HYPER STRUCTURES + VISUAL PROGRAMS

=

HYPERPROGRAMS

A thesis presented in partial fulfillment of
the requirements for the degree of Master
of Science in Computer Science at
Massey University

by

Craig Robert Simmons
March 1994

Preface

This development described herein - the first implementation of a hyperprogramming language - is related to, but far more wide-reaching than my original masterate project, which was to develop an editor for structure diagrams. Early in the course of the research, one of my supervisors, Paul Lyons, invented the hyperprogramming concept and designed a preliminary version of HyperPascal. In view of the fact that this original "flash of inspiration"was not mine, and the subsequent close cooperation during language development between me and my supervisors, it has proved difficult to determine the true originator of various ideas. Furthermore, as there are no earlier detailed descriptions of the principles of hyperprogramming, they must be included in this document. Therefore, in order to avoid claiming undue credit for the linguistic developments, I have described language design decisions using first person plural ('we").

However, converting these design decisions into an implementation of HyperPascal was solely my work.

Abstract

This thesis describes an investigation into the integration of hyper-techniques with visual programming languages to support a multi-dimensional, minimally syntactic program representation.

Programming involves two phases: first, forming a mental model of the problem solution; secondly, mapping the mental model onto a physical representation. The mental model is complex, syntax-free and multi-dimensional; in textual programming languages, the physical representation is complex, syntax-rich and single-dimensional. Performing the mapping is painstaking work which has more to do with easing compilation than with representing data manipulations.

It is believed that a physical representation which better matches the programmer's mental model will significantly reduce the difficulty of generating programs.

Modern computer systems combine powerful processors, and large memories with high-resolution graphics and powerful graphic input mechanisms. This ideally fits them for supporting the building and interpretation of complex multi-dimensional structures with minimal syntax.

The Hyperprogramming paradigm exploits this capability. A hyperprogramming language uses different visual representations for different program dimensions - for example different visual vocabularies are appropriate for algorithms and subroutine nesting. Each *view* is carefully chosen to overlap the others minimally, and where overlap is essential, hyperlinks between views are provided to allow easy navigation between them, and to allow automatic updating of shared information.

HyperPascal was developed using this philosophy, as a testbed for it. In creating a program, a HyperPascal programmer edits information in three separate views:

> the *action window* view, in which subroutines are each represented using a visual language based on structure diagrams
>
> the *scope window* view, in which declarations are stored in a nested structure corresponding to conventional subroutine nesting
>
> the *forms window*, in which the appearance of I/O can be designed using a WYSIWYG editor, free of the distractions of data processing specifications.

A protoype of HyperPascal has been implemented, and a number of programs developed using it.

Acknowledgements

I would like to thank Professor Mark Apperley and Paul Lyons for their suggestions and encouragement throughout the period of this research. In particular, I would like to thank Paul Lyons for the extensive proof-reading undertaken, and guidance given in the production of this thesis.

Thanks also go to my friends; those at Massey, and those outside, for supporting me in my time at Massey and providing a welcome distraction from research when needed. Particular thanks must go to Tracey Murrell for understanding the occasional intrusion of work in my private life, and seeing me through the high and low points of my post-graduate study.

John Grundy, John Hosking, and the Department of Computer Science at Auckland University are thanked for allowing the use of the MViews framework.

The financial assistance of Massey University is also gratefully acknowledged.

My biggest thank you goes to my family; my sister Debbie, and especially my parents Bonnie, and Robert for the support and love they have always provided.

This thesis is dedicated to my family.

Contents

Chapter 1

Representing Programs

There have been many developments in human-computer interaction in the forty-four years since ENIAC was first unveiled. Plugs and switches have been replaced as input media by punched cards. Early direct control computers have been successively replaced by batch-mode mainframes, interactive single-user and multi-user mainframes, stand-alone micro's and networked systems. Output devices have changed from memory-mapped rows of lights through line printers to high resolution colour screens and laser printers. One development in particular has influenced the style of modern interfaces more than all the others. This is the direct manipulation graphical user interface, which combines mouse input, high resolution graphics monitor and multiple-window data presentation.

Most applications produced today make use extensive use of graphical user interfaces and direct manipulation. Many applications also allow users to input information visually, or produce graphical output. This trend towards graphical user interfaces and direct manipulation has, however, progressed more slowly in some areas than in others. An area exhibiting this slow progression is that of computer programming.

In the early stages of computer development, programmers were the only direct users of computers. As computers became more widespread and easier to use, another class of user, the end user, started to appear. These relatively naive users became the driving force behind human-computer interaction, and new interaction styles were developed to make applications more "user-friendly". Interaction styles in the programming process were largely ignored at this stage because programmers were considered experts at using computers, and therefore did not need them to be "user-friendly". Development in the programming process instead concentrated on increasing the power of high level languages, in order to increase productivity and satisfy the user demand.

Of course, this concentration was not total. Early work on syntax-directed editors, for example GED (Moretti and Lyons, 1986), has been incorporated into present program development systems. Most such systems perform enough parsing-on-the-fly to provide comparatively simple aids such as pretty-printing, detection of malformed expressions, and so on. However, the underlying nature of the languages they deal with is unchanged. The languages still use a single-dimensional stream of text to represent complex algorithms and data-structures. Perhaps this should not surprise us. Linearity pervades the computing milieu: we store data in sequentially addressed memory; we squeeze data through the ALU to process it. Little wonder, then, that our compilers are sequence-oriented, and that the languages they deal with are too.

However, it need no be so. We don't always design sequentially - on the contrary, in many fields, it is common practice to initiate a design with informal two-dimensional diagrams and it would be attractive to be able to capture designs in this form. Hitherto, the undoubtedly greater effort of capturing a visual programming language than a

conventional textual language has mitigated against their use. Now however, high-resolution graphics, processor power and memory are now available cheaply enough to render two-dimensional visual programming languages viable.

What is so attractive about a visual language that we are willing to expend this extra effort? There are several benefits:

- Relationships are two-dimensional
  Relationships between components of a complex systems are rarely simple enough to be expressed in a simple linear fashion; more often we find that responsibilities, communication, and division of tasks require a network if they are to be properly described.

- Diagrams can use mnemonic shapes
  At least in systems with simple vocabularies, the shapes of diagrammatic components can be imbued with mnemonic significance - arrows to represent direction of flow are a ubiquitous example of this - and the significance is assimilated subconsciously.

- Partitioning is natural in diagrams
  A two dimensional surface gives a designer more freedom to partition a complex system, and to use physical proximity or separation to indicate similarity or difference between its components.

- Vocabularies can be easily distinguished
  Most complex systems have to be specified in more than one domain - programs, for example, involve algorithms and data structures. It is possible to design a visual language to emphasise the similarities and differences between the different domains. Thus, by associating a distinctive window-type with the current domain, for example, the user can be subliminally cued to employ the correct vocabulary for the current part of the design.

These benefits have been incorporated into a number of Visual Programming Languages (Shu, 1986), some of which will be examined in more detail later. For now, let it suffice that most of these languages are ways of visualising *programs*. This is an inherently flawed approach to visual programming, as it works backwards from a naturally sequential representation to a two- (or more) dimensional representation. The present work has proceeded from the basis that the object of the visualisation should be, not the program, but the programmer's mental model of the solution to a problem.

The idea of a programmer's mental model merits a little further explanation. We may describe a program as the eventual physical representation of a mental conception comprising a group of separate, but related, specifications of aspects of the solution. One such specification would be the data-processing operations which the program will have to perform. Another would be the appearance of the information input and output by the program. Clearly these two are linked - processing of information cannot begin until it has been input. However, they are also largely independent - formatting I/O is at most loosely linked to the program's data processing activities. Mapping them onto a sequential representation must at least compromise their structure, if not conceal it altogether.

1.1. Project Aim

This work aims to test the validity of this idea of using a multi-dimensional physical representation to represent the programmer's mental model. It combines techniques from the Visual Programming Languages programming paradigm and the hypermedia navigation support systems. The result has been the design of a *hyperprogramming* language HyperPascal, and the implementation of a prototype integrated program development environment for HyperPascal.

Note that there has been no attempt to introduce new programming functionality with HyperPascal, and that even the diagrams are not particularly revolutionary. The originality of the work is associated with the multi-dimensionality of the representation, and the use of hyper-techniques for navigating through the multi-dimensional space.

1.2. Statement of Thesis

We can summarise the above arguments thus:

In writing a computer program, a programmer devotes much effort to developing a mapping from a complex multi-dimensional *mental* model of the problem solution onto a *physical* representation which is sufficiently simple and syntactically rigorous for a computer to interpret. It is possible to use modern interface technology and processor speeds to support a closer match between these two representations than is available in conventional programming languages, both textual and visual. An appropriate representation would allow:

- a multi-dimensional representation of programs.
- multiple views with minimal interaction (storage of identical information in more than one view).
- where interaction is unavoidable, editing of the shared information *via* a single mechanism, identical in the different views.
- easy transitions between views.
- automatic updating of information shared between different views.
- subliminal clues to the nature of the current view.
- maximum support for navigation within the representation.

1.3. Project History

In this description, the progress of the research has been split into separate phases. In the research proper, these phases were not strictly separate, but are described this way to better show an overview of the research.

Since the aim of the project is use a visual language to provide a representation of a program that is closer to the programmer's mental model, previous research was

reviewed from the areas of visual programming, human-computer interaction, software engineering, and software comprehension. A number of visual programming languages were also compared. This review identified several areas where visual programming languages were deficient in the representation of a programmer's mental model.

The field of hypermedia was identified as representing documents in a multi-dimensional manner, and techniques from the field were adapted for use in representing computer programs (a program represented multi-dimensionally is termed a *hyperprogram*).

A general purpose visual programming language, HyperPascal (Hyper - *hyper*program, Pascal - based on Pascal semantics), that used a multi-dimensional representation, was designed as a "testbed" language to test the validity of hyperprogramming ideas. A number of different views onto the program structure were identified, and first-draft notations developed for each view. During this phase the notation was most fluid: various alternative formulations were experimented with, and a consistent notation finally emerged.

To support both novice and expert programmer interaction, a number of different interaction techniques were incorporated into the design of HyperPascal, and the programming components modified to better utilise these techniques, while also ensuring that they remained consistent.

A subsidiary goal was to make HyperPascal simple for novices to learn. To facilitate this, the syntactic load was reduced by reducing the number of different programming components. This reduction in the number of component types was performed by generalising similar components into a single, specialisable, consistent component. The generalisation of the programming components also allowed a compaction of the notation, and provided generalised components that are powerful enough for use by expert programmers.

A prototype of the language was developed so that we could test the multi-dimensional representation of, and navigation within programs in HyperPascal. This implementation has concentrated mainly on the major components of the language, and their ability to specify an executable program. The prototype language enables simple hyperprograms to be translated to Pascal source code for later compilation and execution, and a number of hyperprograms have been developed and executed using this prototype.

Chapter 2

Related Research

2.1. Visual Programming

Visual Programming is a very general term which is used to describe different ways to use graphics in programming. Shu (1986) divides the area into two general sub-areas, *Visual Environments* and *Visual Languages*.

Visual Environments

Shu defines Visual Environments as "...the incorporation of graphical techniques in a software environment that supports the program or system development.", and splits this broad area into three categories, *Visualisation of Data or Information, Visualisation of Program and Execution,* and *Visualisation of System Design.*

A Visual Environment can be placed in the Visualisation of Data or Information category if it presents data or information in a graphical form. These environmets are most often used to support program debugging. Amethyst (Myers, Chandhok, and Sereen, 1988) is an integrated environment which allows the editing of Pascal programs, dynamic visualisation of selected data structures in the program, and allows the user to trace through the execution of the program to observe changes in the data. This allows users to "see" what is happening to the data in the program, and helps them locate errors. Figure 2.1 (after Myers *et al.*, 1988) shows the appearance of the Amethyst screen while viewing the values of x, y, and sum.

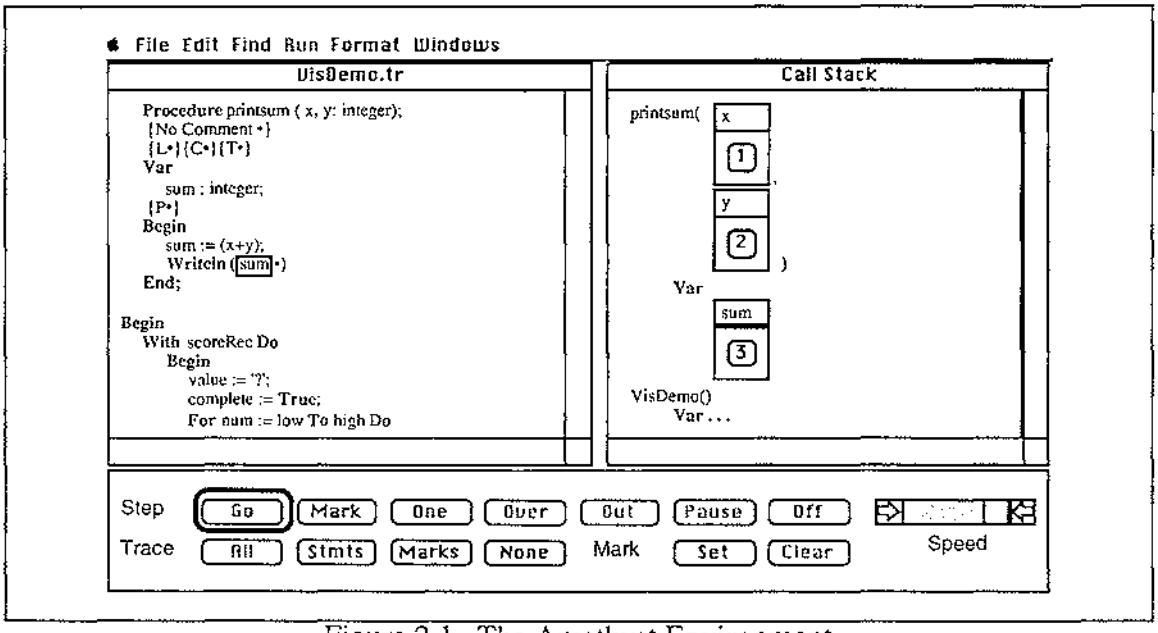


Figure 2.1. The Amethyst Environment

A Visual Environment can be placed in the Visualisation of Program and Execution category if it allows static and/or dynamic (run-time) visualisation of a program. This visualisation of a program can help the user to better comprehend the structure of a program. An example of an environment which allows both static and dynamic visualisation of a program is PECAN (Reiss,1985). PECAN allows a program to be visualised statically as a structured flow graph, and edited using a syntax-directed editor. Dynamic program visualisation is provided by highlighting the statement currently executing in both the structured flow graph, and the syntax directed editor. PECAN can also be placed in the Visualisation of Data or Information category because the data structures present in a program can be presented in a graphical form. Figure 2.2 shows the flowchart representation of a program in Pecan, with the currently executing node hilighted.

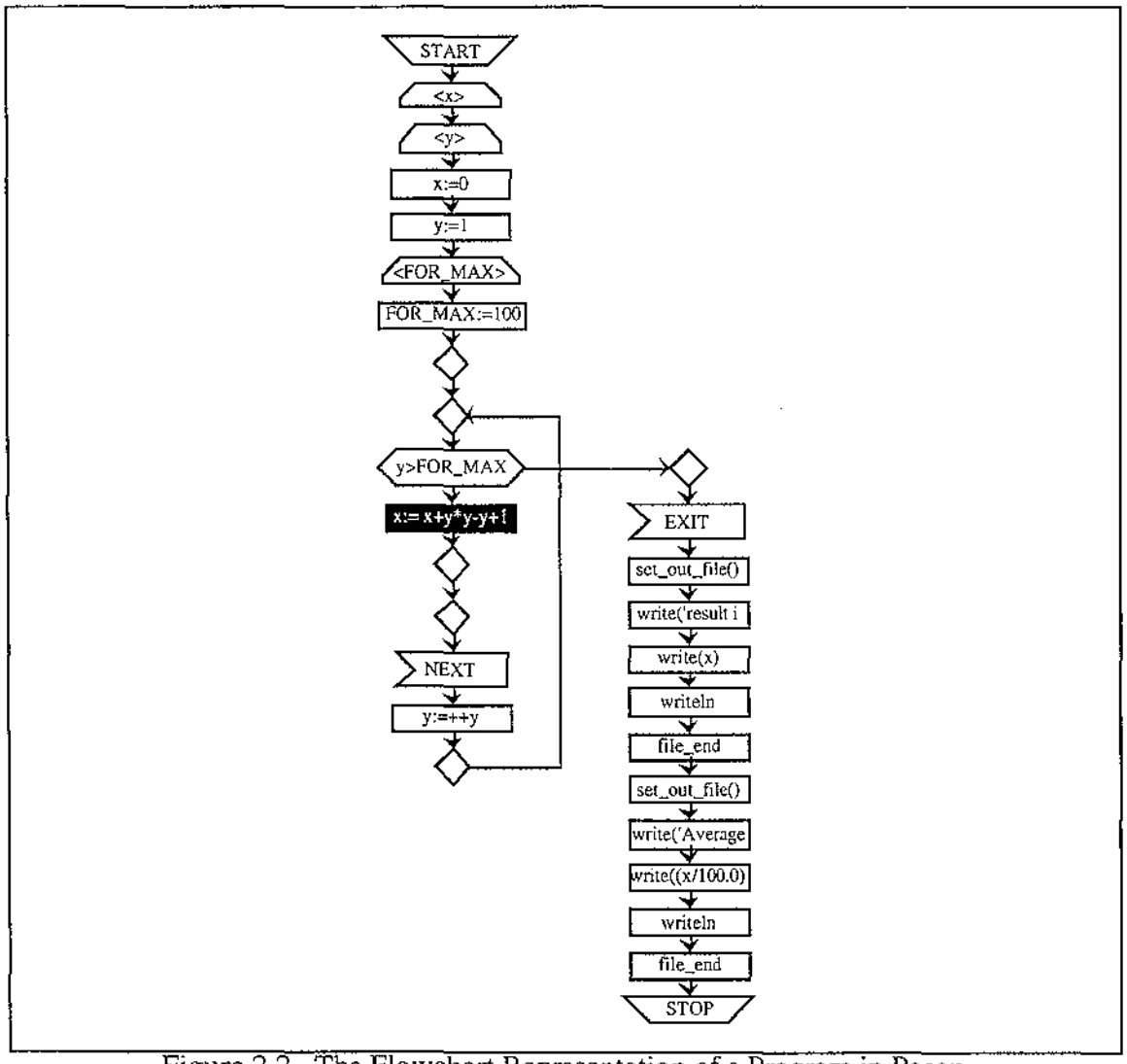


Figure 2.2. The Flowchart Representation of a Program in Pecan

Visualisation of System Design is the approach used to visualise an abstraction of the overall system design. The visualisation of an overall system can help the user to locate errors or flaws in the system (such as a module not being utilised). Conic (Kramer, 1989) is an example of an environment which allows programming and visualisation of high level system configurations. The Conic environment permits the specification of distributed processor systems, and "can both describe the structure of the required system and specify the allocation of components to machines". The visualisation of the system

configuration is not as expressive as the (textual) specification language, and doesn't show the physical location of modules, or the module parameters. Figure 2.3. shows a Conic system configuration for a hospital ward monitoring system.

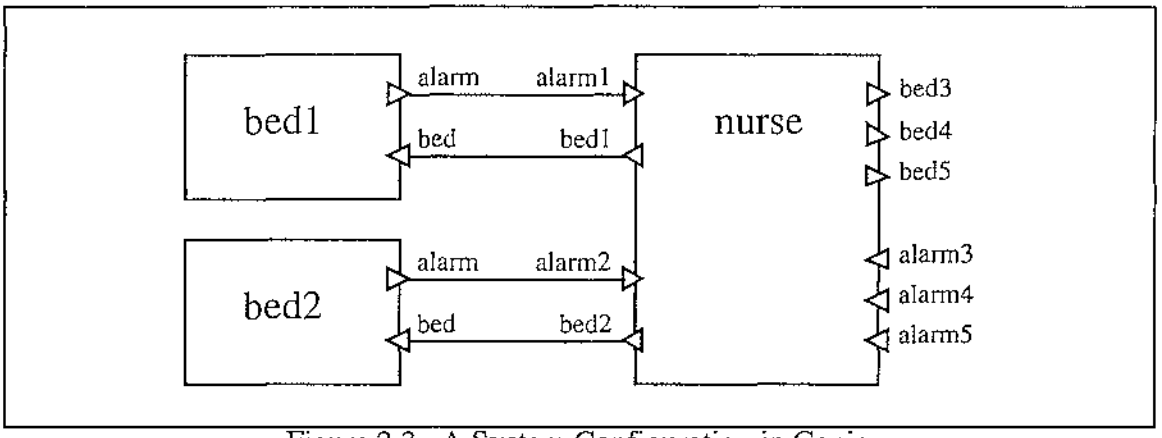


Figure 2.3. A System Configuration in Conic

Figure 2.4 categorises the three visual environments already mentioned, as well as Zeus (Brown,1991), Balsa-II (Brown,1988), Incense (Myers,1983), PV (Brown, Carling, Herot, Kramlich, and Souza, 1985), TPM (Eisenstadt and Brayshaw,1990), and C^2 (Kopache and Glinert,1988). These environments are now described very briefly.

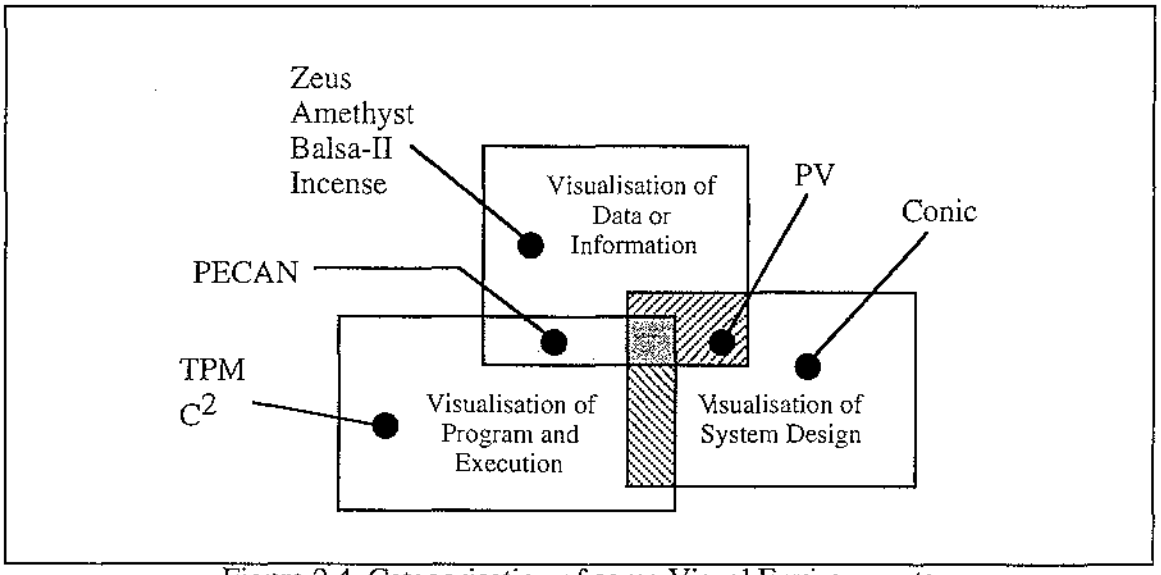


Figure 2.4 Categorisation of some Visual Environments

Incense is designed to visualise data structures in a Pascal-like language called *Mesa*, and allows the data structures to be viewed while debugging the program.

Zeus and Balsa-II are similar systems for visualising algorithms through graphical animation. Through their use of multiple views, they can visualise (by animation) many algorithms simultaneously, thus facilitating comparisons between the methods and the speeds of the competing algorithms.

PV is a system that allows the visualisation of both the high level modular structure of a system, and (by simple animation) the data structures and algorithms.

TPM is a system for visualising the dynamic execution of Prolog programs, and is used mainly for assistance in the teaching process. TPM allows a Prolog program to be

stepped through, and also allows *fast-forwarding* to a particular clause. The notation used is an augmented And/Or tree.

C^2 is a system which allows the graphical visualisation and (limited) manipulation of programs written in a subset of the C language. C^2 represents programs with a *BLOX*-type notation (Glinert, 1990), in which program components fit together like pieces of a jigsaw puzzle, but C^2 does not store the graphical representation between runs.

We now turn from the more general category of visual programming environments, in which the emphasis is on the use of direct manipulation and visualisation to support program development, to the category of visual languages, in which the emphasis is on manipulating visual information, or programming using visual expression.

Visual Languages

Shu (1986)defines three categories of visual language, those for *supporting visual interaction*, those for *programming with visual expressions*, and those for *processing visual information*. Chang (1987) generalises this classification scheme by positioning languages in a two-dimensional space, which allows him to divide the third of these classifications, languages for processing visual information, into *visual information processing languages*, and *iconic visual information processing languages*.

The first dimension of the classification space, which Chang calls *visibility*, is the type of syntax used for specifying the transformations of data objects dealt with by the language. A syntax may be *linear* (data transformations are strung together in one dimension) or *visual* (data transformations may be interconnected in some more complex arrangement).

The second dimension of the space defines the type of object, which Chang describes as a *generalised icon*, manipulated by the language. Generalised icons have two component parts, a meaning and a visual representation, and languages are classified either as (meaning, image') - these manipulate objects that have a logical meaning and an imposed visual representation, or as (meaning', image) - these impose a meaning on naturally visual information[1].

