action Tree view of HyperPascal Programs turn out to be little larger, and sometimes slightly smaller, than conventional textually-based programs, and provides windows through hyperspace to information contained in other views of the solution.

References

Chang, S.K., 1987: "Visual Languages: A tutorial Survey", *IEEE Software,* **4**, 1, 23 - 29

Doran, B., and Tate, G, 1972a: An Approach to Structured Programming, Part I, *Massey University Department of Computer Science Publication no 6.*

Doran, B., and Tate, G, 1972b: An Approach to Structured Programming, Part II, *Massey University Department of Computer Science Publication no 9*

Fix, V., Wiedenbeck, S., and Scholtz, J., 1993: Mental Representations of Programs by Novices and Experts, *INTERCHI '93.*

Graf, M., 1990: Visual Programming and Visual Languages: Lessons learned in the Trenches, in *Visual Programming Environments: Applications and Issues*, ed. Ephraim P. Glinert, 452-455

Ngan, P.M., 1991: OpShop: an iconic programming system for image processing, *Proceedings of the Australasian Apple University Consortium Conference, ANU.*

Pearson, M.W., Apperley, M.D., and Lyons. P.J., 1991: A Data Flow Approach to Silicon Compilation, *NELCON Proceedings* ***28,*** 168 - 173.

Raeder, G., 1985: A Survey of Current Graphical Programming Techniques, *Computer*, **18**, 8, 11-25.

Reiss, S.P., 1985: PECAN: Program Development Systems that Support Multiple Views, *IEEE TRANS Softw. Eng,* **SE-11**, 3, 1985, 276-285

Shu, N.C., 1986: Visual Programming Languages: A Perspective and a Dimensional Analysis, in *Visual Languages*, ed. Shi-Kuo Chang, Plenum Publishing, NY 1986.

Shu, N.C., 1988: *Visual Programming* p48, Van Nostrand Reinhold Co. Inc., New York.

Williams, C.C., and Rasure, J.R., 1990: A Visual Language for Image Processing, *1990 IEEE Computer Society Workshop on Visual Languages.*