An example of a (meaning, image') generalised icon is the folder icon in the Macintosh user interface. The folder icon corresponds to a logical item in the directory structure of the file system, and has the  image imposed. An example of a (meaning', image) generalised icon could be a physical image such as  (possibly in an image database) having the logical meaning *face* imposed on it.

The classification of visual languages in Chang's two dimensional space is shown in figure 2.5.

---

[1]Chang's rather confusing notation uses (Xm,Xi) to represent a generalised icon, (Xm,e) for an icon with a logical meaning but no image, (e,Xi) for an icon with an image but no meaning, (Xm,X'i) for an icon with a logical meaning and an imposed image, and (X'm,Xi) for an icon with an image and an imposed logical meaning. Chang then describes languages in terms of the mapping they perform on these ordered pairs (a language that manipulates objects with a logical meaning, and an imposed image is described by the mapping (Xm,e) -> (Xm,X'i)).

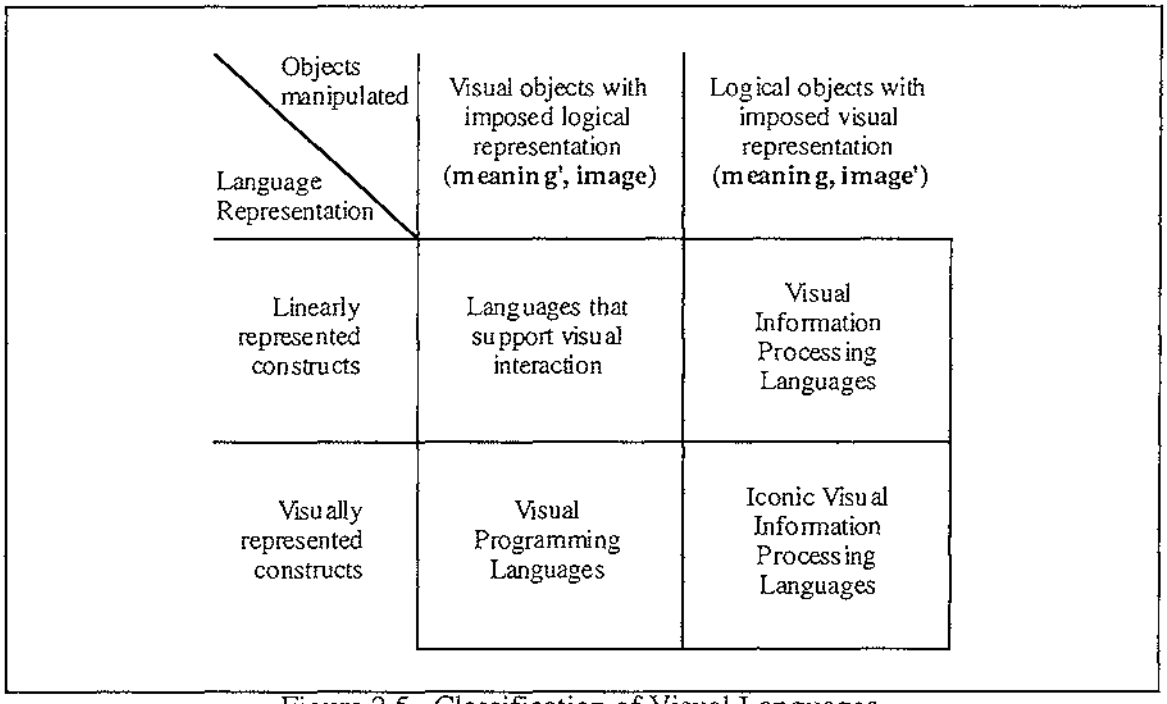


Figure 2.5. Classification of Visual Languages

Languages that support visual interaction (linear construct, (meaning, image')) are linear (usually textual) languages that support visual interaction with logical objects. An example of a language that supports visual interaction is DAL (Anderson **get reference**), which supports the textual specification of dialogs and interaction in the user interface.

Visual programming languages (visual construct, (meaning, image')) are considered in detail in Section 2.2.

Visual information processing languages (linear construct, (meaning', image)) are languages that manipulate naturally visual information using linear (usually textual) constructs. Examples of visual information processing languages include augmented (for querying pictorial databases) conventional query languages such as ISQL (Assman, Venema, Höhne, 1986) and PSQL (Roussopoulos, Leifker, 1984), and image processing languages such as VIPS (Bailey and Hodgson, 1988).

Iconic visual information processing languages (visual construct, (meaning', image)) are languages that manipulate naturally visual information using visual constructs. Examples of this type of language include HI-VISUAL'86 (Hirikawa, Monden, Yoshimoto, Tanaka, and Ichikawa, 1986), OpShop (Ngan, 1991), Cantata (Williams and Rashure, 1990), and VPL (Lau-Kee, Billyard, Faichney, Kozato, Otto, Smith, Wilkinson, 1991).

We now discuss visual programming languages (visual construct, (meaning, image')) in detail.

2.2. Visual Programming Languages

A *Visual Programming Language* is defined by Chang (1987) as a language that deals with logical objects with an imposed visual representation, and uses visually represented programming constructs to manipulate these objects. Shu (1986) states that visually represented constructs must be meaningful (in representing program functionality), not

merely decorative. Myers(1986) is of the emphatic opinion that the representation of programs in a visual programming language must not be restricted to a single dimension.

Research into visual programming and visual programming languages has focused on more powerful ways of programming, learning to program, and facilitating end-user programming.

The area of visual programming languages has been subdivided differently by different authors.

Shu(1988) recognises three classes of visual programming language(s); *Diagrammatic systems* combine existing diagram notations and executable code to make programs easier to comprehend, document and maintain. *Iconic systems,* where icons or graphical symbols play a central role in programming[2]. *Table- and form-based* systems use a fill-in form, or table, to specify programs.

Shu's classification of visual programming languages into three areas based on the extent of (or approach to using) graphics in a language, is complicated by the fact that some languages such as FPL (Cunniff, Taylor, and Black, 1986) and Prograph (Cox, Giles, and Pietrzykowski, 1989) are on the boundary between diagrammatic systems and iconic systems. An arbitrary decision therefore has to be made as to their classification (Shu classifies FPL (based on flowcharts) as diagrammatic, and Prograph (based on dataflow diagrams) as iconic).

Ambler and Burnett (1989) subdivide the area of visual programming languages into two categories; *Visually Transformed Languages* emphasise the specification and comprehension of programs using existing language paradigms. *Naturally Visual Languages* are an attempt to develop new programming paradigms[3] whose natural expression is visual, and which may not have a strictly textual equivalent. An example of a programming paradigm whose expression is naturally visual is *programming-by-demonstration*, where the user's actions when demonstrating a process are recorded and mapped directly to a program. ThinkPad (Rubin, Golin, Reiss, 1985) is an example of a language that uses programming-by-demonstration to produce data structure manipulation programs in Prolog.

We assert that it is more appropriate to divide the area of visual programming languages on the basis of the area of application of the language. This would enable languages designed for a similar purpose to be placed in the same class. We identify two broad areas for the classification of visual programming languages; *Domain-specific* visual programming languages are languages designed specifically for use in one area of application. *General-purpose* visual programming languages are languages designed to for use in a wide application area.

A breakdown of the areas of visual programming, visual languages, and visual programming languages is shown in figure 2.6.

---

[2]Korfhage and Korfhage(1986) use a more formal definition of Iconic Languages based on a four layered language model. The first layer of this model is the iconography (the set of possible icons), and the second is the syntax for combining these icons into statements and commands. The last two layers are functional mappings, with the third layer being the semantics of the statements and commands, and the fourth being a pragmatic layer that determines whether a particular command or statement makes sense in a given environment.

[3]Discussions continue amongst researchers (Dudley and Mahling,1991) as to whether or not Visual Programming is itself a new programming paradigm.

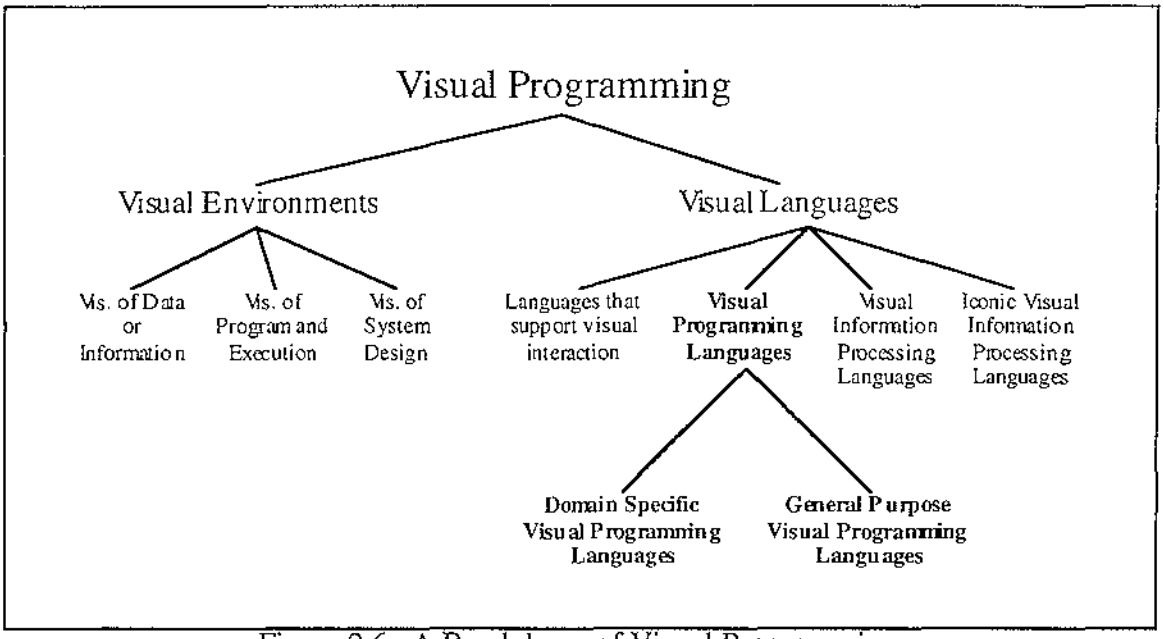


Figure 2.6. A Breakdown of Visual Programming

Arguments For and Against Visual Programming.

Shu(1988) lists four premises as stimulating the use of graphical representations and pictures in the programming process. These premises are: pictures aid understanding and remembering; pictures provide an incentive for learning to program; well designed pictures are understood by people regardless of what language they speak; and pictures are more powerful than words as a means of communication.

Shu's first premise, that pictures aid understanding and remembering, is justified by research into cognitive psychology (Rohr(1986) and Scanlan(1989)), which has shown the use of *appropriate* pictures and diagrams can aid both the understanding, and recall, of information.

The second premise, that pictures provide an incentive for learning to program has been demonstrated in an empirical study performed by Glinert and Tanimoto (1984), using the Pict visual programming system. Glinert introduced 65 computer science students to the system, had them attempt to write a number of simple programs, then fill out a questionnaire. A summary of the results showed that a large majority of the participants found the system "fun to use" as compared to normal (textual) programming, and that a majority of students would react favourably if asked to participate in another Pict session.

The premise that pictures can be understood regardless of language is supported by the wide, and successful, use of pictures for providing information in situations where language-independent communication is needed (Wood, Wood, 1990). Examples of pictures used in these situations (such as in airports, and on road signs) are shown in Figure 2.7.

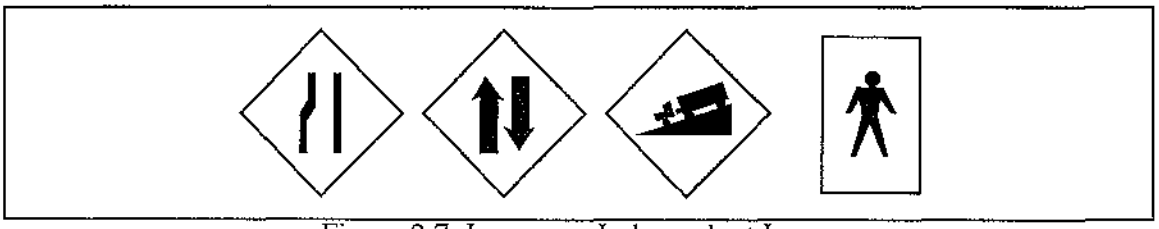


Figure 2.7 Language Independent Icons

When explaining complex ideas, people frequently use diagrams or pictures to aid communication. This use of pictures for representing complex information would imply that pictures are a powerful means of communication, and the old saying "a picture is worth a thousand words" would bear this out. Raeder (1985) identifies 8 ways in which pictures are superior to text as a means of communication:

1/ Random vs. sequential access.
The eyes provide instant access to any part of a picture.

2/ Dimensions of expression.
Pictures provide many more dimensions in which to encode information.

3/ Transfer rate.
Pictures generally transfer information faster than text.

4/ Concrete vs. abstract.
Pictures of real-world objects can be used to illustrate abstract ideas.

5/ Pictures without names.
Pictures can be referenced by pointing (Direct manipulation).

6/ Real-world pictures.
Pictures reflect the real world, whereas text only points to the real world.

7/ Animated Pictures.
Pictures are dynamic and can be animated, whereas text is static.

8/ Metaphorically rich pictures.
Information which is undesirable can be encoded to appear unpleasant.

We assert that a ninth point may be added to this list:

9/ Visual Discrimination.
Pictures aid discrimination between objects and relationships.

Other authors have argued against the use of visual programming.

Brooks (1987) tells us that "Nothing even convincing, much less exciting has yet emerged from such efforts [research into visual programming] ... I am persuaded that nothing will". Although Brooks centres his arguments mainly around the use of flowcharts as a visual programming notation, he (and others) have raised a number of general arguments against visual programming.

Software is invisible and, because of its structural complexity, it is also inherently unvisualisable (Brooks, 1987). Brooks also states that this invisibility is undesirable because it hinders understanding and communication of information about software. In this argument, Brooks is talking about the *visualisation* of existing software systems (using paper-based techiques), not the use of visual programming to implement future systems. Visual programming languages can be used to enforce structure on a software system, and because their representation is visual, the resulting software structures will be inherently *visualisable*.

Visual programming languages, while they can be useful in teaching novices to program, are of no use for computer professionals (Ichikawa and Hirikawa, 1990). It is generally agreed that visual programming can be easier to learn and understand than textual programming. This can be helpful in teaching novices to program (Glinert and

Tanimoto, 1984), and facilitating end-user computing (Hirikawa *et al*, 1986). The challenge is to make them useful to people such as computer professionals, who are comfortable with traditional methods of programming.

The "computer professionals" argument has two parts. First, it suggests that today's computer professionals won't accept visual languages, because, being easier to use, they remove much of the mystery of the programers craft. For many people, this argument may be correct, though it says more about human nature than it does about visual programming languages.

Secondly, it suggests that visual languages will never be as good as textual languages for the sorts of programs computer professionals write. Certainly, there are areas in which textual representations are superior to visual ones (arithmetic expressions, for example). Even today, however, there are areas in which visual representations are superior to textual ones, such as the design of programs for massively parallel, or distributed architectures. Computer professionals may feel most comfortable with languages that combine visual and textual representations.

Visual programming languages use screen space inefficiently (Edel,1988). This is often the case, but visual programming is not *always* less efficient in the use of screen space (Ichikawa and Hirikawa (1990), and Lyons, Simmons, and Apperley (1993)). Screen use by a visual programming language is often a trade-off between the space used, and program comprehensibility. Visual languages do have an advantage when using multiple views, because they allow different parts of a program (that could be widely separated in a textual representation) to be viewed at the same time. Visual languages also allow the placement of conceptually related constructs close to each other.

Comparison between Visual Programming Languages

Traditional (textual) programming languages can be compared using a measure of the level of the language, and the scope of applicability of the language.

The level of a language is a measure of the amount of detail the programmer must provide in order for a task to be completed. For instance, if, to achieve a particular goal, more detailed information must be provided using language A than necessary using language B, then language A is of a lower level than language B.

The scope of applicability of a language refers to how wide a range of applications the language can produce. A language such as C is very general in the types of application it can produce, whereas a language such as Cobol is more specific in its application area (Cobol can not, for example, easily perform complex mathematical manipulations). The C programming language therefore has a wider scope of applicability than Cobol.

Shu (1986) compares visual programming languages by adding the dimension of *extent of visual expression* to these two dimensions of language comparison. *Visual expression* is defined by Shu as "the meaningful visual representations (for example, icons, graphs, diagrams, pictures) used as language components to achieve the purpose of programming".

Figure 2.8(a) (after Shu (1986)) shows the three aspects of a visual programming language (level, scope, and extent of visual expression) as the axes of a three dimensional space. The relative values of a visual programming language for these three aspects are then plotted on the axes, and the points joined to form a surface as shown in Figure 2.8(b). This surface is the *profile* of the language. A number of visual programming systems are compared in section 2.3 and section 2.4 using Shu's profiling approach.

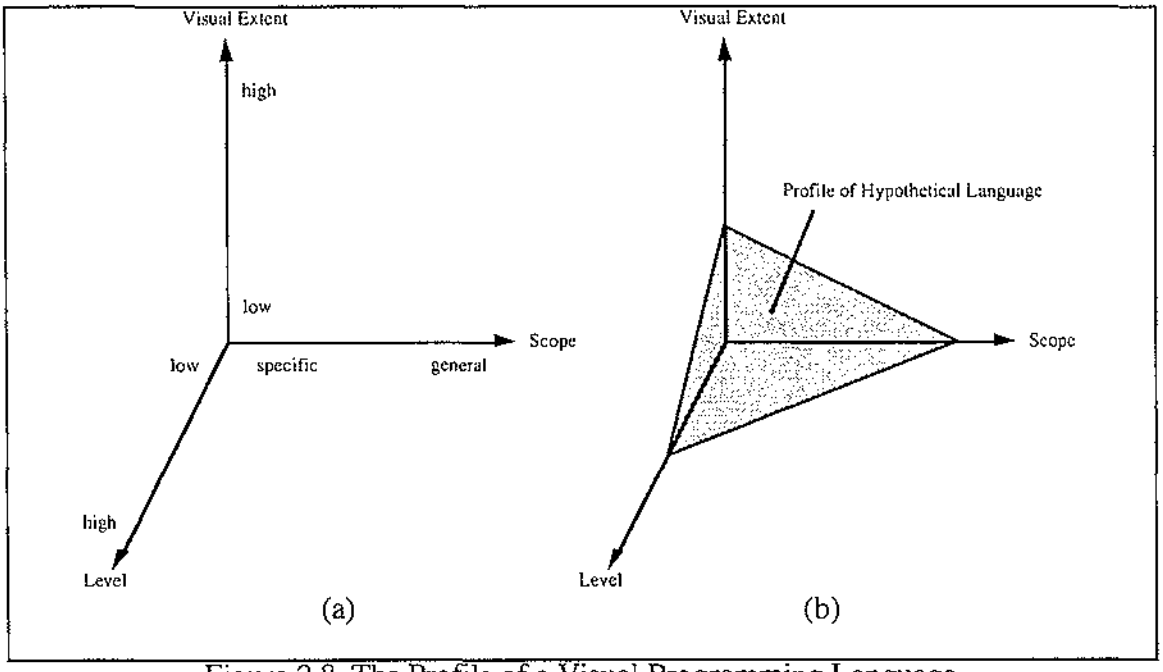


Figure 2.8 The Profile of a Visual Programming Language

While a wide scope and a high level are generally considered desirable for programming languages, Shu (1986) states that a high visual extent is not necessarily desirable for a visual programming language, and that *effectiveness of visual expression* could be a better dimension. The reason for this is that extensive use of, or inappropriate, diagrams may confuse the user. However, the effectiveness of visual expression is an even more subjective measure than visual extent.

Glinert (1989) proposes a metric for assessing visual programming languages that considers a number of *dimensions* which have been identified as "good" attributes of visual languages and visual displays, such as the use of animation (Graf, 1990), colour (Reynolds(1987) and Travis(1991)), and sound (Gaver, 1989).

Glinert's metric assesses a visual programming environment on how accessible it is to the novice, and how appealing it is to the expert user. A visual programming environment is assessed in terms of 5 aspects; the overall system characteristics, the programming environment, the flow of data from machine to user, and the speed of performance. Each of these aspects is given a *coefficient of satisfaction* which is used to calculate the *coefficient of environmental attraction H(E)*, and the *Coefficient of environmental repulsion I(E)* for that environment. For the environment to be accessible to the novice, the coefficient of environmental attraction must be maximised, and to be appealing to the expert, the coefficient of environmental repulsion must be minimised.

The three-dimensional coordinate system proposed by Shu (1986) can be obtained from the metric proposed by Glinert (1989) as a special case where only three major aspects are considered, each consisting of a single relevant dimension. Neither of the profiling and/or assessment approaches discussed takes into account the effectiveness of visual expression of the language (as suggested by Shu (1986)).

Appropriateness of Visual Language Notations

There is a level of detail in visual programming languages at which text is more appropriate than pictures. This is illustrated by Graf (1990), who states, "As deeper levels of detail and complexity are expressed, always ask if anything good is added by a

visual presentation", and "Shift to another presentation mode (usually text) as soon as appropriate". In addition to this question of when to shift to the use of text, there is also the question of which visual notation is most effective and appropriate.

Fitter and Green (1979) suggest five principles that are requirements for a good computer language notation. These principles are: give the user *relevant* information; *restrict* the user to the use of "good" structures; *redundantly re-code* important information; clearly *represent* the underlying processes; and make the notation easily modifiable *(revisability).* These principles are now discussed in more detail.

The information encoded perceptually (visually) instead of textually should be relevant to the user. The relevance of the information encoded depends on the task being performed by the user. If a task requires an understanding of structure, then structural information should be encoded visually. Likewise, for a task requiring an understanding of sequence, sequence information would be the most relevant to encode visually.

It is suggested that the user should be restricted to the use of "good" programming constructs only. A "good" programming construct is one which allows hierarchical decomposition of a program (such as single entry - single exit components in a language designed for procedural programming). This allows the "divide and conquer" approach to be used, but usually results in an increase in the number of program components (and therefore screen space).

Redundant re-coding usually occurs where information (such as a program) is presented both in pictorial form, and textual form. The information presented in pictorial form (e.g. program structure) can be found from the symbolic form (the program text), and so is redundant. Redundant re-coding is related to the issue of relevance, where the redundantly re-coded information should be relevant to the user's task.

The underlying processes must be clearly represented by the notation in order to give the user some idea of what the program is doing (providing a *conceptual window).* This allows the user to build a conceptual model of the underlying processes, which is very beneficial when teaching novices to program (Mayer,1988).

The final principle suggested is that a program represented in the notation should be able to be changed easily. This ease of revisability allows a user to correct errors or extend the program easily.

These principles are summed up by Fitter and Green(1979) in stating that "different programs should be perceptually as different as possible". If placed in an imaginary perceptual space, the distance between programs (their perceptual difference) should be proportional to the conceptual difference between them.

Many visual programming languages have been implemented using differing notations and paradigms. These languages can be placed into two categories depending on their range of application. Those designed for a specific area of application are termed *Domain-specific*, and those designed for a general range of applicability are termed *General-purpose.*

2.3. Domain-specific Visual Programming Languages

Domain-specific visual programming languages are designed for a specific area of application, such as scientific control and data analysis, office work, image processing, and graphical user interfaces. Two examples of domain-specific visual programming languages are described, then compared using Shu's (1986) three dimensional profile.

G/LabView

LabView (Kodosky, MacCrisken, Rymar, 1991) is a visual programming language used for automating the collection of data from laboratory instruments, and for analysis of this data.

The LabView system uses the idea of *virtual instruments*. A virtual instrument is a hierarchy of modules which have interactive *front panel* interfaces, and *block diagram* programs. The definition of a simple Scale Number virtual instrument is shown in Figure 2.9(adapted from Hils (1992)).

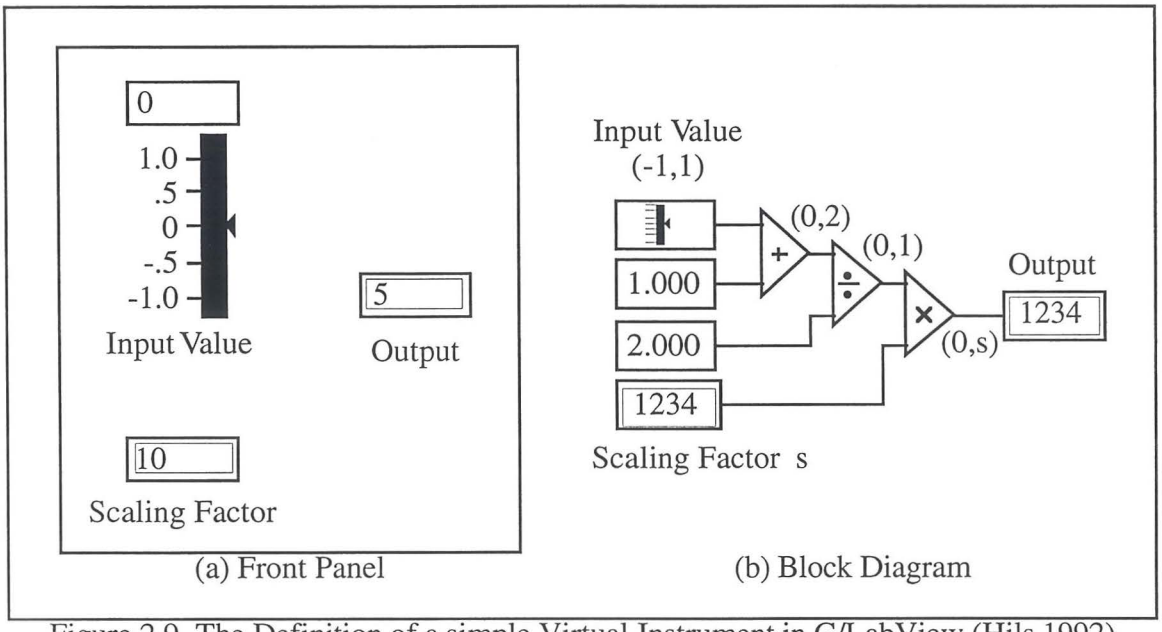


Figure 2.9 The Definition of a simple Virtual Instrument in G/LabView (Hils,1992)

Front panels (Figure 2.9(a)) can contain input and output data, sliders, knobs, meters, strip-charts, and x-y graphs. These items are used to graphically show the inputs and outputs to and from the corresponding block diagram program, and are constructed using a front panel editor. The use of front panels by LabView is a simple and effective way of specifying a user interface.

Block diagram programs (Figure 2.9(b)) are developed in LabView using a visual programming language called G. The rectangular icons in figure 2.9(b) correspond either to constants (such as 1.000), or to inputs and outputs (simple fill-in inputs such as 1234 , or to sliders such as). The triangular icons (, , and) are primitive mathematical transformations, and the bracketed numbers associated with the connecting arcs show the range of values possible on that arc.

The G visual programming language uses a notation called *Structured Dataflow Diagrams*, which are an extension to pure dataflow diagrams (Davis, Keller, 1982). Structured dataflow diagrams are traditional dataflow diagrams extended by the addition of case, sequence, while-loop, and for-loop control structures. These structures are added to facilitate the definition of algorithms that can be cumbersome and sometimes impossible to specify using pure dataflow.

The control structures in G have an outline shape which is used to contain the sub-diagram being controlled. Data is passed to and from the sub-diagrams via *tunnels*, which are arcs that pass through the enclosing shape of the control construct, connecting the sub-diagram enclosed to external nodes. Figure 2.10 (adapted from Kodosky *et al*, 1991) shows the block diagram for a frequency response virtual instrument, that uses the for-loop control construct.

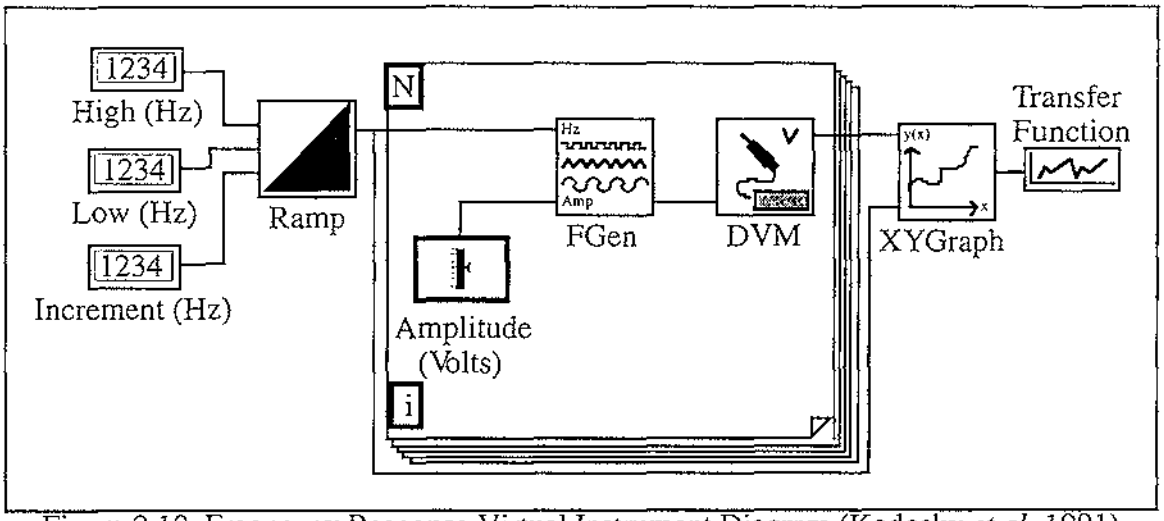


Figure 2.10 Frequency Response Virtual Instrument Diagram (Kodosky *et al*, 1991)

The for-loop shown in figure 2.10 executes the contained diagram a specific number of times. The number of times the sub-diagram is to execute is set to the number of items present in the indexed type (in this case, an array) used as input (it is also possible to set the number of iterations by placing the number on an arc connected to the N terminal). The i terminal is the source of a token whose value is a count of how many iterations have taken place. The complete set of G control structures are shown in Figure 2.11 (adapted from Kodosky *et al*, 1991).

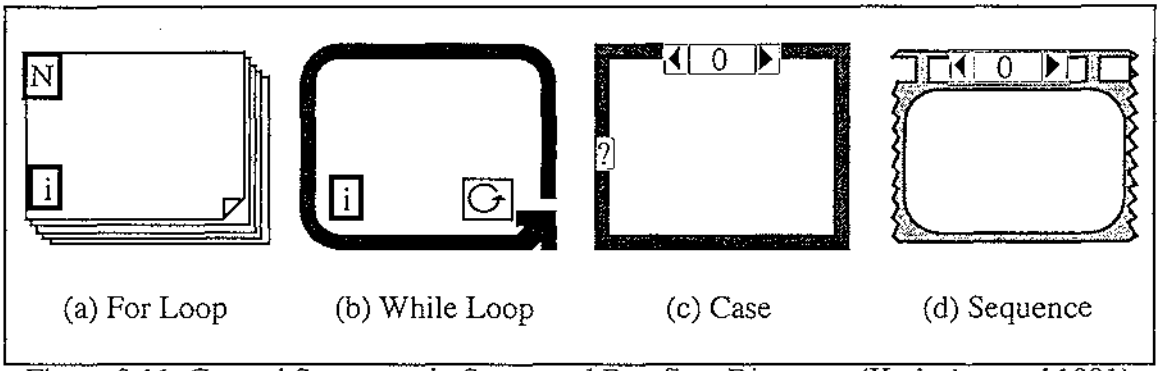


Figure 2.11 Control Structures in Structured Dataflow Diagrams (Kodosky *et al*,1991)

The while-loop (Figure 2.11(b)) executes the sub-diagram contained using the inputs currently on the input tunnels, and the sub-diagram produces a boolean token on the arc connected to the terminal. If the sub-diagram completes with the boolean token set to false, the loop terminates, and the values present on its output tunnels are propagated along their arcs. If the sub-diagram completes with a true value, the sub-diagram is executed again using the same values on the input tunnels, and overwriting the tokens on the output tunnels. The i terminal has the same function as in the while-loop.

The case (Figure 2.11(c)) contains a number of sub-diagrams overlayed on each other. Each of these sub-diagrams has a numeric label, and is visible when the viewing selector ([◀ 0 ▶]) shows the appropriate number. A particular sub-diagram is executed when all of the input tunnels have values on them, and the [?] terminal has the value of the sub-diagram's label.

The sequence (Figure 2.11(d)) also contains overlayed sub-diagrams which are viewed the same way as those contained by the case. When the input tunnels have values on them, the sub-diagrams execute sequentially, in the order specified by their labels (low to high).

G allows hierarchical decomposition of modules by the association of a virtual instrument with an icon. An icon can then be used in place of the virtual instrument definition, in the same way subroutines are used in traditional textual languages. The LabView system also includes a large library of pre-defined modules for use as terminal nodes in diagrams. The purpose of these terminal nodes range from simple mathematical operations (such as the addition, multiplication and division nodes in Figure 2.9), to modules that interface to physical instruments (such as the digital voltmeter, and frequency generator in Figure 2.10).

The G/LabView notation is concise and visually expressive when specifying the *high level* control of scientific instruments, and the analysis of the resulting data. The use of structured dataflow diagram control constructs also allows the graphical specification of algorithms that would be cumbersome (or impossible) to specify with pure dataflow. However, a problem does arise with space usage and visual complexity when the notation is used to describe even simple mathematical equations, such as the one presented in Figure 2.9(b) $\left(\frac{\text{input} + 1}{2} \times s \rightarrow \text{output} \right)$. This suggests that structured dataflow is good for representing high level abstractions of a program, but isn't appropriate for low level use such as specifying equations.

HI-VISUAL

HI-VISUAL is a high level iconic visual programming language that was originally developed for use in image processing (Hirikawa, Monden, Yoshimoto, Tanaka, Ichikawa, 1986). HI-VISUAL has since been applied to the domain of office work (Hirikawa, Tanaka, Ichikawa, 1990), and the system interface changed (Hirikawa, Yoshimi, Tanaka, Ichikawa, 1989) so that actions are performed (and recorded as a program) by overlapping icons (Hirikawa, Nishimura, Kado, Ichikawa, 1991). This system, HI-VISUAL'89, will be referred to as HI-VISUAL in the following discussion.

HI-VISUAL uses icons to represent real-world objects (such as sales books, and a calculator), and established concepts in the application domain (such as departments, and companies). It does not, however, use icons to represent functions. Functions are represented by using the hand pointer to overlap two or more icons in the HI-VISUAL interface which is shown in Figure 2.12 (Hirikawa *et al*, 1989).

The function represented by the overlapping of icons depends not only on the icons, but also on the spatial arrangement of the overlapping. An particular icon can either be *active* or *passive* depending on its role in the overlapping. If a calculator icon is moved on top of a sales book icon, the calculator icon is the active icon, and the sales book icon is the passive icon, therefore the calculator acts on the sales book. If this overlapping is interpreted as the *Produce-Totals* function, then the totals from the sales book will be

calculated, and an icon representing a list of totals will be produced as a result (as shown in Figure 2.13(a)).

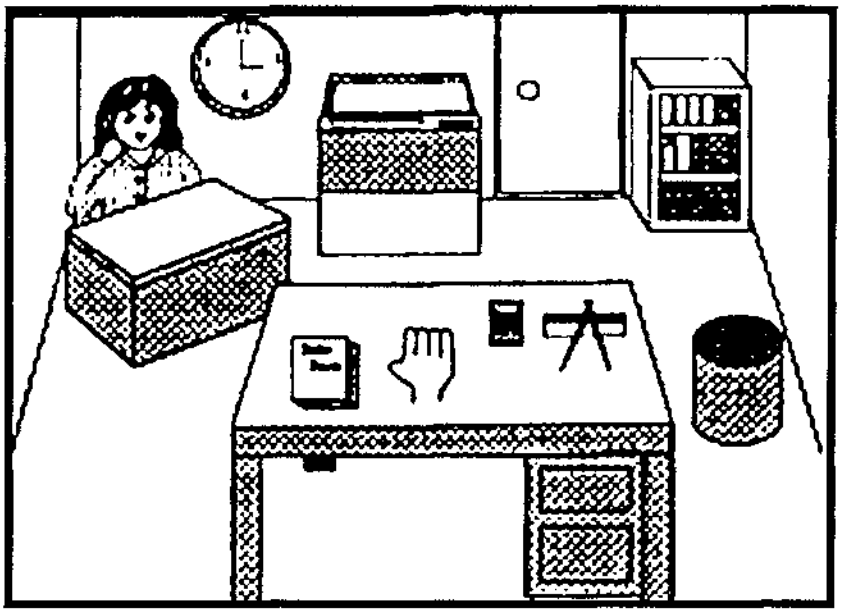


Figure 2.12 The User Interface of HI-VISUAL (Hirikawa *et al*, 1989)

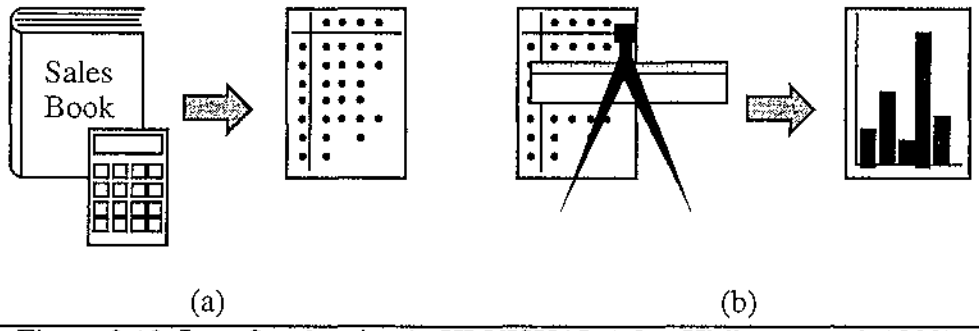


Figure 2.13 Icon Overlapping in HI-VISUAL (after Hirikawa *et al*, 1989)

It is possible for an overlapping of icons to be ambiguous. Overlapping the drawing kit icon and the totalled sales icon as shown in Figure 2.13(b) could be interpreted either as *Draw a pie chart that breaks down last months sales by product*, or *Draw a bar graph comparing total monthly sales over the last year*. In an ambiguous situation like this, the user must choose one of the possible interpretations.

When more than two icons are overlapped, two types of icon behaviour are possible. The case where more than one passive icon is present in an overlapping is termed a *plural data* type, and the case where more than one active icon is present is termed a *restriction* type.

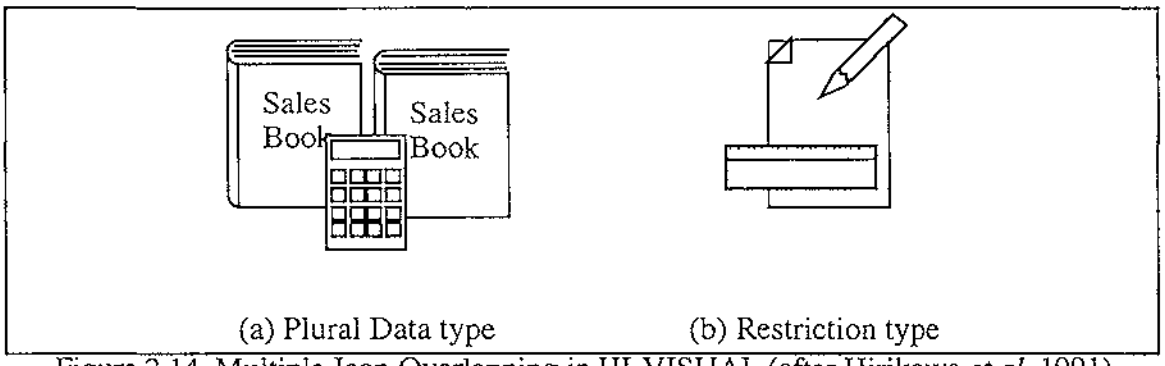


Figure 2.14 Multiple Icon Overlapping in HI-VISUAL (after Hirikawa *et al*, 1991)

In a plural data overlapping, the active icon acts on all of the passive icons overlapped. In the example shown in Figure 2.14(a), where a calculator is overlapped on two sales books, both sales books would be totalled.

A restriction overlapping is used to restrict the active icon to a limited type of behaviour. The example shown in Figure 2.14(b) shows a pencil and a ruler both overlapped on a piece of paper. Overlapping a pencil on a piece of paper selects a *draw* action, but is ambiguous because there may be many draw actions possible. Overlapping the ruler on the paper as well, places a restriction on the draw actions possible, and in this case removes the ambiguity and selects the *draw-table* action (Hirikawa *et al*, 1991).

A program can be registered with the secretary (pictured at top-left in Figure 2.12) by specifying a start scene, an end scene, icon(s) used as input, and the icon(s) to be output. This program is then invoked by overlapping the input icon(s) on the secretary icon. The output icons will be produced as a result.

HI-VISUAL is a very high level language that uses icon overlapping to perform actions in programming. These overlappings cause executable modules (which are written in a conventional programming language) to be invoked by the system. The problem with this is that, to implement behaviour even slightly different from that put in place by the system designers, a knowledge of traditional programming methods is needed.

Comparison of G/LabView and HI-VISUAL

When comparing visual languages using Shu's (1986) profiling framework, three dimensions need to be considered: the language's level; the language's scope; and the language's visual extent. G/LabView and HI-VISUAL are compared below using these three dimensions.

The level of a language is an inverse measure of how much detail must be specified by the user in order to perform an action. HI-VISUAL is of a very high level because an action is typically performed by simply overlapping two or more icons. G/LabView, in comparison, is of a lower level because more information (a structured dataflow diagram) is needed to perform an action.

The scope of a language is a measure of how wide the area of application for the language is. HI-VISUAL has a narrow scope, due mainly to its high level and the inflexibility of the underlying executable modules (they cannot be changed easily). G/LabView, in comparison, has a wider scope of applicability because its actions can be defined more flexibly (due to the lower level of the language).

The visual extent of a visual language is a measure of how great a part graphical representations play in the programming process. HI-VISUAL has a very high visual extent because programming is almost exclusively carried out by overlapping icons.

G/LabView has a lower level of visual extent than HI-VISUAL because text is needed for the identification of the input and output variables.

The three dimensional profile of G/LabView and HI-VISUAL is shown in Figure 2.15. Note that while the three dimensional profiles can be used to compare the individual aspects of the languages, the overall size or shape of the profile cannot be used to judge which is the 'better' language overall.

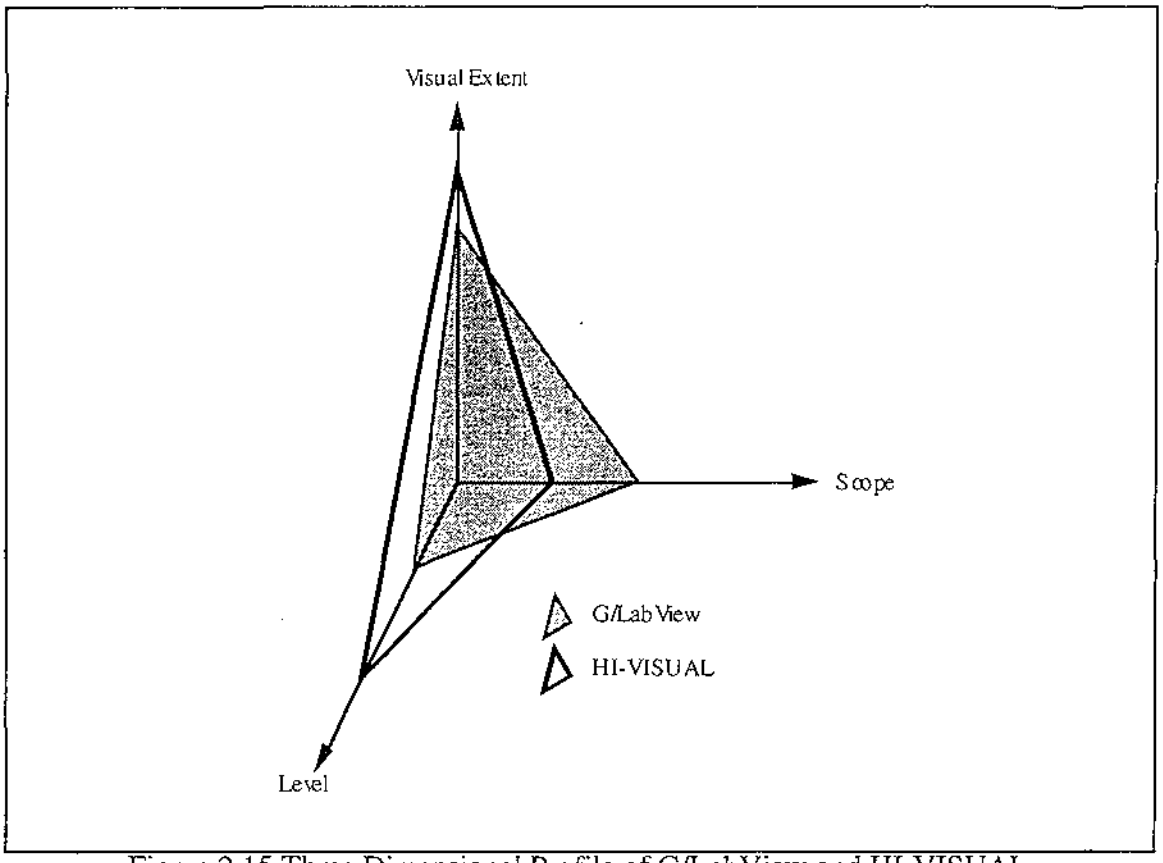


Figure 2.15 Three Dimensional Profile of G/LabView and HI-VISUAL

2.4. General-purpose Visual Programming Languages

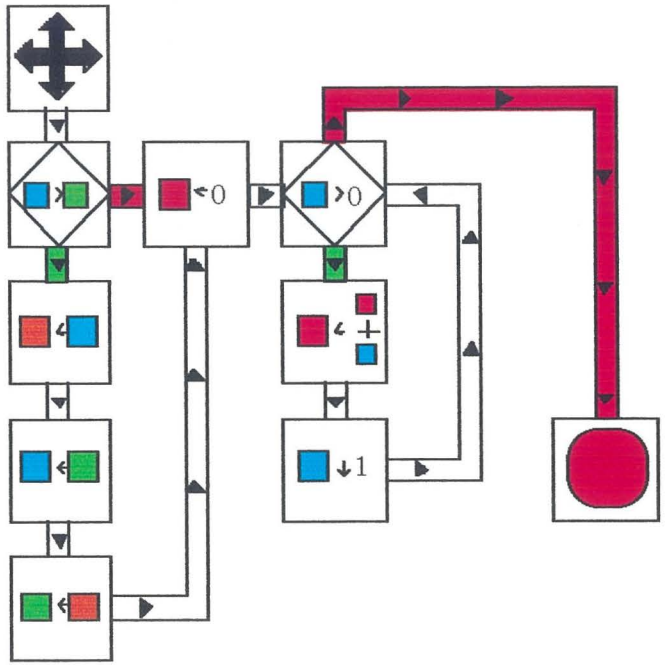
A general-purpose visual programming language is one that is designed for use in a range of application areas. Two general-purpose visual languages are described, then compared in this section, using Shu's profiling approach (Shu, 1986).

Pict/D

Pict/D (Glinert and Tanimoto, 1984) is a flowchart-based visual programming language, designed for general-purpose programming but, because of its limitations, used mainly as an aid for teaching programming concepts to novices.

Once the Pict/D system is started, the programmer works in a completely iconic environment where a keyboard is not needed for interaction. Users arrange and design coloured icons in order to build simple programs, and then watch them execute in the interactive run-time environment. Pict/D uses as its base a subset of Pascal that allows only one type of variable (a six digit non-negative integer), and four variable identifiers

(Red, Green, Blue, and Orange) per (sub)program. These variables are represented in the flowcharts by coloured blocks. The Pict/D flowchart and equivalent Pascal code for a simple iterative addition function are shown in Figure 2.16 (after Glinert and Tanimoto, 1984).



(a) Pict program

```
type SixDigits=0..999999;

function Red(Green, Blue:
SixDigits):SixDigits;
var Orange: SixDigits;
begin
 if Blue > Green then
 begin
   Orange := Blue;
   Blue := Green;
   Green := Orange;
 end;
 Red:=0;
 while Blue>0 do
 begin
   Red := Red + Green;
   Blue := Blue - 1;
 end;
end;
```

(b) Pascal Program

Figure 2.16 A Pict/D Function for Iterative Addition (after Glinert and Tanimoto, 1984)

In a Pict/D program the flow of control starts at the  icon, and follows the arrowed paths around the rest of the flowchart. When a conditional branching is reached, the condition is evaluated and, if evaluating to true, the branch coloured green is taken, otherwise the red branch is taken. Loops can be created by branching the flow of control back to a conditional icon, resulting in a circular flow. This circular flow continues until the branch without the circular flow is taken, or the finish of the flowchart (the  icon) is reached.

Subprograms are created in Pict/D by associating an icon with a flowchart. The icon specifies the variables the subprogram uses for input, for returning a result, and for local variables. Subprograms in Pict/D can call themselves, so recursive programs can be created.

Pict/D programs are limited mainly to simple mathematical manipulations, with the most limiting factor of Pict/D being its complete reliance on graphical representations. Representing variables with coloured blocks is not as descriptive to the user as a textual name, and restricts the number of possible variables excessively. Allowing only one manipulation (such as assigning one variable to another) per icon means that, unless parts of the overall manipulation are grouped into subprograms, valuable screen space can be taken up with simple variable manipulations, or functions. The association of a subprogram with an icon also raises problems when trying to design meaningful icons

for subprograms, as some actions have a meaning not easily represented by an icon. This is demonstrated by Glinert and Tanimoto (1984) when designing the icon to represent a subprogram that calculates a Fibonacci sequence. The "icon" designed simply contains the abbreviation "Fib".

Prograph

Prograph (Cox, Giles, and Pietrzykowski, 1989) is an object-oriented dataflow-based visual programming language, designed for general-purpose use.

The object-oriented programming paradigm (see Budd(1991)) makes use of *objects*, which are encapsulations of data, and the actions performed on that data. An object is an instance of a *class*, where a class is a description of the data types and actions shared by a group of similar objects. The data values of an object are called its *attributes*, and the actions defined for an object are called its *methods*. Classes can be formed into *inheritance hierarchies*, where classes inherit attributes and methods from those higher in the hierarchy.

Prograph programs are constructed by first defining their classes in the class hierarchy. The two components of  , the class icon, represent the classes attributes and methods. Opening the left half of a class icon takes the user to the definition of the attributes, which allows the user to view and modify attribute names and default values, or to add new attributes. Opening the right half shows the user the methods defined in the class (but not those inherited). Opening a method will then allow the user to view and modify the method definition in the Prograph dataflow notation. The Prograph user interface is shown in Figure 2.17.

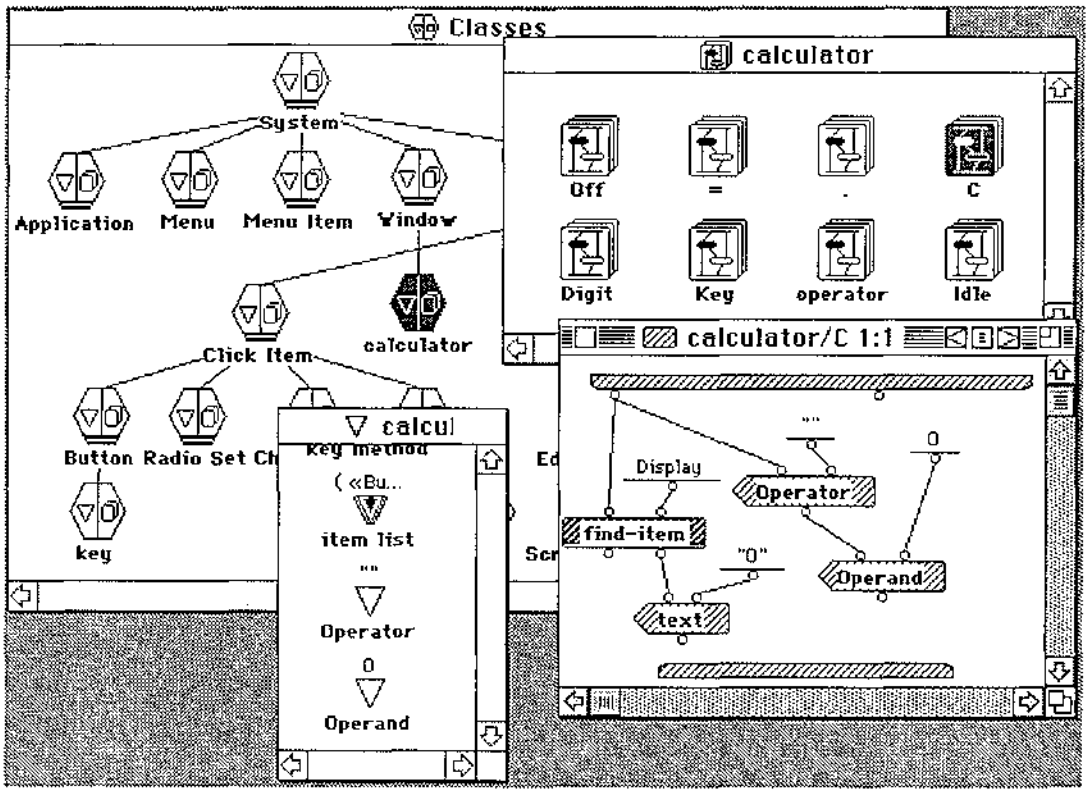


Figure 2.17 The Prograph User Interface

The Prograph dataflow notation consists of icons with different shapes and names, joined together with arcs. Arcs terminate at inputs and outputs, which are shown by small circles called *roots*. A number of different shaped icons, and their meanings are shown in Figure 2.18.

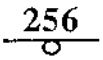
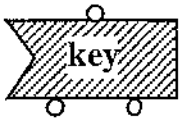
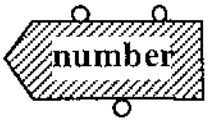
Operation	Sample Call	Action
Input		Copy value from terminal of calling operation
Output		Copy value to root of calling operation
Simple		Call the Quicksort method
Constant		Output constant 256 on root
Match		Next case if value on root is not NULL
Get Attribute		Output value of attribute "key" of input instance on right root, and the instance itself on the left root
Set Attribute		For instance on left root, set the value of attribute "number" to the value on the right root, and output

Figure 2.18 Prograph Icons (excerpt from Cox *et al*, 1989)

Conditionals and iteration are difficult to represent with pure dataflow notations. In Prograph, a method consists of one or more *cases*. When a method is executed, its first case is executed first. In order to implement conditionals, Prograph provides an icon that can terminate a case, depending on the value of the data on its input. When a case is terminated, the next case is executed in its place. This mechanism is similar to the unification and backtracking mechanisms provided in Prolog. An example of a conditional in Prograph is shown in Figure 2.19. In this example, if the input value is "Bubble Sort" the bubble sort method is executed, otherwise the quicksort method is executed.

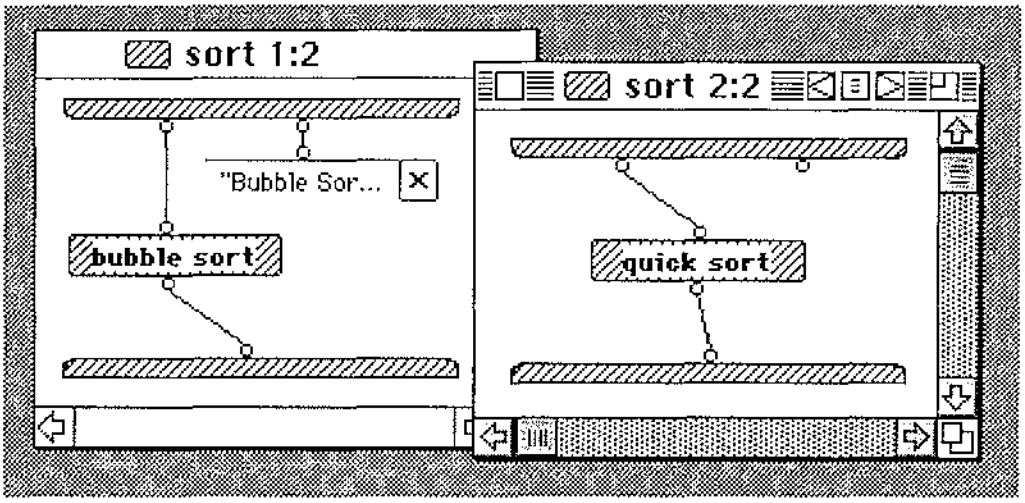


Figure 2.19 A Conditional in Prograph

Iteration is provided in Prograph with the use of *multiplexes*. A multiplex iterates through a list, performing an action on all or some of the items in the list. An example of a multiplex is shown in Figure 2.20. In this example, firstly the value of the "entries" attribute of an object is found. This list is then iterated through, finding the value of the "key" attribute of each of the objects in the "entries" list. These values are then assembled into a list, and stored in the "key list" attribute of the original object.

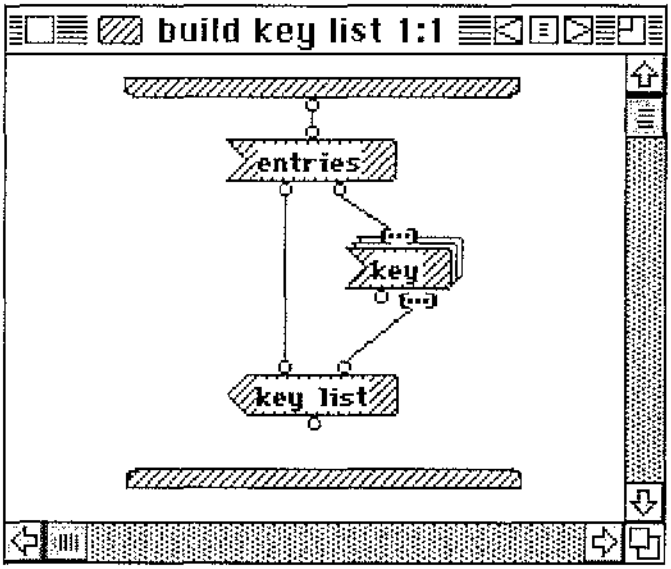


Figure 2.20 A Prograph Multiplex

Prograph is a powerful general-purpose visual programming language and is possibly the most widely used. Prograph's object-oriented nature allows the use of many pre-defined classes and methods in the user interface, while its dataflow notation copes well with both conditionals and iteration.

Prograph's dataflow notation suffers when methods become complex. Not only does the number of different (but quite similar in appearance) icons cause confusion, but in many cases, confusion is caused by arcs intersecting and becoming hard to follow. The notation also suffers, similarly to G/LabView's, in space usage when trying to represent even simple mathematical equations. A Prograph method is shown in Figure 2.21,

representing the equation previously shown in figure 2.9(b), $\frac{input + 1}{2} \times s \rightarrow output$.

This again illustrates the inefficient use of space resulting from graphical dataflow notations being used at low levels.

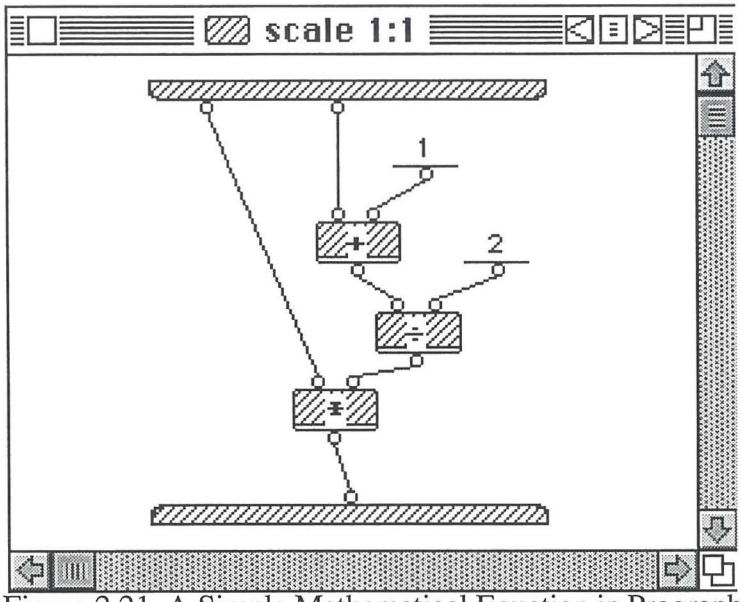


Figure 2.21 A Simple Mathematical Equation in Prograph

Comparison of Pict/D and Prograph

When comparing visual languages using Shu's (1986) profiling framework, three dimensions need to be considered: the language level; the language scope; and the visual extent of the language. Pict/D and Prograph are compared below using these three dimensions.

Pict/D and Prograph are both algorithmic, but Prograph is of a higher level because it allows iteration to be handled at a higher level of abstraction, and conditionals with more than two alternatives.

Pict/D has a narrow scope (limited mainly to simple mathematical manipulations), due mainly to its complete reliance on graphical representations, and the limitation to one data type. These restrictions mean that programs and subprograms are limited to using a maximum of four variables (including input and output parameters), each of which is a six digit non-negative integer. Prograph, in comparison, has a much wider scope of applicability because it allows the user to declare many variables (objects) of user-defined types (classes). This, combined with the inheritance hierarchy of system classes, allows a wide range of applications to be produced.

The visual extent of a visual language is a measure of how great a part graphical representations play in the programming process. Pict/D has a very high visual extent because it is completely reliant on graphical representations. This reliance extends to both variable names (the colours Blue, Green, Red, and Orange), and subprograms, which are named with icons only. Prograph has a lower level of visual extent than Pict/D because text is used for the identification of objects, classes, attributes, and methods. This is a good example of a high visual extent not necessarily being desirable, with the limitations of Pict/D being brought about mainly because of its complete reliance on graphical representations.

The three dimensional profile of Pict/D and Prograph is shown in Figure 2.22.

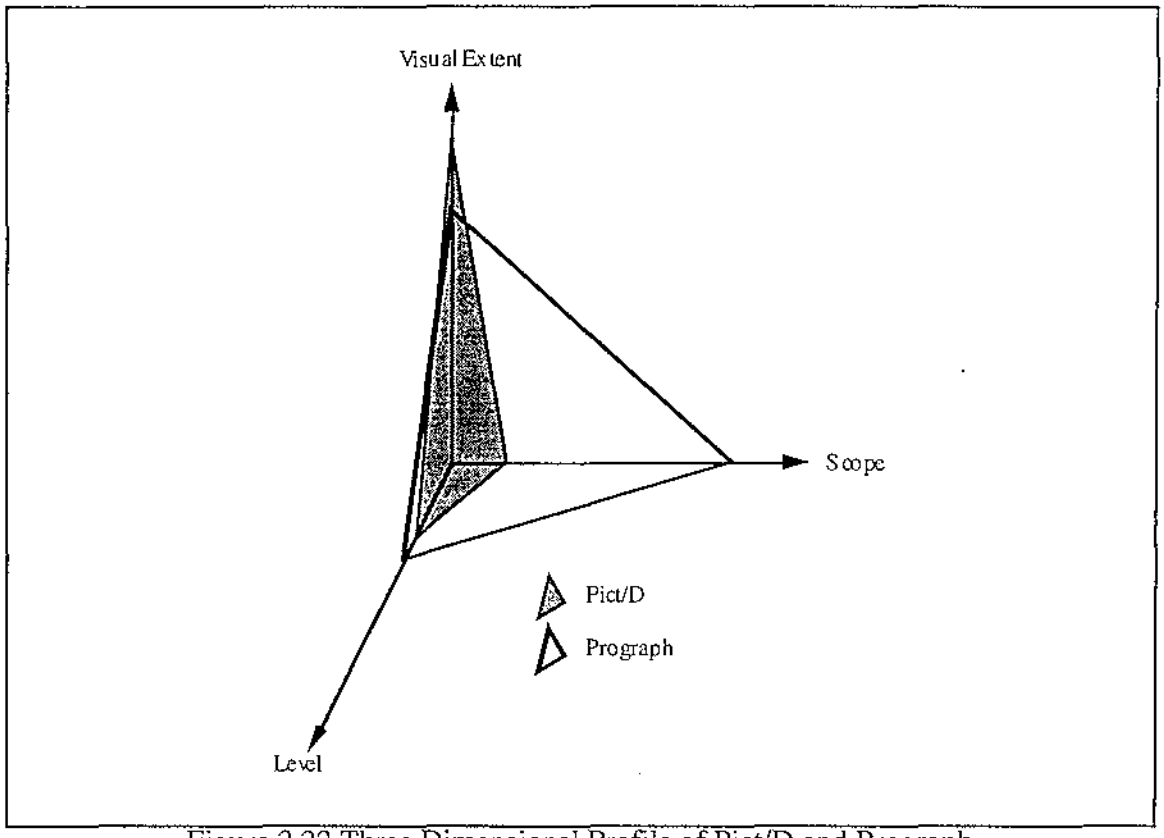


Figure 2.22 Three Dimensional Profile of Pict/D and Prograph

In general, Prograph is a more powerful language than Pict/D but, because of its complexity, it would be more difficult to teach to novice programmers. Pict/D however, while being easy to teach to novices (Glinert and Tanimoto, 1984), uses a flowchart notation that may emphasise 'bad' programming practices (such as the use of the *goto* statement). Issues dealing with how novices learn to program are dealt with in Section 2.5.

2.5. Learning to Program

One of the issues stimulating the development of visual programming languages is the desire to help novices learn to program. In this section, we examine the differences between novice and expert programmers, and how novice programmers can be helped to become experts.

Differences Between Novices and Experts

Mayer (1988) identifies four areas of difference between novice and expert programmers. These are the areas of *syntactic knowledge*, *semantic knowledge*, *schematic knowledge*, and *strategic knowledge.*

Syntactic knowledge is the knowledge of the language units, and the rules on how these units are combined to produce meaningful code. Wiedenbeck (1985) suggests that experts have automated their syntactic processing, whereas novices must devote more attention to obeying syntactic rules. This means that experts can devote most of their attention to the higher level problem solving tasks involved in programming.

Semantic knowledge is the knowledge of the underlying concepts involved in programming. The semantic knowledge of a system is often referred to as a *mental model*. Findings by Goodwin and Santani (1986) suggest that expert programmers, when compared to novices, have a more accurate and complete mental model of programming concepts.

Schematic knowledge is the ability to recognise the function of common software structures. Wiedenbeck (1985) suggests that experts recognise common "chunks" of code as having a particular function, and therefore can be treated as a single unit. This allows expert programmers to program at a higher ("chunk") level, rather than the statement level common amongst novices.

Strategic knowledge is the ability to organise program units in order to solve problems. Mayer (1988) suggests that expert programmers are more likely than novices to decompose problems into smaller parts, and better comprehend the abstract goals of a program.

Fix, Wiedenbeck, and Scholtz (1993) have compared the mental representation of programs between novice programmers and experts, and found five characteristics that were present in the experts' representation, but not in novices'. These characteristics are:

It is hierarchical and multi-layered;
It contains explicit mappings between high and low levels;
It is founded on the recognition of basic patterns;
It is well connected internally;
It is well grounded in the program text.

Learning to program is a transition from novice to expert, so a good programming language for novices should support novices in the areas where they are deficient, as well as helping them to attain the attributes typical of experts.

Helping Novices Become Experts

Visual programming languages can be suitable for supporting novices because they can sidestep many of the problems associated with traditional programming languages. Many visual programming languages avoid the complex syntax typical of textual languages (by using simple graphical representations), thus decreasing the amount of syntactic knowledge required by the programmer. The use of direct manipulation, rather than typing, can also minimise the confusing effect of spelling mistakes.

Human beings are good at recognising visual patterns and thus recurring "chunks" of code should be inherently easier to recognise in a visual programming language than they are in a textual one. This can compensate for the novices' lack of schematic knowledge.

Visual programming languages can be used to explicitly show information about programs that novices find hard to conceptualise with their incomplete mental model. Fix *et al.* (1993) identified a number of tasks that novices find significantly more difficult than experts. These tasks include: matching procedures with the procedures they call; matching variable names to the procedures in which they occur; listing the names used for the same data object in different program units; and filling in the names of program units with a skeleton outline of the program. A visual programming language could provide this information graphically, thereby increasing program comprehensibility for novice programmers.

One attribute present in an expert's mental representation of a program is that it is well grounded in the program text (Fix *et al.* 1993). This includes specific information on

where structures and operations are located in the physical program. It would be useful to novices (and programmers in general) if navigation in the physical program was aided in some way by the environment. Navigation inside a program is discussed in Chapter 3.

While supporting novices is important, it is also important for a language to support expert programmers.

Supporting Expert Programmers

The support needed by experts when programming is not as extensive as the support needed by novices but, while experts performed better in tests than novices (Goodwin and Santani (1986), Wiedenbeck (1985), and Fix *et al.* (1993)), their performance was not perfect. Experts can therefore benefit from the provision of similar support to that provided for novices. It is important, however, that the provision of this support does not hinder the expert programmer. The graphical presentation of visual languages has the capability to hide, or show selected information, therefore allowing expert programmers to view only the information they need.

Expert programmers have a complex multi-dimensional mental model of programs. This multi-dimensional model is inadequately represented using text. Visual programming languages enable programs to be represented in a two, or more, dimensional manner, and can therefore provide a representation closer in dimensionality to the mental model held by programmers. It is also possible to encode information into the representation using colour or shape to provide extra dimensions.

The interaction techniques used by expert programmers often differ from those used by novices (Shneiderman, 1988). A visual programming language can enable a wide range of interaction techniques to be used. By allowing the direct manipulation of graphical components and menu selection, support is provided for novices, and by allowing the use of "shortcuts" such as keyboard entry of components and the use of power keys, support is also provided for expert programmers.

2.6. Summary

Research into visual programming has focussed on two main areas; *Visual environments*, and *Visual Languages*.

Visual environments use the visualisation of data, programs, or systems, to support the debugging and comprehension of programs. The visualisation of data can enable users to 'step through' a program, viewing the effect program statements have on the data, thus aiding in the locating of errors, and comprehension of program statement functionality. The visualisation of programs enables users to view the structure of a program, and more easily identify structural errors in a program. The visualisation of systems allows users to view the modules and interconnections of a system, and can aid in the identification of over-utilised/under-utilised nodes, and errors in the connections between modules.

Visual Languages manipulate either naturally visual information, or logical information with an imposed visual appearance, and may have either a linear (textual) expression or a visual expression. Languages that manipulate naturally visual information (such as pictures) are called visual information processing languages or iconic visual information processing languages, depending on their language expression (iconic = visual expression). Visual languages that have a linear expression, and manipulate logical information with an imposed visual appearance (such as dialogs), are called languages for supporting visual interaction. The last type of visual language, one that manipulates

logical information and has a visual expression, is called a visual programming language, and is the type of language this thesis is mainly concerned with.

The area of visual programming languages has been divided differently by different authors. We divide the area into two classes, based on the area of applicability of the language; general-purpose visual programming languages, and domain-specific programming languages. Domain-specific visual programming languages are designed for a specific area of application, whereas general-purpose visual programming languages are designed for use in many application areas.

Research has shown several areas where novices need (and experts can be aided by) support from a programming language. Visual programming languages have the capability to provide this support, and should therefore provide a language that is not only easy to learn and use for novices, but is also powerful enough not to hinder expert programmers.

Chapter 3

HyperProgramming

3.1. The Nature of Software

The design and implementation of software is a process of mapping an often vague concept to a concrete programmatic representation. This mapping is usually divided into two phases, firstly mapping the conceptual idea to an abstract representation of the program, and then mapping this abstract representation to a concrete representation of the program (implementation).

The Multi-dimensional Abstract Representation

An abstract representation of a program is a description of the components and actions which are necessary for the completion of a particular task. Within broad limits (a functional programmer's abstract representation is likely to differ from a procedural programmer's), this representation is independent of the actual method of implementing the components and actions.

An abstract software entity is described by Brooks (1987) as "... a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions". This representation makes up a complex and multi-dimensional structure, and can include additional dimensions representing concepts such as time sequence, flow of control, flow of data, overall appearance of input and output, and scoping and dependency relationships. The components and relationships that make up this representation may occupy many dimensions.

Research by Fix *et al.* (1993) suggests that the multi-dimensional nature of software is also present in the mental representation of programs held by expert programmers. They found that an expert's mental representation of a computer program was hierarchical and multi-layered in nature, and included explicit mappings between the layers. These layers, and the mappings between them, correspond to the interlocking concepts discussed by Brooks.

We suggest that, since both the abstract representation of software and the mental representation of programs are multi-dimensional in nature, a software entity is best modelled in a hyperspace, with different concepts occupying different (but related) dimensions.

Concrete Representations of Software

Until recently, technological factors have restricted the concrete representation of software to a linear string of text. Much of the complexity involved in programming is the mapping of the multi-dimensional, interlocked, abstract representation of a program to this linear, textual, concrete representation. This textual representation also hides many of the relationships present in the multi-dimensional abstract representation, and disperses closely related components throughout the text. Consider the Pascal program excerpt in figure 3.1 as an example.

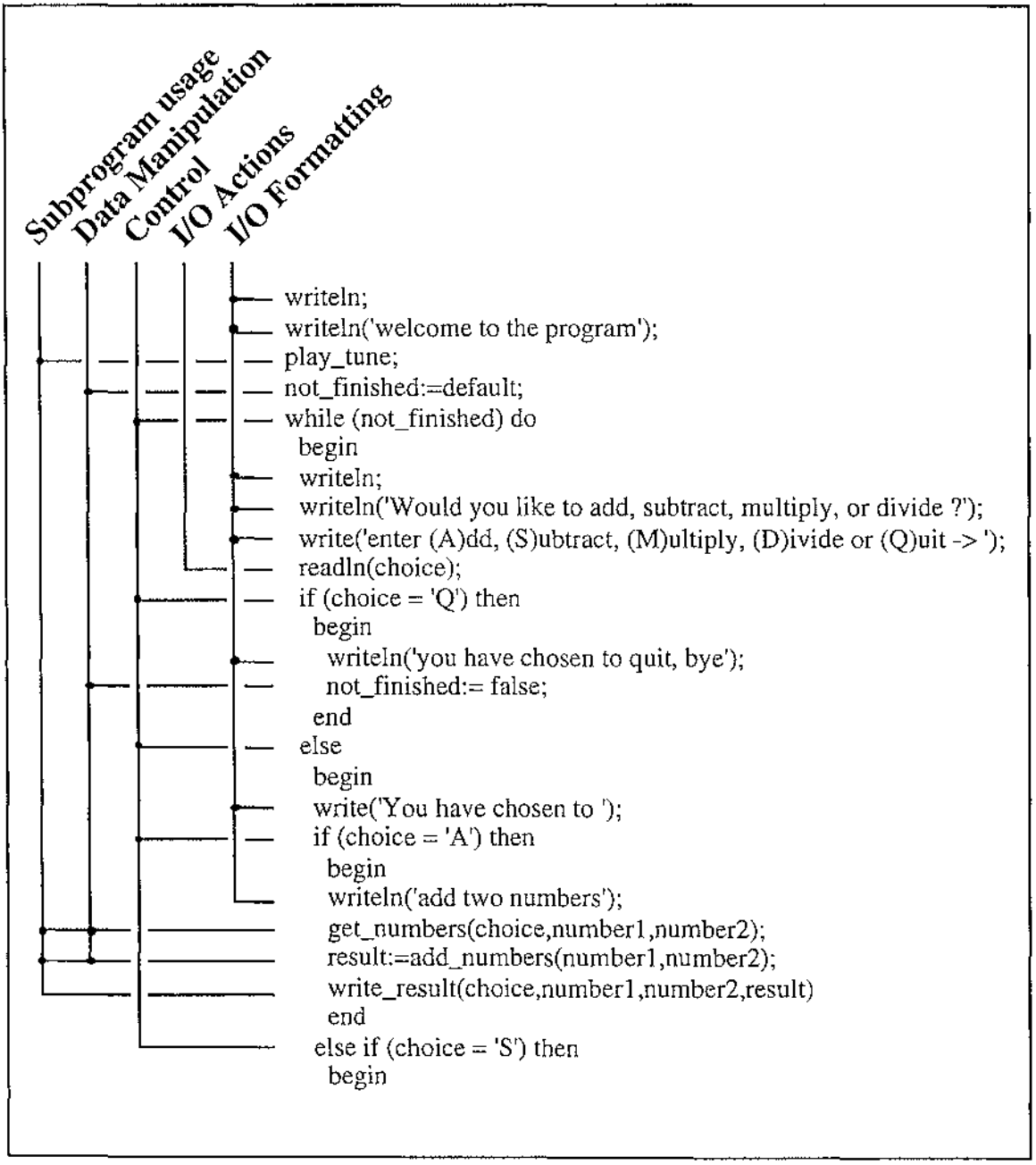


Figure 3.1 Distribution of Related Information in a Linear Program Representation

Figure 3.1 shows how related information in a program is dispersed throughout a linear representation. The input and output formatting is dispersed throughout the program, which makes it difficult to comprehend as a whole, and also serves to conceal the underlying functionality of the program.

The functionality of this program excerpt can be described by its input and output of data, control structure, data manipulations, and subprogram invocations. There is only one data input action, and it is important because the data value input controls the execution of the rest of the program, yet it is hard to identify at a glance.

The control structure of the program is dispersed throughout the linear representation, and it is difficult to match different branches of the same control structure (in the general case, they may be many pages away from each other). The scope of control is indicated in a linear representation by matching keywords or symbols (for example, the 'begin' and 'end' keywords in Pascal, and the '{' and '}' symbols in C), which becomes difficult to recognise with nested control structures, or when the matching keyword or symbol is on another page. Indentation is usually used in program listings to further illustrate scope of control, but it loses its effect over several pages of text.

A language such as Pascal allows data manipulations to take the form of assignments or side-effects. Assignment statements are typically dispersed throughout the program text, but can be identified by their 'a:=b' format. Side-effect data manipulations are also dispersed throughout the text, and occur when subprograms are invoked. A common method for returning more than one value from a subprogram (in Pascal) is by side-effect data manipulations using variable parameters. This causes the values of variables passed to a subprogram to be different when the subprogram returns. The invocation of these subprograms is no different from the invocation of a normal subprogram, so without viewing the subprogram header (which may be hard to find), there is no way of differentiating between these data manipulations and a normal subprogram invocation (in figure 3.1, the very similar invocations 'get_numbers(choice,number1,number2)' and 'write_result(choice,number1,number2, result)' are side-effect manipulation, and normal invocation respectively).

An invocation of a subprogram can be many pages (occasionally hundreds of pages) separated from the definition of the subprogram. This separation of description and invocation may also be in either direction in the text (by using forward declarations). To investigate the functionality of a subprogram when using linear representation, the whole document has to be searched.

Due to the distribution of related information and hiding of important relationships typical of linear (textual) representations of programs, we suggest that a multi-dimensional software structure is poorly modelled using a linear representation such as text. Diagrams are more suitable as a non-linear specification method.

A Non-Linear Concrete Representation of Programs

Describing all of an abstract software structure on paper usually involves the drawing of several diagrams of different types, each sharing components with other diagrams. These diagrams are two-dimensional projections from the multi-dimensional software structure. There are many such projections (showing different relationships, or groupings of components), and the use of a visual programming language allows a combination of these two-dimensional projections to form a concrete representation of a program. Also, since the abstract software structure is independent of implementation, it is possible to describe the same program using different combinations of projections (in the same way it is possible to implement the same program in different languages, or using different methodologies).

A number of two-dimensional diagrammatic notations already exist for the description of different aspects of a software structure. These two-dimensional projections include data flow diagrams, object diagrams, structure charts, HIPO diagrams, structured flowcharts, state transition diagrams, and decision tables (Fairley, 1986). A visual programming language could be constructed to use an appropriate subset of these (or other) projections.

When using projections from a multi-dimensional structure onto multiple two-dimensional spaces, different relationships and component groupings are present in different projections, and the same relationship or component may be present in more than one projection. These projections are also related to each other (because of shared components, or other conceptual relationships), and it is necessary to navigate between, and within, these projections in order to fully specify the multi-dimensional program. Navigation through, and specification of, a multi-dimensional medium is an issue dealt with in the field of *hypermedia.*

McDaid (1991) defines hypermedia as ". . . computer mediated storage and retrieval of information in a non-sequential fashion". The non-sequential nature of hypermedia documents (*hyperdocuments*) is supported by an underlying structure composed of *nodes* and *links.* A node is a container of information (possibly a view onto a software structure), and nodes are interconnected by pointers called links. The links allow a user to navigate between nodes easily, and support a multi-dimensional structure of nodes.

Combining these two concepts, (a visual programming language that uses several views of an abstract multi-dimensional program space, and hypermedia techniques for creating and navigating through such a set of views) leads us to conceive of a new type of computer program, the *hyperprogram.* The process of creating and modifying a hyperprogram is called, not surprisingly, *hyperprogramming.*

3.2. HyperPrograms

Let us consider the possible makeup of a programming system which uses hypermedia techniques to facilitate the construction of, and navigation within, a multi-dimensional computer program. Hypertext techniques can then be used with this hyperprogram to group, and navigate between, related components of a program, with each set of related components providing a different view of the (single) multi-dimensional abstract representation. These different views can be stored as nodes, inspected by traversing the links between nodes, and modified by altering the information in the nodes.

It is possible for the same component to be present in more than one view. Since these multiple occurrences are actually views of a single component, the modification of the component (in any view) should be reflected in the other views that contain it. It is convenient, therefore, to have automatic update links between views onto the same component.

Consider as an example, an abstract program consisting of a number of subprograms, each of which is associated with a subprogram declaration, and some actions (which can include the calling of other subprograms). The simple example in figure 3.2 shows a program with one subprogram. The scoping relationship between the program and the subprogram is shown in the view of the subprogram declarations. Both the program and the subprogram have associated actions (which can include the invoking of a subprogram), which are shown (with the subprogram's declaration) in that subprogram's action view. The relationships between these views can be represented using hyperlinks. The relationships represented by hyperlinks in figure 3.2 include those between a subprogram declaration and the view of its actions (activated from the subprogram

declaration view), between a subprogram declaration and the view of its scope (activated from the subprogram's action view), and the relationship between the invocation of a subprogram and the subprogram's actions.

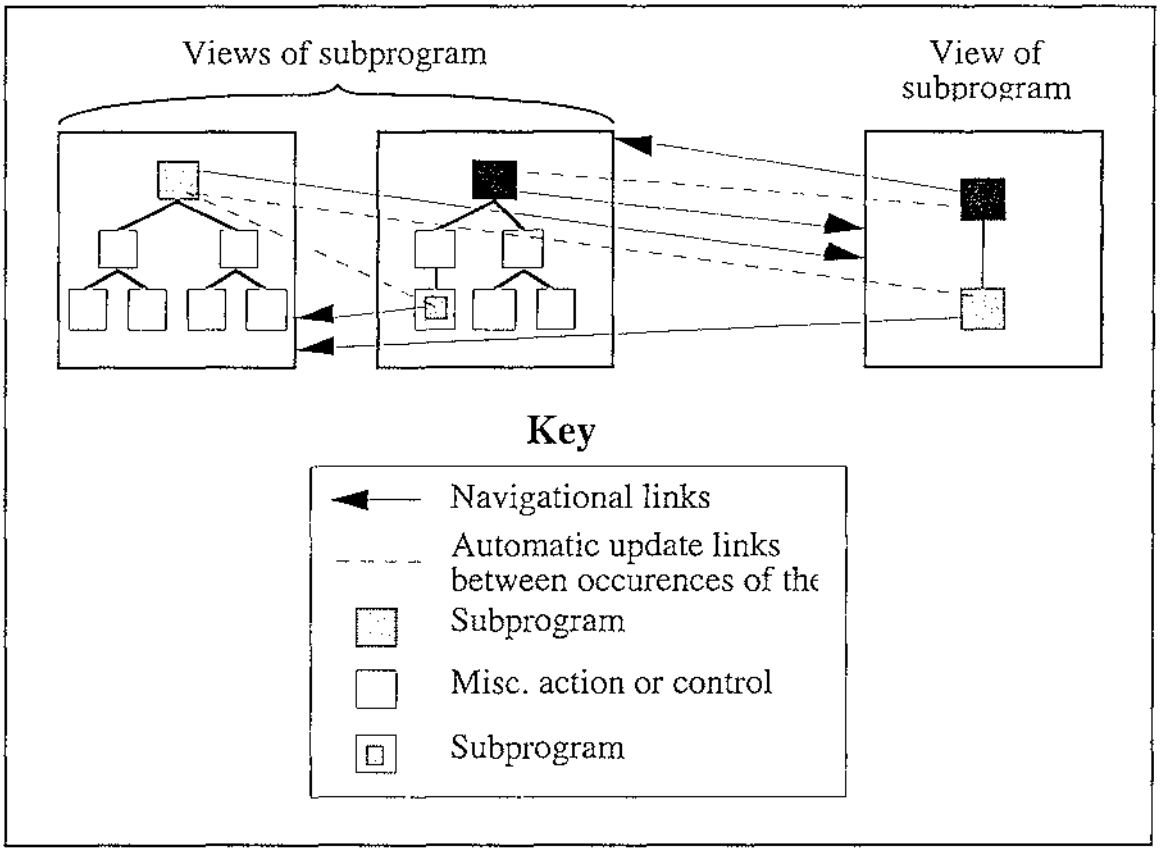


Figure 3.2 A Possible Underlying Structure for a HyperProgram

The hyperlinks shown in figure 3.2 not only represent relationships between the views, but also enable the programmer to navigate between views. In this simple model, navigation between the subprogram action views and the declaration scope view involves by activating the hyperlink from a subprogram declaration (which is common to both types of node). Navigation is also possible from one action view to another by activating the hyperlink from a subprogram invocation to the view of that subprograms actions.

The other links shown in figure 3.2 are the update links which allow modifications of a component to be propagated to other occurrences of that component. In this example, a subprogram declaration can be present in both the subprogram declaration view, and in the subprogram action view. The representations of a component in more than one view are actually views of the same component. Therefore in figure 3.2, a modification of a subprogram (in any view) will be propagated to all other views containing that subprogram.

The grouping of components according to a shared property or characteristic (e.g. all being actions of a particular subprogram) can be used to define node types (a node (view) is an instance of a node type). Nodes of a particular type have a common syntactic and navigational structure. The hyperprogram shown in figure 3.2 has two node types; the actions for a subprogram, and the subprogram scoping. The structure of, and navigation from these node types is shown in figure 3.3.

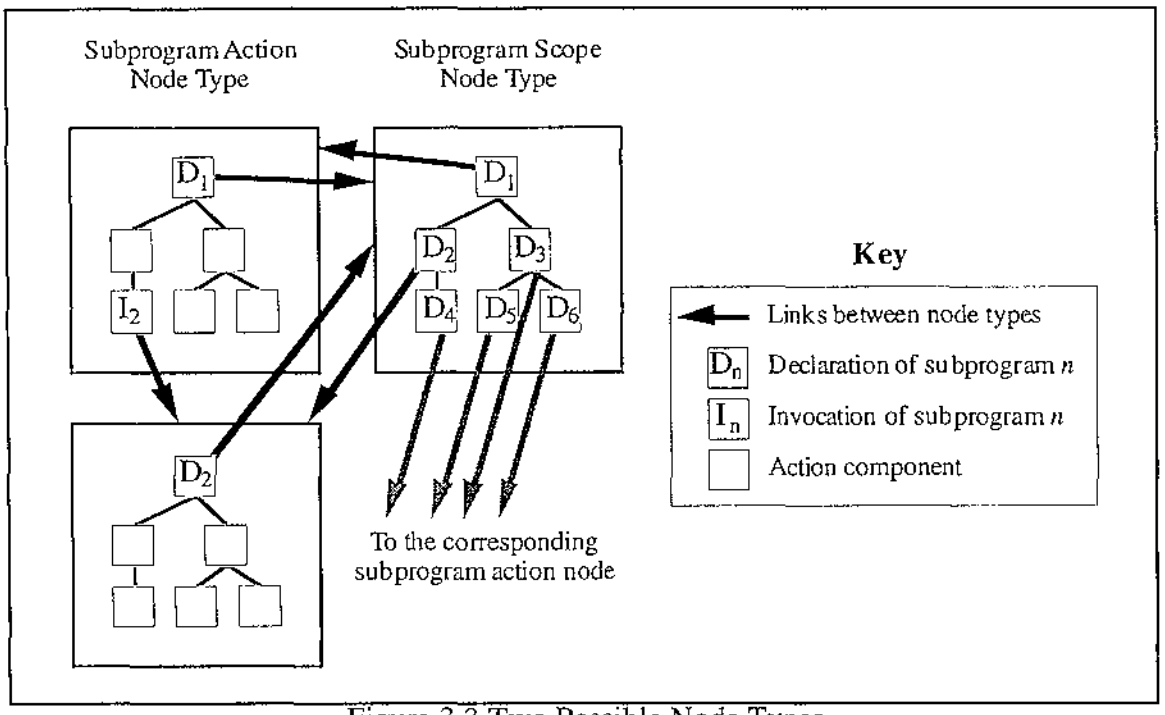


Figure 3.3 Two Possible Node Types

Figure 3.3 shows that a node of type 'Subprogram Action' contains a hierarchy of action components headed by a subprogram declaration, and a node of type 'Subprogram Scope' contains a hierarchy of subprogram declarations. The linking structure of the node types is also shown, with every subprogram declaration in the 'Subprogram Scope' type linked to a 'Subprogram Action' type, and the single subprogram declaration in the 'Subprogram Action' type linked to a 'Subprogram Scope' type. An invocation of a subprogram in a 'Subprogram Action' type is linked to the 'Subprogram Action' type for the subprogram invoked. Every instance of these node types will have this structure and linking behaviour.

3.3. Discussion of HyperProgramming

A hyperprogram is a multi-dimensional representation of a program that uses multiple (graphical) views of the abstract program space, and hypermedia techniques for creating and navigating through the set of views. The process of creating and modifying a hyperprogram is called hyperprogramming.

The use of a concrete representation of programs, based on two-dimensional projections from the abstract representation, allows the related structures within the abstract representation to be retained intact, and avoids the complexity involved in breaking them up and dispersing them throughout a linear concrete representation.

While a multi-dimensional concrete representation allows a structure of related components to be stored together in a view, this view may also be related to other views. It is desirable to enable the programmer to move to these related views easily. The hypertext technique of linking nodes of information allows us to travel between views easily (because views are stored as nodes).

Apperley (1989) identifies three problem areas concerned with hypermedia; the structuring of information, the management of this information, and the problems of navigation. These problems are addressed (in relation to hyperprogramming) below.

The first of the problems identified is the structuring of information. Apperley states "The link flexibility of hypermedia permits what were once orderly hierarchies to become anarchical networks". The hyperprogramming approach allows the existence of link networks, but prevents them from becoming "anarchical" by reducing link flexibility through the definition of common linking behaviour for nodes of a particular type.

The management of information in a conventional hypermedia system becomes a problem because people with different mental models of hyperspace can add their own links between nodes, thus creating confusion. Node types can be used with the hyperprogramming approach to define common linking behaviour for nodes of a particular type, thus ensuring links are added in a consistent manner, and reducing the risk of people with differing mental models adding confusing links.

There are two aspects to the problem of navigating in a hypertext system; not being able to find the information you want, and not knowing where you are in the system (Shneiderman and Kearsley, 1989). While visual cues (such as colour and shapes) are helpful to the user for locating their current position in the hyperdocument, Martin (1990) states that the most important way to help the user navigate efficiently in a hyperdocument is by structuring it well, and making its structure clear to the user. The use of node types with well defined (and easily comprehended) structure and linking behaviour, imposes a consistent and easily comprehensible structure throughout a hyperprogram. A central, overview node, such as an instance of the "Subprogram Scope" node type (in figure 3.3), can also help reduce the "lost in hyperspace" problem by acting as a kind of central index.

In summary, the application of hypermedia techniques to programming could allow programs to be created and stored in a multi-dimensional concrete representation we have called a hyperprogram. The multi-dimensional nature of a hyperprogram is closer to a programmer's abstract mental representation than is text, and it therefore reduces the complexity of mapping an abstract representation to a concrete one. Finally, a hyperprogram allows the grouping of information and programming components into related views of the hyperspace, and allows the programmer to navigate between these views easily.

We now present the design of a visual programming language, HyperPascal, that enables the creation and manipulation of a hyperprogram.

Chapter 4

HyperPascal

This chapter describes the design of a general-purpose visual programming language called HyperPascal (Lyons, Simmons, and Apperley, 1993). HyperPascal is a hyperprogramming language that uses the imperative programming paradigm, and whose semantics are based closely on those of Pascal.

4.1. The Visual HyperProgramming Language

When designing a hyperprogramming language, different views of the hyperspace are designed to highlight different aspects of the hyperprogram. The different views in HyperPascal are designed to represent the elements commonly found in procedural programming languages (Pascal in particular), and to show the relationships between the elements in a way that is helpful to programmers. There are four different views of a multi-dimensional program in HyperPascal, and these different views are now introduced.

Most procedural languages support the nesting of subprograms in order to allow the hierarchical decomposition of actions, with a subprogram able to call other subprograms at the same, or higher level, and also able to call the subprograms that it declares. A subprogram has a number of attributes that can be accessed by the (sub)program that calls it (such as a name, input parameters, a return type, and possibly output parameters), and also has information encapsulated inside it (such as types, constants, variables, and other subprograms), that can only be accessed by the subprogram, and the subprograms it declares.

The scope tree view is used in HyperPascal to view and modify the declarations made in a hyperprogram, and their scope. This information is represented in a graphical tree structure, showing the hierarchy of subprograms declared, their input and output parameters, and the declarations made in a (sub)program. The calling relationships between (sub)programs can be viewed or hidden as desired, and the declaration and scoping information is edited by direct manipulation of its graphical representation.

Information about a subprogram that can be accessed from other subprograms includes its name, input and output parameters, and its return type. A subprogram also declares information that can be accessed only by itself and subprograms declared inside it. This information is the types, constants, and variables declared by the subprogram. All of this information is conceptually contained within a subprogram, and should therefore be contained within the representation of a subprogram in the scope tree view. The declaration of its external attributes (name, return type, and input and output parameters)

should, however, be represented separately from the definition of its local attributes (types, constants, and variables).

Subprograms are related to other subprograms by both parent-child relationships, and calling relationships. These relationships, because they are between distinct object representations, are effectively represented in the scope tree view as arcs connecting the subprograms together. Because the nesting of subprograms is by hierarchical decomposition, the parent-child relationships among subprograms forms a tree, with a subprogram having the subprograms it declares as its children.

The calling relationships between subprograms is more complex than the nesting relationships because a subprogram is permitted to call its children, its siblings, and its ancestors. This forms a complex calling network which, if all calls were shown in the scope tree at once, could easily become incomprehensible. A way of allowing the calling relationships to be shown, but still be easily comprehensible, is to enable the user to interactively hide and show the calling relationships they want to view at a particular time. The scope tree view is discussed in more detail in section 4.2.

The functionality of a (sub)program can be divided into data manipulation actions and constructs that control the conditional, or repeated execution of the data manipulation actions. The distinction between these two classes of constructs, and between the individual constructs, can be aided by using a graphical notation (a visual programming language). This view of the functionality of a (sub)program should also show the declaration information associated with that (sub)program.

The action tree is a view of the functionality of a (sub)program, and views its functionality by showing the control structure and actions specified in the (sub)program. This information is, like the scope tree view, also represented using a tree notation, and edited by direct manipulation. The use of a graphical representation in this view allows easy differentiation between actions and control structures, and allows them to be specified without much of the complex syntax normally associated with textual languages. The action tree is discussed in more detail in section 4.3.

Part of the actions of a program is the input and output of data, and when using a text-based input/output scheme (as HyperPascal does at this stage), the input is usually from the keyboard (or a file), and the output is usually to the screen (or a file). When inputting data from the keyboard, feedback and prompts are usually provided on the screen, and information written to the screen also usually has some sort of formatting or layout information surrounding it. In traditional languages, the sending of this information to the screen is done by the same process as sending the actual data to the screen. Even worse is the fact that sending this non-essential information to the screen often camouflages the actual input and output of data (somethimes obscuring the data manipulation actions). It would be desirable for the programming language to clearly show the data being input and output, whilst also allowing an overall format and layout to be viewed and designed incorporating the input and output data.

The forms window is a view of the appearance of the input and output of a program. This view allows the input and output behaviour of a group of actions to be viewed and formatted as a whole. In the version of HyperPascal designed in this project, the input and output from the program is text only, and the forms window allows the programmer to holistically design the input and output format of the program using a direct manipulation text editor. Later versions of HyperPascal will have a more general forms window, allowing for the specification of graphical output. The forms window is discussed in more detail in section 4.4.

While these three classes of views (the scope tree, the action trees, and the forms windows) are sufficient for the construction of a program in HyperPascal, we also suggest that a view onto the tree of possible states of a program would be useful to the

programmer. This state tree mirrors the structure of the action tree, but shows a tree depicting the state of variables (where this can be ascertained at write-time) at each node. This (limited) state information can be obtained by examining the conditions satisfied in order for a particular point in the program to be reached, the results of constant assignments, values 'left behind' by loops, etc[1]. Figure 4.1 shows an example of obtaining state information at different points in a Pascal program extract.

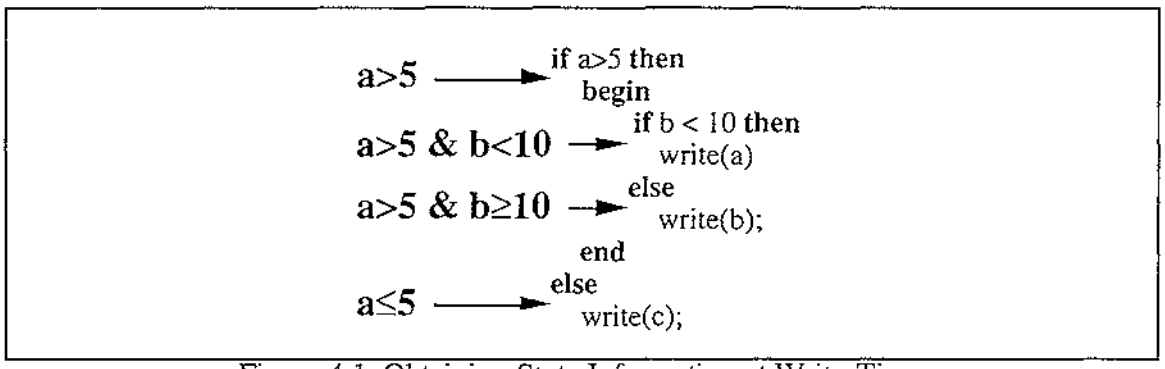


Figure 4.1 Obtaining State Information at Write-Time

It is important to note that this state information doesn't hold at all stages of the program. Consider, for example, if the 'write(a)' statement in figure 4.1 (point A) was preceded by the assignment statement 'a := c;'. Before this assignment statement, it is possible to state that a>5 and b<10, but after the assignment statement we can only state that b>10 because we have no state information about 'c'.

While we suggest a view of the state tree would be useful to a programmer using HyperPascal, a HyperPascal program can be constructed using just the scope and action tree views, combined with the forms windows. Research has concentrated on the design and implementation of these three essential views, which are now discussed in detail.

4.2. The Scope Tree

The scope tree of a program is a view onto the declarations and the scope (hence *scope* tree view) of these declarations in a program. In a HyperPascal program, as with many conventional programming languages, it is possible to nest subprograms, and to declare variables, types etc. in these subprograms. A common way to represent subprogram nesting is as a tree, with nodes representing (sub)programs, and arcs representing the nesting arrangement (as shown in figure 4.2).

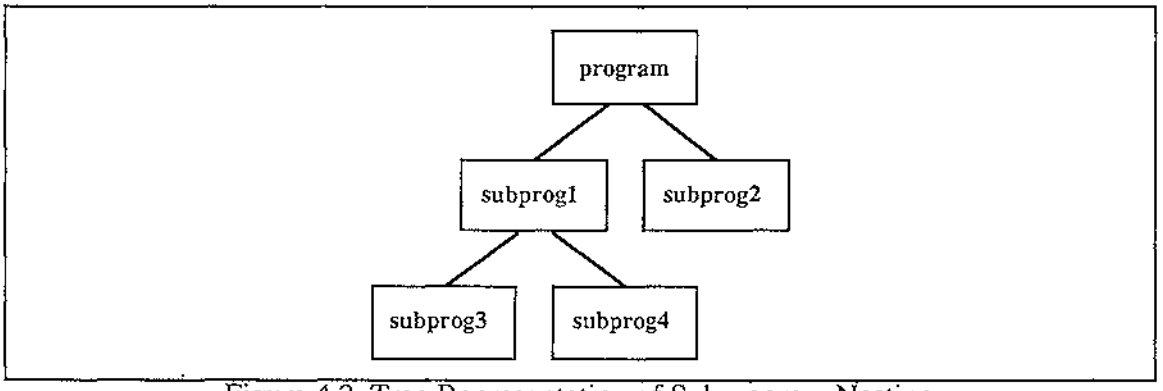


Figure 4.2 Tree Representation of Subprogram Nesting

[1] Similar to the information obtained by an optimising compiler.

The tree representation in figure 4.2 is useful (as a paper tool) for visualising the nesting of subprograms but doesn't show the variable and parameter declarations, and the calling relationships associated with the subprograms. The scope tree view is based on this simple tree representation, but allows the programmer to hide, show, and edit the declarations associated with a subprogram.

The overall appearance of the scope tree (using maximum information hiding) is shown in figure 4.3. The nodes in the scope tree view are termed *program units*, and the parent-child arcs are termed *connectors*. All program units defined in the program are present in the scope tree, so this view shows an overview of the program structure. This overview can be used as a 'central' node for navigation in the program. The background of the view is lightly colour coded in order to give a visual cue as to the type of view being edited.

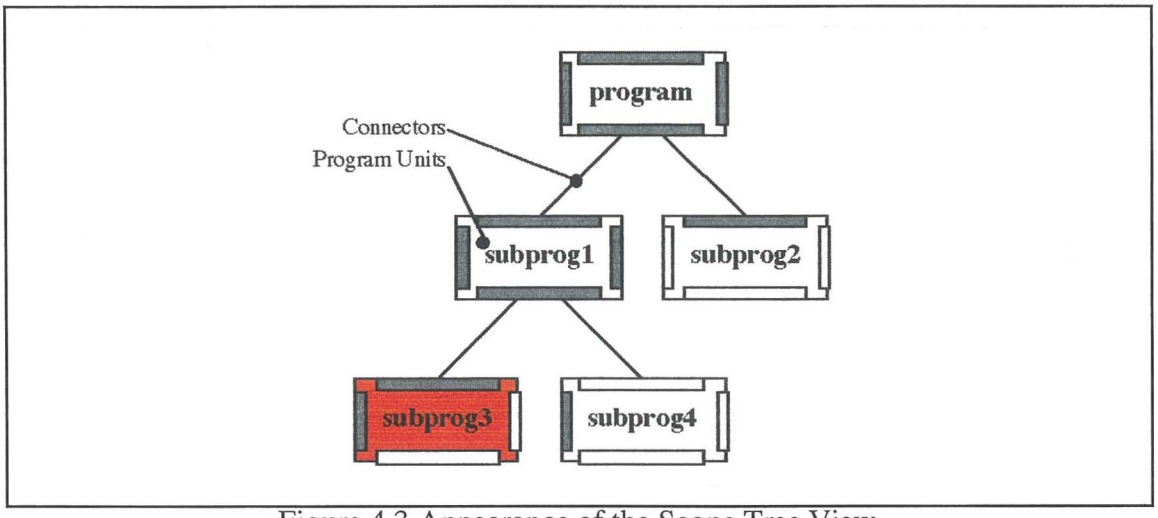


Figure 4.3 Appearance of the Scope Tree View

Program units have associated with them all declarations of types, constants, and variables defined in the program. To add, modify, or delete declarations, the program units must be edited. Program units and their component parts are now examined in detail.

Program Units

A program unit contains sub-views onto the declarations of identifiers (types, constants and variables) associated with it, and its input and output parameters. The program unit icon must display this information (as well as the subprogram calling information), while not cluttering up the scope tree diagram. Our solution to this problem is to allow the programmer to selectively hide and show different information about a program unit. The hidden areas of information are represented as thin rectangles called *collapse bars*. A program unit (with information hidden using collapse bars) is shown in figure 4.4.

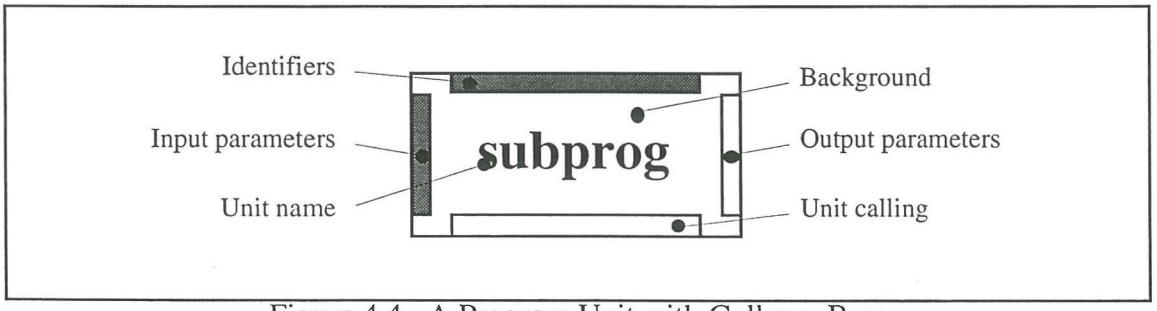


Figure 4.4. A Program Unit with Collapse Bars

The program unit is made up of six component parts, the subprogram name and background, the collapse bars representing the declarations of the identifiers, input parameters and output parameters, and the collapse bar representing the units called. The input parameters are placed on the left, and the output parameters on the right because the flow of information, and control, is generally left-to-right in HyperPascal, and this arrangement of input and output parameters represents the flow of infomation into and out of a subprogram. The identifier declarations were placed on the top of the icon so that when the area is expanded in the action tree view, it does not obscure the actions. The unit calling area is on the bottom edge of the icon because the information that it is showing is defined by its calls to other units from the action tree, and the action tree is attached on the bottom edge of the icon (in the action tree view).

The unit name is the name the programmer has given to the program unit. Program units may share the same name as long as they don't have the same parent unit. A program unit can have a return type specified, with its background colour-coded to show this information (white - none; red - real; orange - integer; green - string; blue - boolean; purple - porthole (analogous to a pointer in Pascal), grey - user defined type). If an output parameter(s) is specified for a program unit, the program unit loses its return type (the background returns to white).

Collapse bars can be expanded to show the declaration areas. They are coloured white if they are empty, and grey if they ave some contents. A program unit with the identifier, input, and output declaration areas expanded (the unit calling information is discussed later) is shown in figure 4.5.

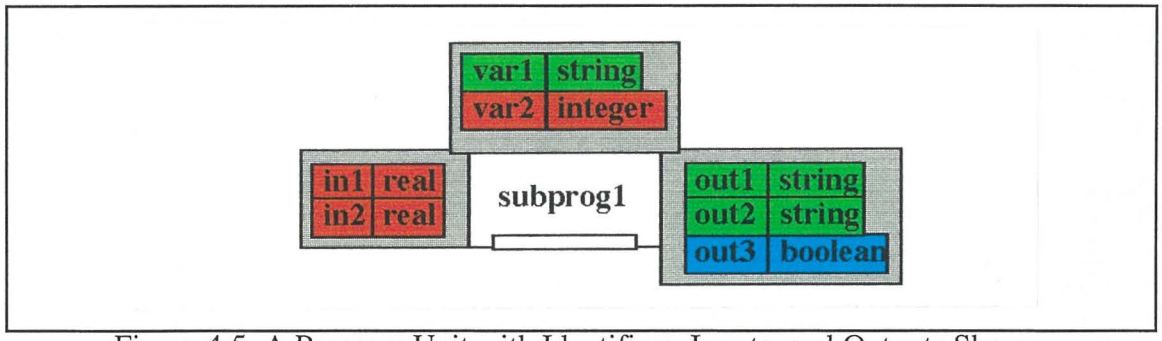


Figure 4.5. A Program Unit with Identifiers, Inputs, and Outputs Shown

The declaration areas shown in figure 4.5 each show the representations of a number of declarations. These representations are edited to create, modify, and delete the declarations defined for a program. The declaration of constants, types, and variables is performed in the identifier declaration area, the declaration of input parameters in the inputs declaration area, and the declaration of output parameters in the output declaration area. Declarations of constants, types, and variables are now discussed (the declaration of an input or output parameter is identical to the declaration of a variable, except that it occurs in the input or output declaration areas).

Declarations

A declaration is composed of an name and a description (as shown in figure 4.6). For a simple variable or I/O parameter, this description is a type identifier (simple type, user defined type, or pointer to a type), and for a constant, the description is a value. Declarations are colour-coded according to their type, where the simple types available in HyperPascal are; real (red), integer (orange), string (green),and boolean (blue). Complex types are shown grey (with the exception of arrays and files, which are colour-coded according to their component type), and portholes purple. If a declaration has an undefined description, or is declared to be a variable of an as yet undefined type, it is shown multi-coloured. Once a valid definition is entered (or a type of the same name is

defined in the program unit, or one of its ancestors), the declaration is colour-coded appropriately.

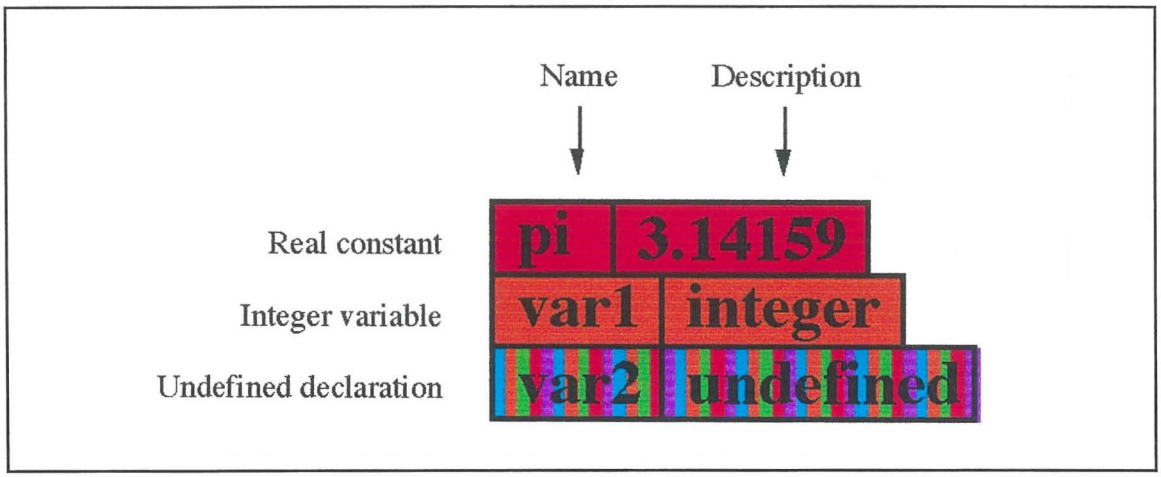


Figure 4.6. Simple Declarations in HyperPascal

In addition to the simple declarations shown in figure 4.6, HyperPascal allows complex declarations of user-defined types such as, records, enumerated types, sets, arrays (of many dimensions), and files. A detailed description of the declaration may be viewed and edited in the declaration area, but these complex descriptions can take up a lot of screen space. Therefore, a reduced representation of the description is shown, and can either be expanded for viewing/editing in place, or can be viewed/edited by traversing a hyperlink from it to a complex type editor (discussed in more detail later). The definition of the different complex types are now discussed in more detail.

User-defined types can be defined in HyperPascal using a declaration format similar to the definition of simple variables. The name of the type being declared is shown in the name field of the declaration, and the description of the complex type is shown in the description field. The name of a type is shown underlined, and the description of the type is represented using an icon with a collapse bar underneath it. The description can either be edited by activating a hyperlink from the icon to the type editor, or by expanding the collapse bar under the icon, and editing the type declarations directly. The different type description icons are shown in figure 4.7.

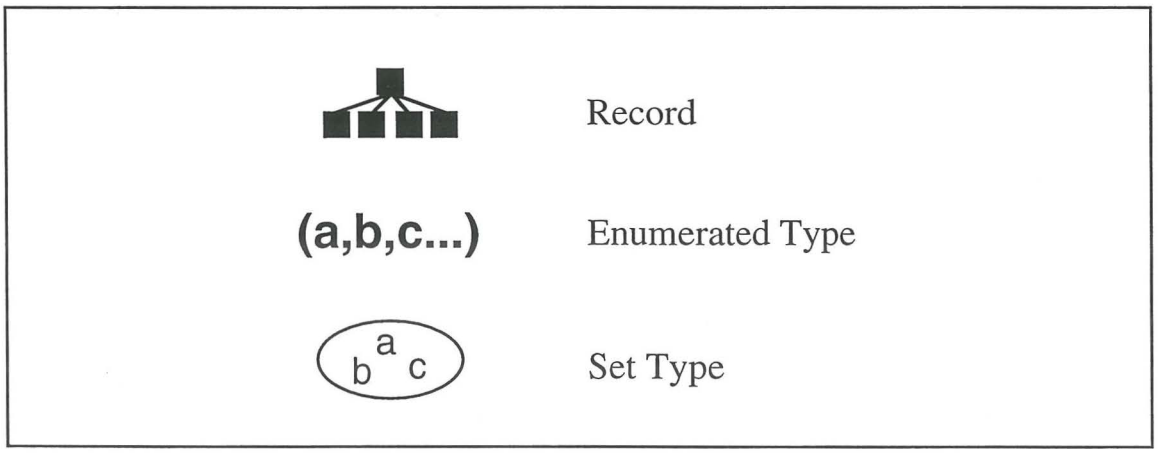


Figure 4.7. The Different Type Description Icons

A record type is a user defined type made up of a number of fields, each of which is a declaration in itself. The appearance of the definition of a record type, and the declaration of a variable of this type, is shown in figure 4.8. Figure 4.8(a) shows the declaration with the reduced representation of the record description, and shows the hyperlinks to the complex type editor. Figure 4.8(b) shows the expanded `emp_data` complex type description, which can be edited by direct manipulation. Note that the address field of this description is a record, and can also be expanded in place.

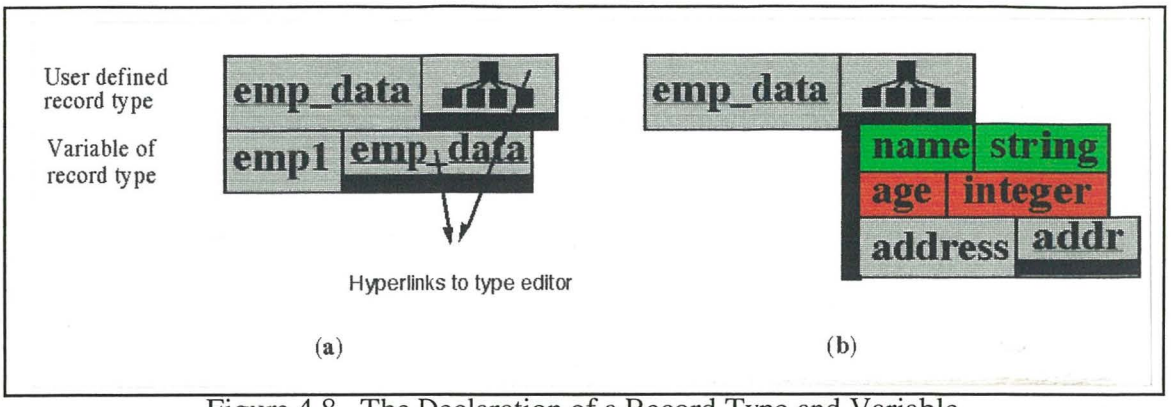


Figure 4.8. The Declaration of a Record Type and Variable

The description of an enumerated type is an ordered list of the values possible for an instance of that type. The definition of an enumerated type and a variable of this type, is shown in figure 4.9. Figure 4.9(a) shows the hidden representation of the declaration description and the hyperlinks to the complex type editor, while figure 4.9(b) shows the expanded (and editable) type description.

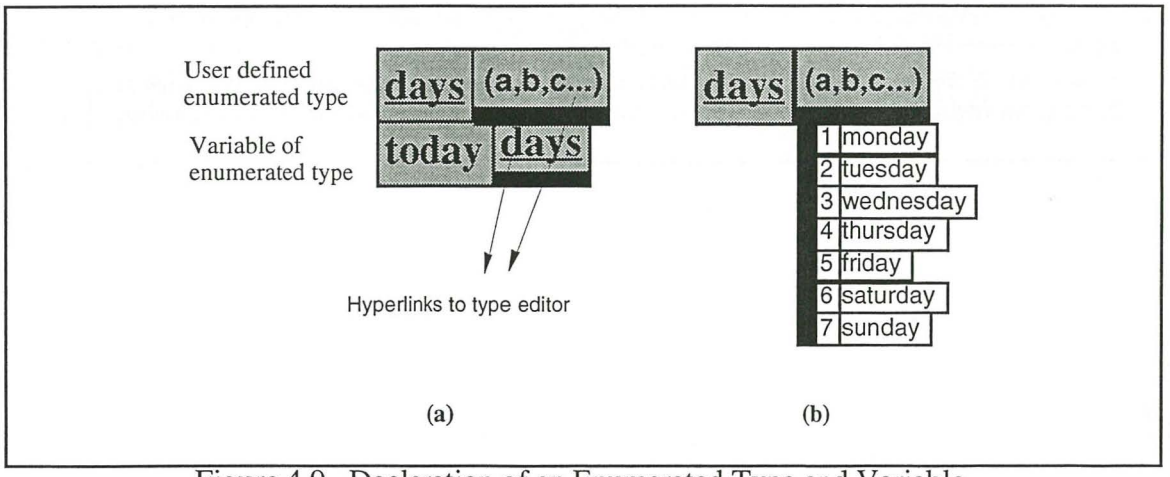


Figure 4.9. Declaration of an Enumerated Type and Variable

The description of a set is a list of values that may be present in a set. This list of values may be the whole of an ordinal type (such as 'integer'), an enumerated type (such as `days` from figure 4.9), a sub-range of these types, or simply a list of possible values. The definition of a number of set types, and a variable of a set type, is shown in figure 4.10. Figure 4.10(a) shows the hidden representation of a type, a variable of that type, and the hyperlinks to the complex type editor. Figures 4.10(b) to 4.10(d) show some of the different formats possible for an expanded type description. Figure 4.10(b) shows the declaration of a set by listing the possible values in the set. Figure 4.10(c) shows the declaration of a set using the whole range of an ordinal type (integer), and figure 4.10(d) shows the use of a sub-range of an ordinal type (monday to friday of the `days` enumerated type).

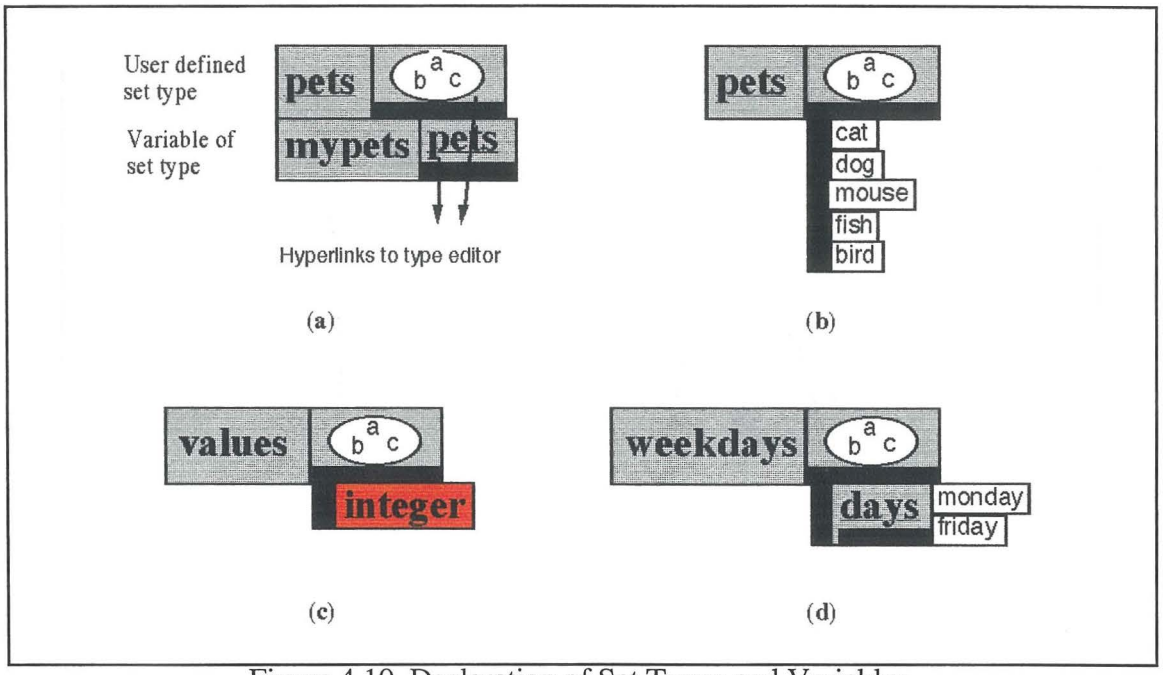


Figure 4.10. Declaration of Set Types and Variables

The description for an array has a type for the components present in the array, a number of dimensions, and maximum and minimum values for these dimensions. The definition of an integer array type, a variable of this type, and an array type with components of a complex type are shown in figure 4.11. Figure 4.11(a) shows the hidden representation of the declarations of an integer array type, and a variable of this type. Figure 4.11(b) shows the expanded type description, displaying the dimensions of the array, and the range of values defined for these dimensions. Figure 4.11(c) shows how arrays of complex types are described by declaring the type to be a complex type, and the type appearing nested inside the array declaration in order to allow it to be expanded in place.

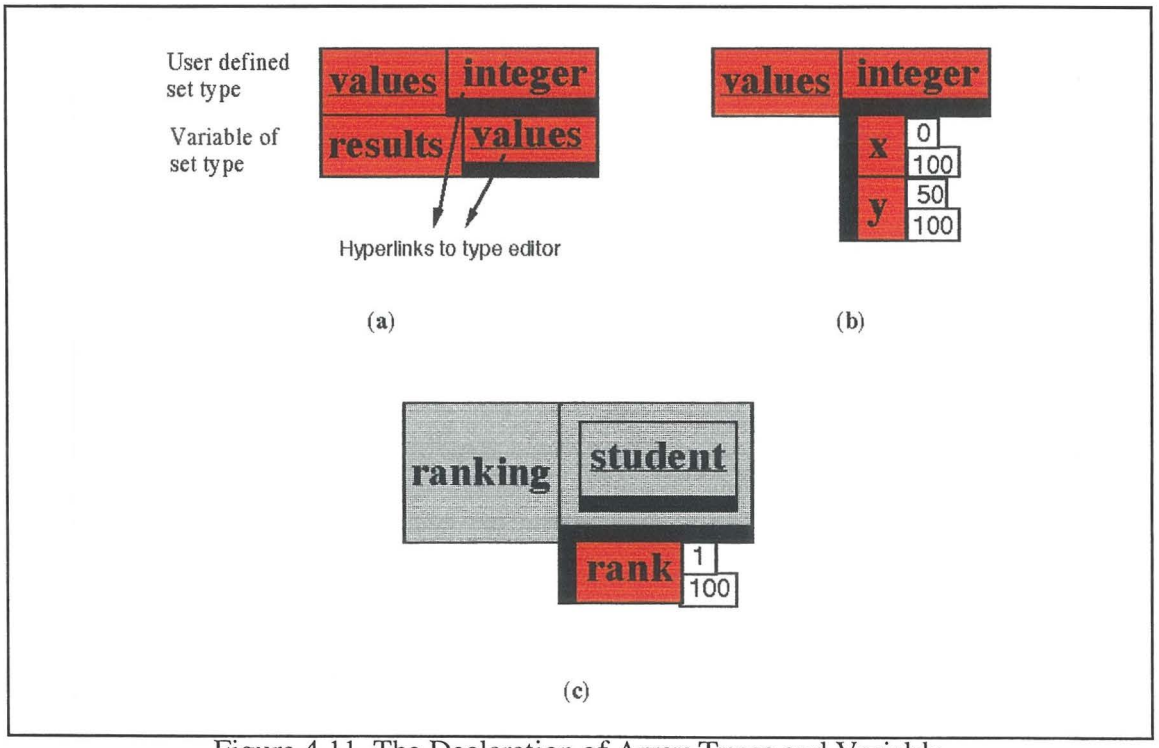


Figure 4.11 The Declaration of Array Types and Variable

The description for a pointer simply indicates that the declaration is a porthole (by displaying a porthole), and the type of value that can be viewed through this porthole. The definition of a porthole type that views an integer value, the declaration of a variable of this type, and the definition of a porthole onto a complex type are shown in figure 4.12. Figure 4.12(a) shows the definition of a porthole type and variable of this type, that views an integer value. Figure 4.12(b) shows the definition of a porthole variable that views a record type. The fields of the record type can be viewed through the porthole by opening the porthole.

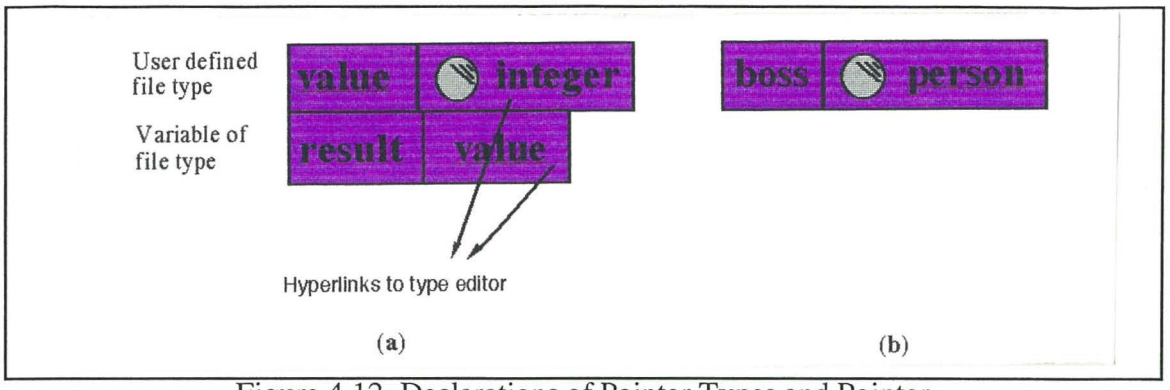


Figure 4.12 Declarations of Pointer Types and Pointer

The description for an file has a type for the data stored in the file, and a filename which specifies the path and name of the file. The definition of an integer file type, and a variable of this type is shown in figure 4.13. Figure 4.13(a) shows the hidden representation of the declarations of an integer file type, and a variable of this type. Figure 4.13(b) shows the expanded type description, displaying the field for storing the file path and name.

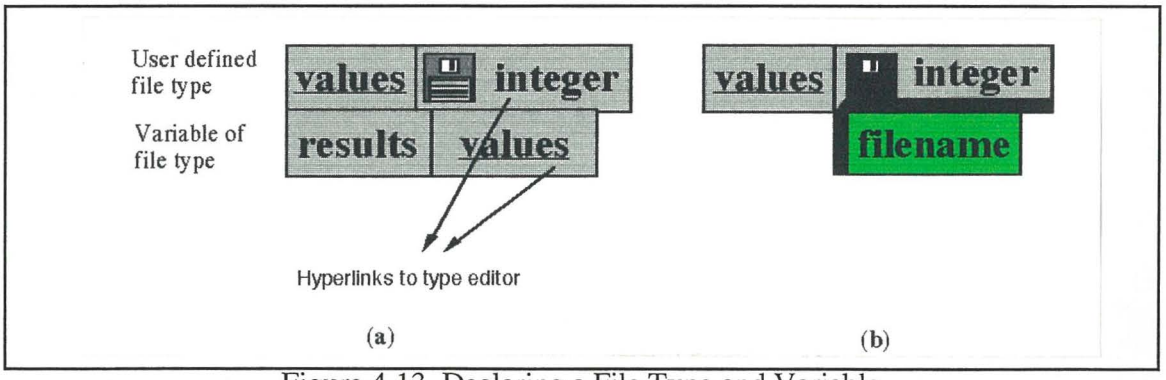


Figure 4.13 Declaring a File Type and Variable

The declaration of input and output parameters is identical to the declaration of variables, except that they are declared in the input or output declaration areas.

We have described the declarations present in the three declaration areas associated with a program unit. The remaining area of a program unit is the unit calling area.

The Unit Calling Area

It can be useful to visualise the calling interdependency of a program at the program unit level (the scope tree view) in order to gain an overview of the function of the program. The unit calling area can either show the units called by the program unit, or the program units that call it. An example showing the units *called by* a particular program unit is shown in figure 4.14.

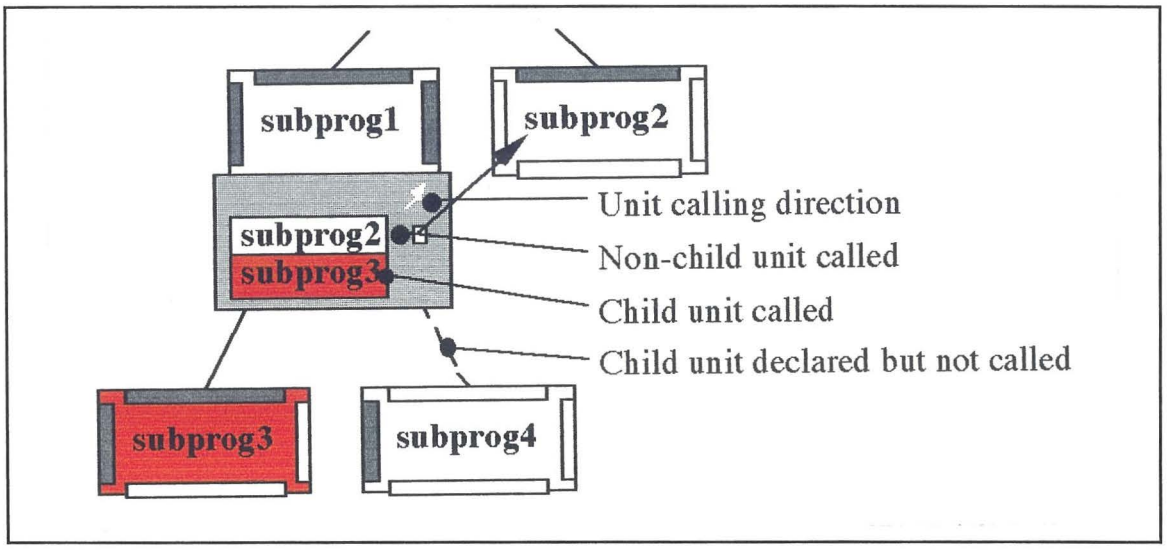


Figure 4.14 The Unit Calling Area

The units called area in figure 4.14 shows that subprog1 calls both subprog3 and subprog2. Subprog2 has a small box next to its name, with an arrow going from it to the corresponding program unit. This is a graphical representation of the fact that subprog1 calls subprog2, and can also be used as a navigational aid to find the corresponding program unit. In figure 4.14, it is relatively easy to find the corresponding unit, but if the unit was not on the screen, locating it would be more difficult. In this case, a hyperlink from the arrow can be activated in order to travel to the location of the called unit. Activating the hyperlink again scrolls the view back to the calling unit. This arrow may be hidden and shown again.

The calling of subprog3 in figure 4.14 has no small box or arrow associated with it because (since subprog3 is a child of subprog1), subprog3 should be located close to subprog1, and therefore relatively easy to find. Every child unit that is called by its parent is connected to the parent unit with a solid connector. Those children that are not called by their parent are connected to the parent unit with a dashed connector.

The direction of calling displayed by the unit's calling area is shown by the arrow in the top right-hand corner of the unit calling area. The arrow pointing up and to the right (the units called by this unit) in figure 4.14 can be changed to an arrow pointing down and left (the units that call this unit). The names in the unit calling area then become the names of the units that call this unit, and the arrows to the corresponding units change direction.

It would be useful if we could use this calling, and called by, information to travel to the appropriate location in the actions of the program to view this information in detail. This navigation is performed in HyperPascal using hyperlinks.

Hyperlinks from the Scope Tree View

The scope tree view has hyperlinks defined to the complex type editor, action tree views for the program units, and back to itself. The background of a program unit contains a hyperlink to the action tree view of that program unit, with the view focussed on the program unit (the program unit in the middle of the view window). This, and other hyperlinks from the scope tree view, are shown in figure 4.15.

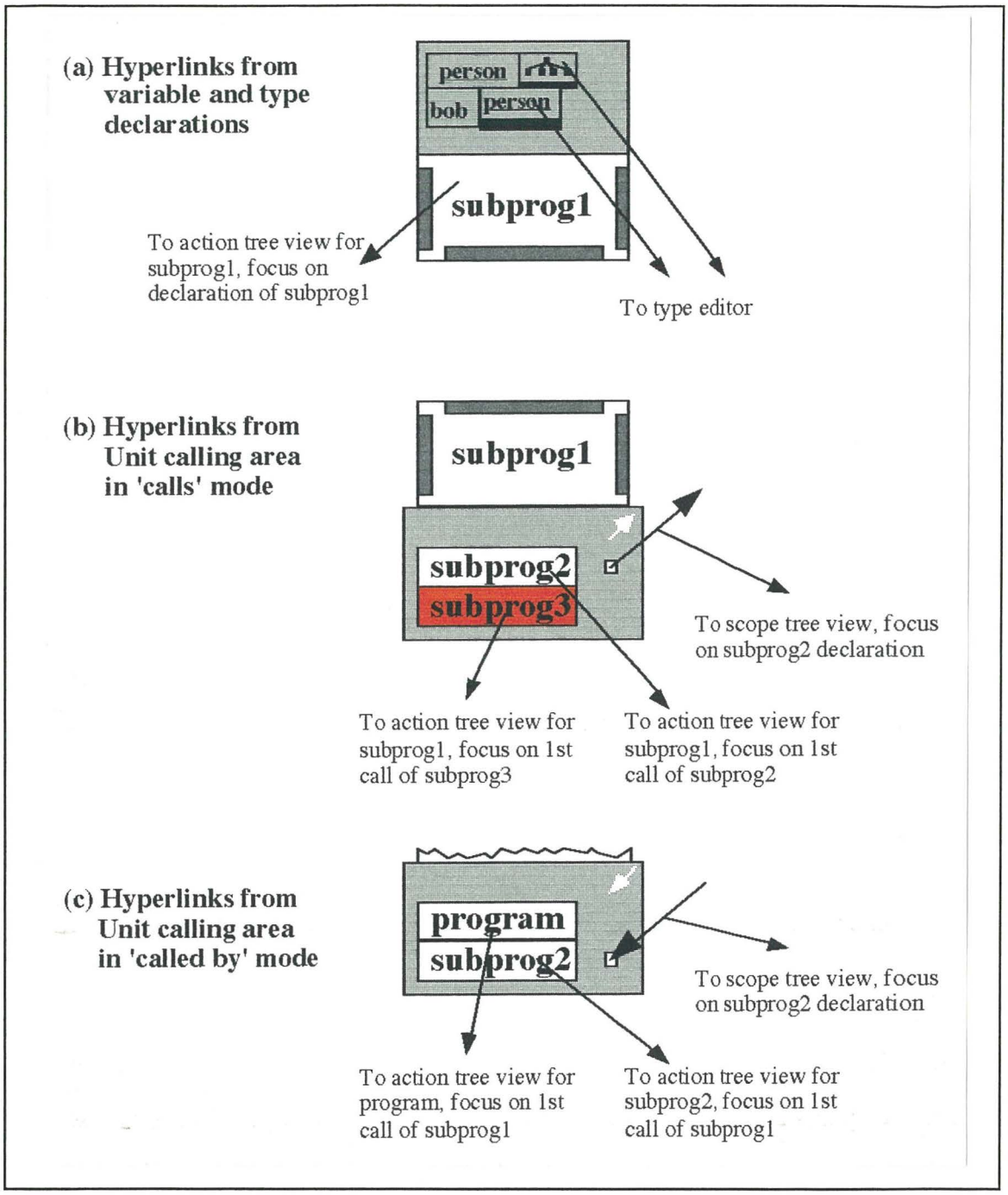


Figure 4.15 Hyperlinks from the Scope Tree View

Figure 4.15 shows the hyperlinks from the scope tree view, with figure 4.15(a) showing the hyperlinks from declarations. Figures 4.15(b) and 4.15(c) show the hyperlinks from the unit calling area while in 'calling' and 'called by' mode respectively. In both cases activating the hyperlink on the arrow from (or to) another program unit takes the user to another location in the scope tree view (focussing on the program unit at the other end of the arrow). Activating this hyperlink again, takes the user back to the location focussing on the original program unit.

The destination travelled to by activating the hyperlink from a program unit name in the unit calling area is dependent on the current mode. If the unit calling area is in 'calls' mode, the destination of the hyperlink activated is to the action tree view of the current program unit, with the focus on the first call to the named unit. If the unit calling area is

in 'called by' mode, the destination is to the action tree of the named calling unit, with the focus on the first call to the current program unit. In both cases the hyperlink is to the action tree of the calling unit.

In this section we have described how the scope tree represents program units, declarations of types, constants, variables and input/output parameters, and how to navigate both within the scope tree view, and to the action tree, where the processing part of a program is represented. The action tree is described in section 4.3.

4.3. The Action Tree

The constructs of a procedural programming language can be placed into two broad classes, those that manipulate data, and those that control the flow of execution of the program. The distinction between these two classes of construct, and the individual constructs can be aided, using a graphical notation to represent them using different shaped icons. The use of different icons to represent different programming constructs enables the constructs to be recognised quickly. Doran and Tate's (1972a, 1972b) structure charts, a paper-based notation that uses different shapes to represent different programming constructs, were identified as a suitable basis for a formal graphical language for specifying data manipulations.

Structure Charts

Doran and Tate (1972a, 1972b) invented a notation which used a tree-structured chart with different shaped nodes to represent an algorithm. The tree-structured chart shows explicitly the tree-structured nature of programs, with different shaped (and easily recognisable) nodes in the tree representing different control constructs. In the notation, statements and comments are shown in rectangular boxes, choices in hexagonal boxes, and loops in boxes with rounded corners. An example of a structure chart is shown in figure 4.16 (Doran and Tate, 1972a).

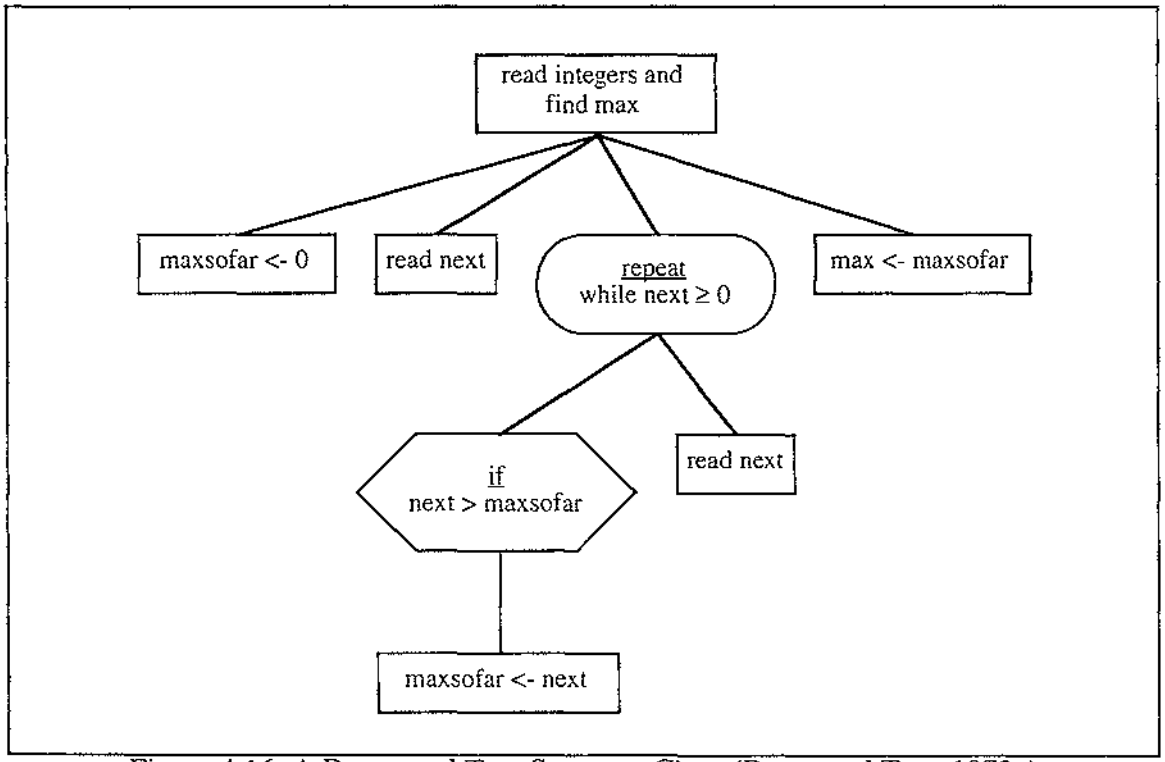


Figure 4.16 A Doran and Tate Structure Chart (Doran and Tate, 1972a)

The nodes in a structure chart are traversed from left to right in a depth first manner. The two simple types of node, comment nodes and statement nodes, do not affect this traversal order. Comment nodes are for documentation only, and therefore represent simply an intermediate node in the traversal order. Statement nodes must be terminal nodes and, while they can contain many statements, have no order imposed on the statements contained in a single statement node. The other two types of node, loop nodes and choice nodes, do affect the traversal order, depending on the condition they contain.

A loop node repeats the traversal of its sub-trees a number of times depending on the result of a condition in the loop node. There are three possible types of loop node, a loop-until, a loop-while, and a loop-for. The loop-until type and the loop-while type differ in the order of testing the condition, and allowing the traversal of their sub-trees. The loop-until allows the traversal of its sub-trees, then tests the condition. If the condition evaluates to true, the traversal continues in the loop's parent, otherwise the sub-trees are traversed again, and the condition re-evaluated. This traversal of the loop's sub-trees continues until the condition evaluates to true.

The loop-while differs from the loop-until by evaluating the condition before (possibly) allowing its sub-trees to be traversed. If the condition evaluates to false, the sub-trees are not traversed, and the traversal continues in the loop's parent. If the condition evaluates to true, the sub-trees are traversed, and the condition evaluated again.

A choice node is used to conditionally traverse one of a number of sub-trees. Each of the sub-trees has a condition attached to it, which must evaluate to true for that sub-tree to be traversed. The conditions on each sub-tree are evaluated in a left-to-right order, and the first sub-tree whose condition evaluates to true, is traversed. A maximum of one sub-tree is traversed, and after its traversal, the traversal continues in the choice's parent.

Doran and Tate's structure chart notation allows a general specification of a program in terms of a hierarchy of control structures and program statements. It is effective both in separating control and data manipulation statements, and in depicting the scope of control. The notation has omitted certain elements necessary for the complete specification of a program (such as declarations), becomes large and complicated to modify as programs become large, and does not make enough use of visual expression for use as an effective visual programming language. The notation has been modified for use in the action tree of HyperPascal in order to enable the complete specification of a program (when used in conjunction with the scope tree), to allow information hiding, use colour-coding for redundant re-coding, and to make the notation more visually expressive.

The HyperPascal Structure Diagram Notation

The action tree notation of HyperPascal is based on Doran and Tate structure charts (1972a, 1972b), and shares many of the components (with modifications) of the original notation. The action tree can be composed of comments, choices, loops, withs, and action sequences. The action tree is headed by a program unit declaration, which is also present in the scope tree. Any modification of this program unit in the action tree will be reflected in the scope tree, and *vice-versa*. The action tree representation of a subprogram (the same actions as shown in figure 4.16, with an appropriate program unit declaration added) is shown in figure 4.17.

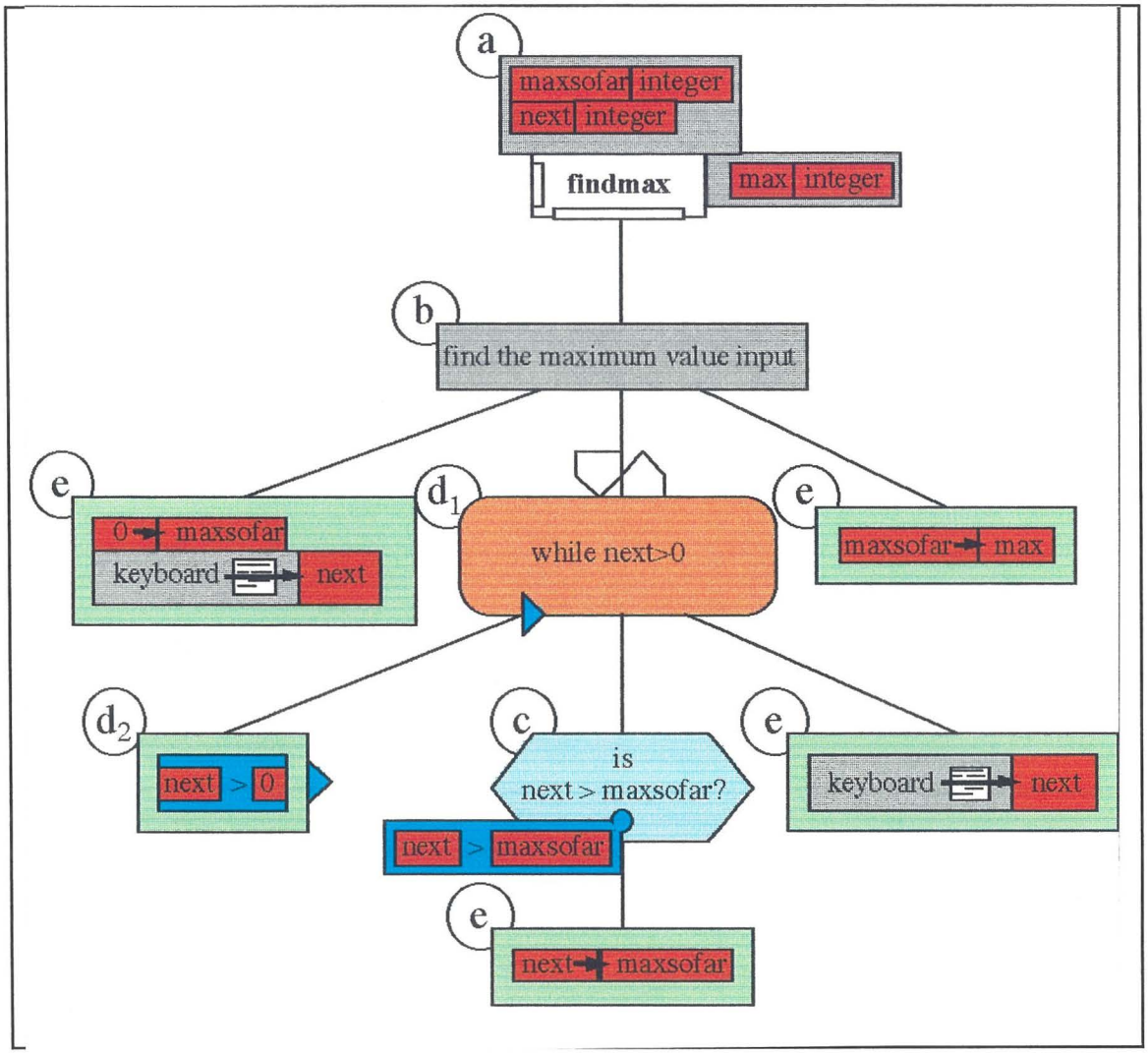


Figure 4.17 An Action Tree in HyperPascal

Figure 4.17 shows an action tree with labels associated with each component. These components are the program unit declaration (figure 4.17(a)), the comment component (figure 4.17(b)), the choice component (figure 4.17(b)), the loop component (figure 4.17(d_1)) and its associated condition sequence (figure 4.17(d_2)), and the action sequence component (figure 4.17(e)).

With the exception of the program unit declaration (which was discussed previously), the components listed above (and the 'with' component, which is not present in figure 4.17) are now discussed in detail.

Comments

Comments are rectangular, grey coloured boxes which contain a short comment, and are used to document a HyperPascal program. Associated with a comment box is a more extensive comment which can be viewed and edited by activating a hyperlink from the comment box to an extended comment editor. Comment boxes are also useful for grouping a number of sub-trees into a single sub-tree, which is necessary when attaching multiple sub-trees to a single choice condition. A comment box is shown in detail in figure 4.18.

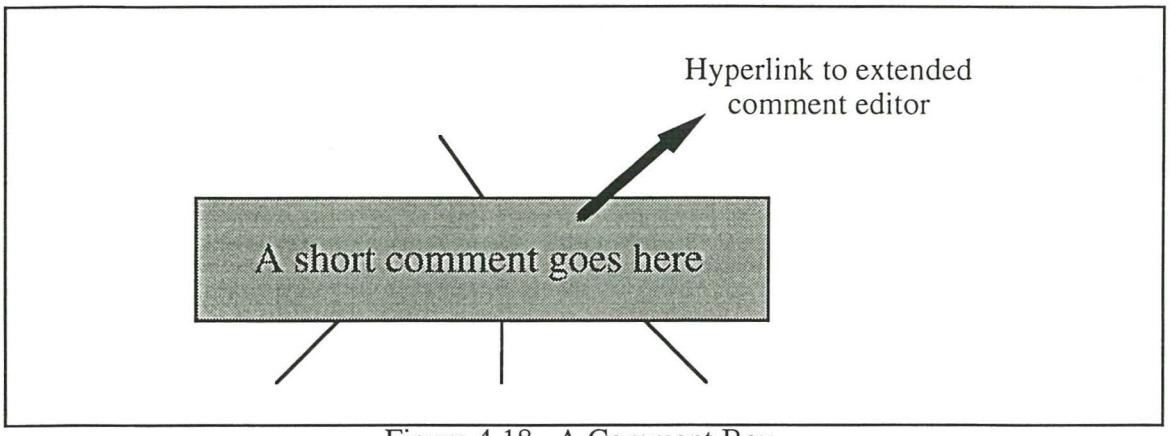


Figure 4.18. A Comment Box

Action Sequences

The purpose of a program is to manipulate data in some way. Data manipulations (for procedural programming) are performed by assignment statements or procedure invocations, which are usually present in small (sequential) groups, under the influence of some control structure. HyperPascal represents a sequential group of data manipulation actions as an *action sequence*.

Action sequences are rectangular, light green coloured boxes that perform the data manipulation in a HyperPascal program. They are the terminal nodes of the action tree, and contain a list of actions which are performed in sequence, from top to bottom. An example of an action sequence for swapping the values of two integer variables is shown in Figure 4.19.

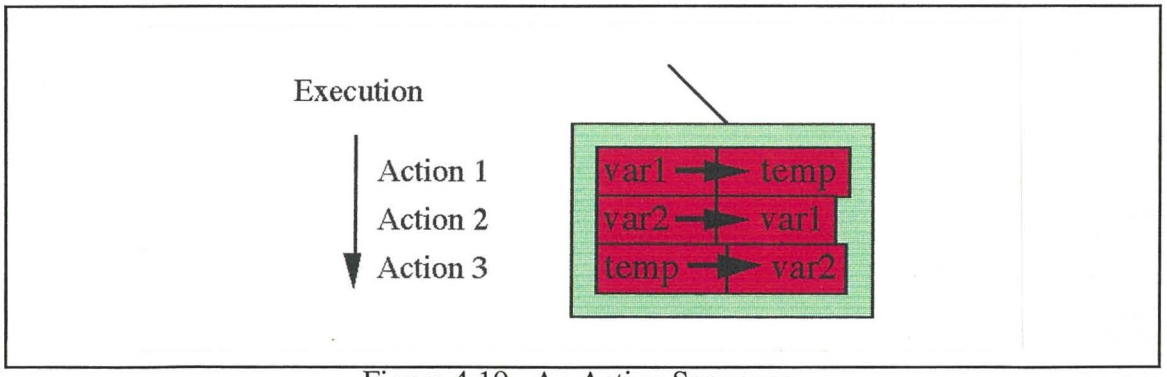


Figure 4.19. An Action Sequence

In HyperPascal, an action generally takes the form of an assignment of the value from a donor identifier (or expression) to an acceptor identifier. If the assignment is a valid one (the types of the two sides of the assignment are compatible), the colours of donor and acceptor fields meet in the middle, and an assignment arrow is shown. In HyperPascal, assignments are analogous to information flow, which for English speakers at least, seems most naturally represented by a flow from left to right. Some valid assignment actions are shown in figure 4.20(a). If, however, the types on each side of the assignment are not compatible, the colours of the donor and acceptor fields retreat from each other, and no assignment arrow appears (shown in figure 4.20(b).

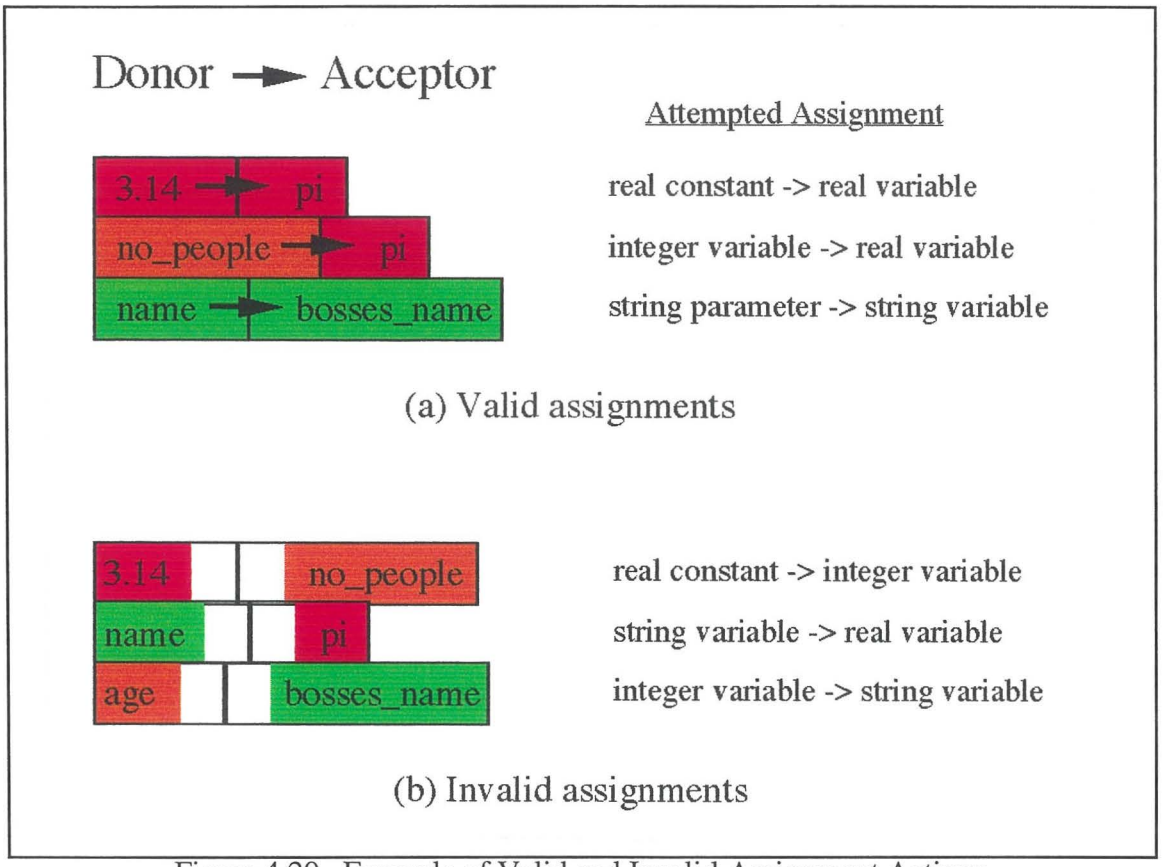


Figure 4.20. Example of Valid and Invalid Assignment Actions

In general, values from the donor field can be assigned to variables or parameters in the acceptor field if their types match, or an integer value in the donor field is being assigned to a real acceptor variable or parameter. The acceptor field of an assignment can only contain variables or parameters, whereas the donor field may contain variables, parameters, constants, or expressions. The assignment actions in figure 4.20 are examples that use simple types, but assignments can also be performed using complex types (as summarised in Figure 4.21).

Input and Output

In traditional programming languages, the input of data from the keyboard and the output of data to the screen are usually performed using special commands. It is possible to represent the movement of data which occurs during I/O by casting I/O in the form of assignment actions. Note that a similar convention, which will be described later, has been adopted for the passing of parameters into and out of subprograms. The input and output actions should therefore should be of a format consistent with other assignment actions.

In a text-based system, the input and output of data from the keyboard or to the screen is usually associated with formatting information. This formatting information can be in the form of prompts for input, formatting of output, or even simple layout information. It is important, for the design of the overall appearance of the input and output of the program, to be able to view this information as it will appear on screen, in relation to the data being input or output. However, the representation of this formatting information should not obscure the underlying input or output assignment.

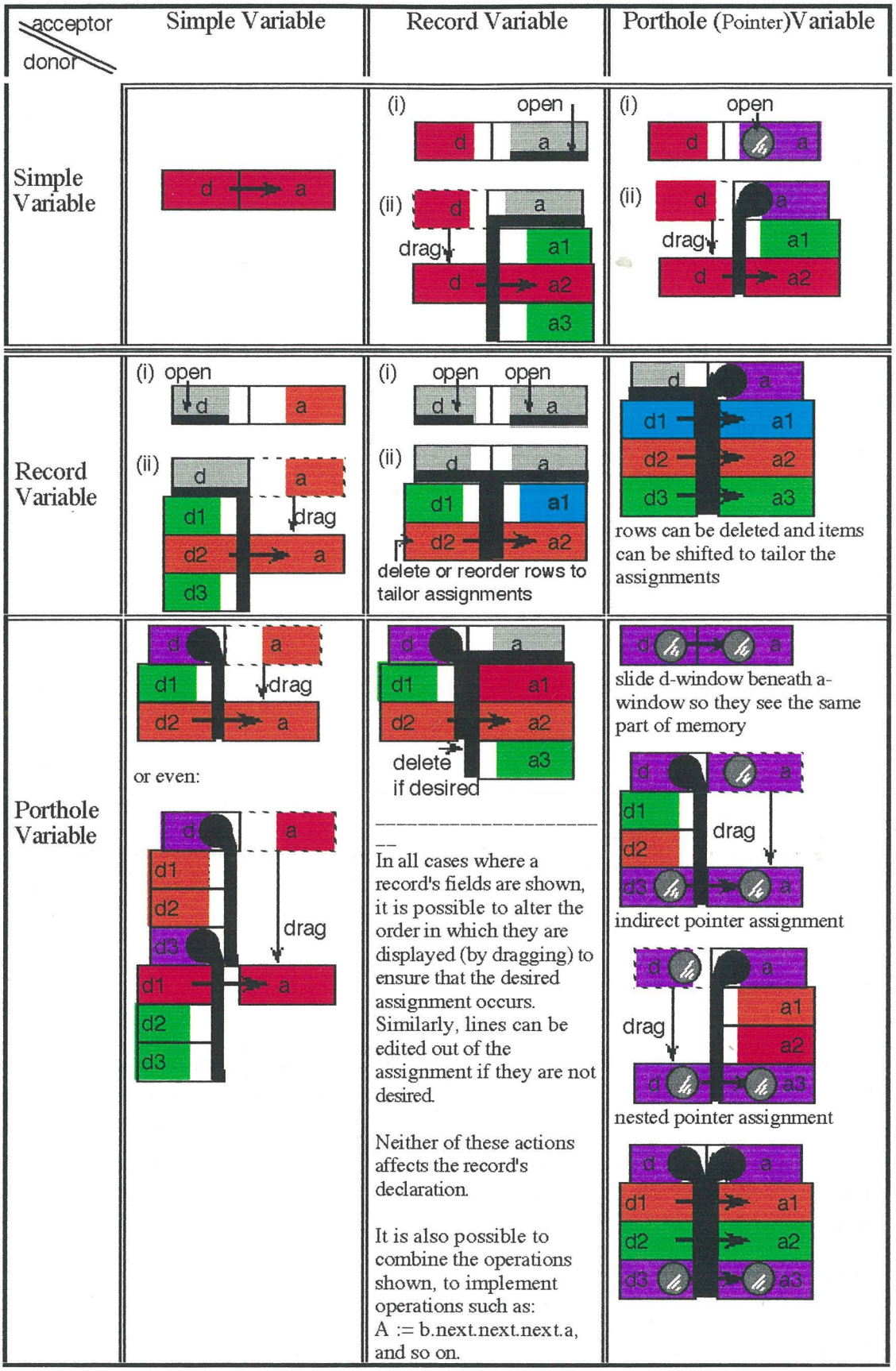


Figure 4.21. Complex Type Assignments in HyperPascal (Lyons *et al.*, 1993)

Input is performed in HyperPascal by assigning values from the keyboard (or a file) to a variable or parameter. Output is similarly performed by assigning values to the screen (or a file). Input from the keyboard, and output to the screen, can be associated with formatting information described using the *forms windows* (which will be described in Section 4.4). The formatting information for an input or output (or a sequence of inputs and outputs) is not shown directly in the action tree (in order not to obscure the underlying functionality), but can be accessed from the input/output action via a hyperlink. The appearance of input assignments from the keyboard and output assignments to the screen are shown in figure 4.22.

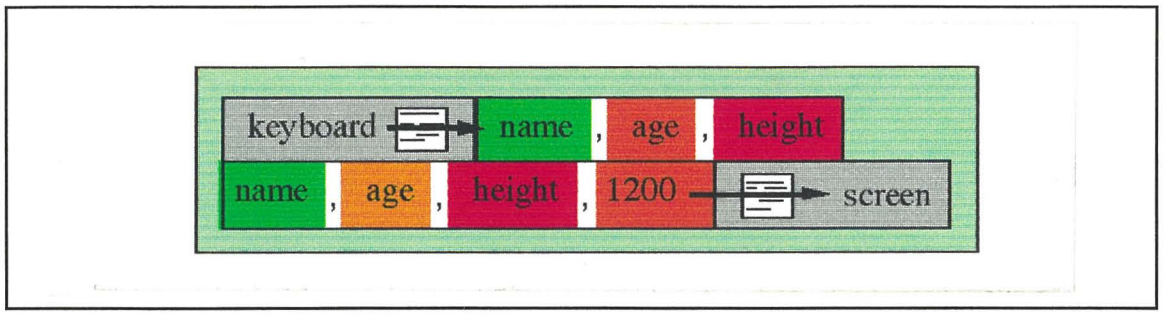


Figure 4.22. Input and Output Assignments

Figure 4.12 shows how input from the keyboard is assigned to a comma separated list of variables and/or parameters, and how a list of variables, parameters, or constants can be assigned to the screen for display. The icon represents the formatting information, and the assignment arrow goes through this formatting "filter".

This approach allows formatting information to be associated with input and output data in a non-intrusive way (through hyperlinks to forms windows), and shows the input and output clearly and consistently in the action sequence.

So far we have shown how to input and output data, and how to assign data values from an identifier or constant to another identifier. To implement more than trivial actions, a language must be able to perform manipulations (such as mathematical transformations) of data. Complex manipulations of data can be specified using expressions.

Expressions

Conventional mathematical notations have traditionally been translated into a linear text string using an arbitrary operator precedence, and parentheses to modify this precedence. This approach has proved effective for representing expressions in computer programs, but with long expressions it is difficult to match parentheses, and therefore the expression structure becomes harder to comprehend. The representation of an expression as a linear string of text also creates difficulties when attempting to add a representation of the expression's type structure.

The structure of expressions and sub-expressions form a tree, in which the non-terminal nodes are operators, and the terminal nodes are values. Each terminal node in the tree has a type associated with it, and the non-terminal nodes have a type resulting from applying the operator to its sub-expressions. The expression tree for a simple expression is shown in figure 4.23.

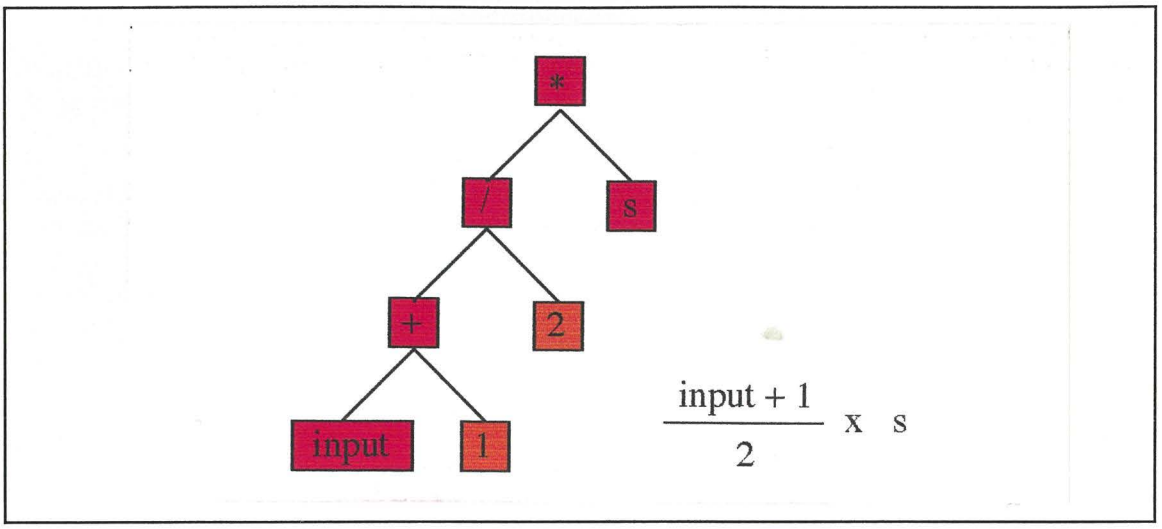


Figure 4.23. An Expression Tree and Equivalent Expression

While the tree notation in figure 4.23 clearly shows the expression structure, the type structure, and the operator associativity (thus eliminating the need for parentheses), it uses space inefficiently. Efficient use of screen space is important when dealing with expressions in a visual programming language, because there are many expressions present in a typical program, and their representation should not conceal the control structure and actions. A more compact representation of an expression tree is therefore needed for use in HyperPascal.

HyperPascal uses a box-within-box notation to represent an expression tree. A non-terminal node is represented by a box that (for binary operators) contains two other boxes separated by an operator, whereas a terminal node is represented by a box containing either the name of an identifier, or a value. A non-terminal node for a unary operator is represented by a box containing the operator and another single box. Operators in HyperPascal have no precedence, and their associativity is shown using the box-within-box notation. Each box has its background colour-coded according to the type associated with the node it represents. Figure 4.24 shows an expression tree and its corresponding representation in HyperPascal.

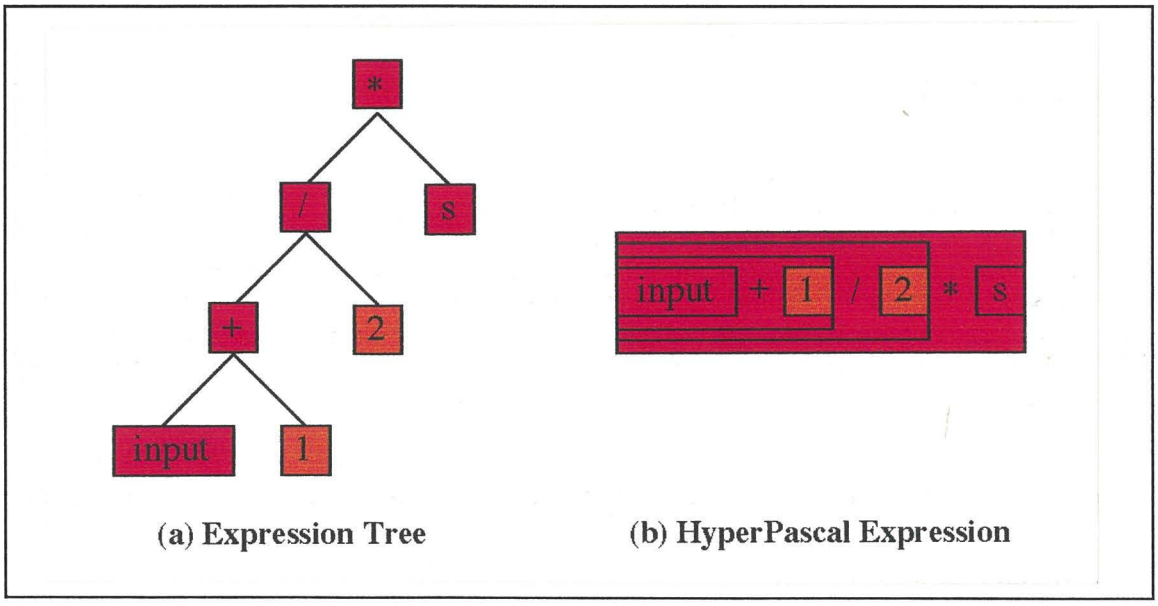


Figure 4.24. An Expression Tree and its Representation in HyperPascal.

The nested box representation of expressions shown in figure 4.24 is drawn automatically as an expression is entered or edited. The entering and editing of expressions is discussed later, in section 4.5.

The expression shown in figure 4.24 uses variables of simple types, but expressions using complex types are also supported. Fields of records and pointers can be used in expressions by expanding the record or pointer to show the fields, then aligning the operator with the appropriate field. Fields not used in the expression may be hidden to reduce the size of the expression. An expression that uses record and pointer fields is shown in figure 4.25.

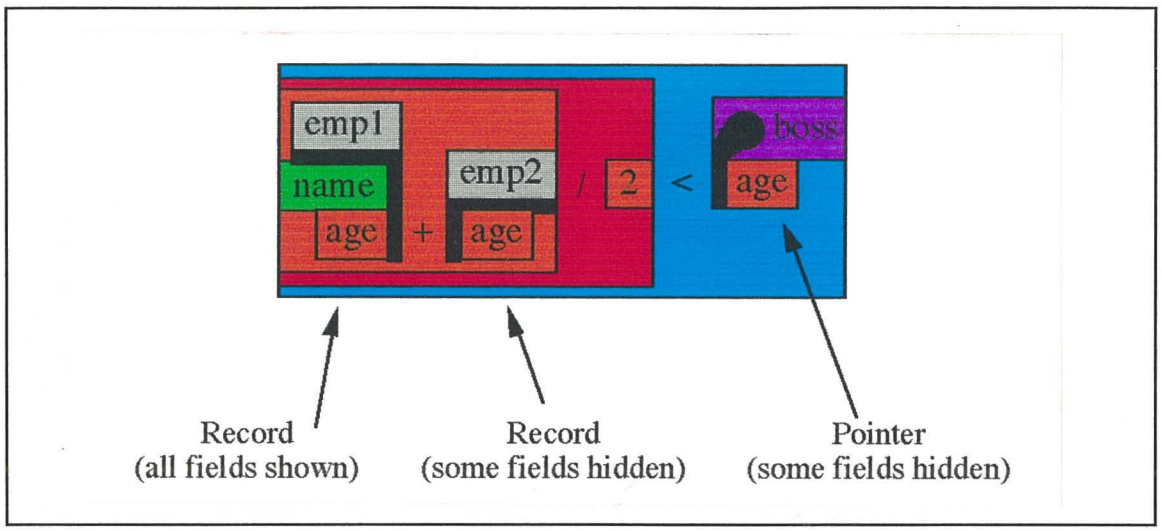


Figure 4.25. An Expression that uses Record and Pointer Fields.

It is sometimes useful to use fields of the same record (or pointer) in different parts of an expression. To facilitate this, HyperPascal uses a connected dot notation to represent the field's location in the expression. An example of the use of the connected dot notation in an expression is shown in figure 4.26.

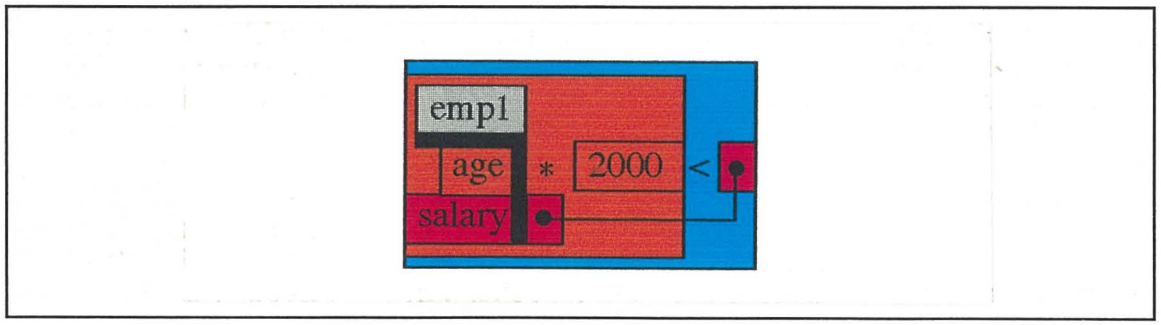


Figure 4.26. An Expression that uses more than one of a Record's Fields

As well as allowing complex data manipulations, HyperPascal expressions can include the invocation of subprograms.

Invoking Subprograms

A subprogram in HyperPascal can have multiple inputs and output parameters, and possibly have a return type associated with it. It is convenient therefore, to represent the inputs and outputs of a subprogram as pseudo-records with their fields being the input and output parameters, and also to colour-code the background of the subprogram according to its return type. The pseudo-records for the inputs and outputs may be either expanded or reduced, to show or hide the formal input and output parameters. The representation of some simple subprogram invocations are shown in figure 4.27.

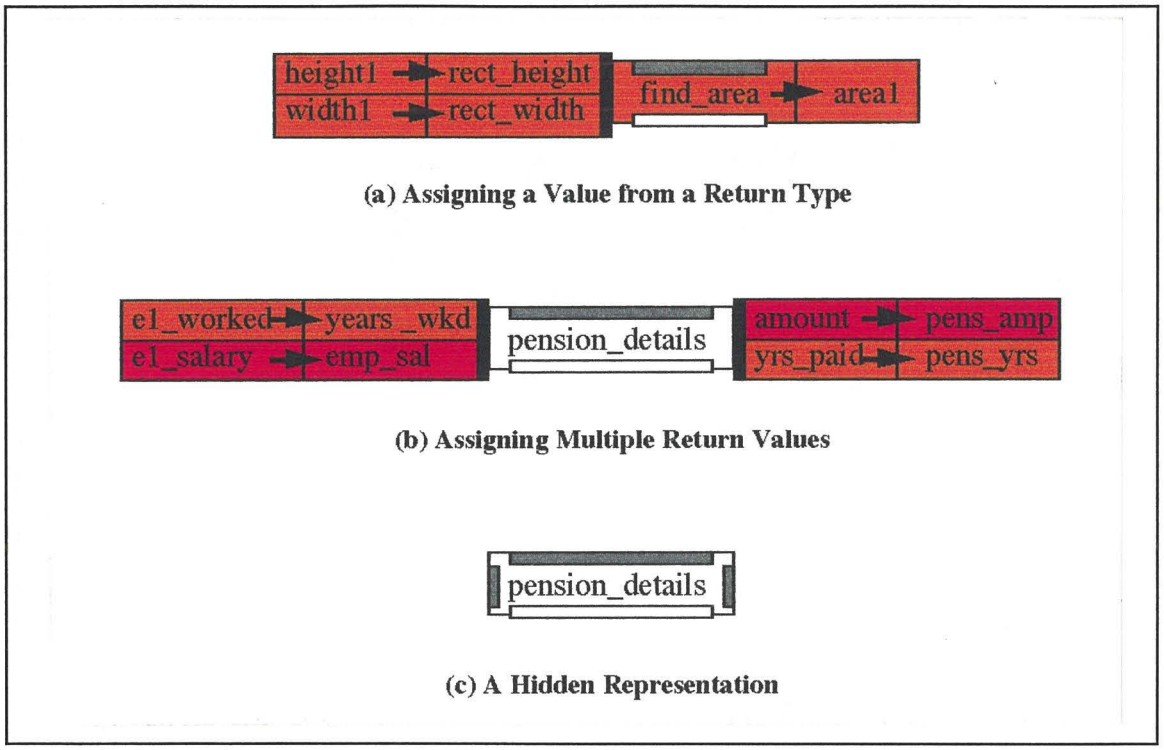


Figure 4.27. The Invoking of a Subprogram

Figure 4.27(a) shows a subprogram being invoked, with variables being assigned to the input parameters, and a returned value being assigned to another variable. Figure 4.27(b) shows the fully expanded version of a subprogram invocation, with variables being assigned to the input parameters, and the output parameters being assigned to variables. Figure 4.27(c) shows the reduced version of the same subprogram invocation as that in figure 4.27(b).

The representation of the subprogram in an invocation is the same as the subprogram declaration in the scope tree, and its heading in the action tree. This means that the other collapse bars can be expanded to show (and enable the editing of) the identifier declarations of the subprogram, and to show the subprograms it calls, and is called from.

The subprogram invocations shown in figure 4.27 used only simple variables providing the inputs and receiving the outputs of the subprograms. Subprogram invocations can have expressions as actual input parameters, which are evaluated, then the value assigned to the formal parameters. Subprograms can also be used in expressions, providing one or more values to the expression. Figure 4.28 shows a number of subprogram invocations that illustrate the use of expressions as inputs to a subprogram, and the use of subprograms for providing values to an expression.

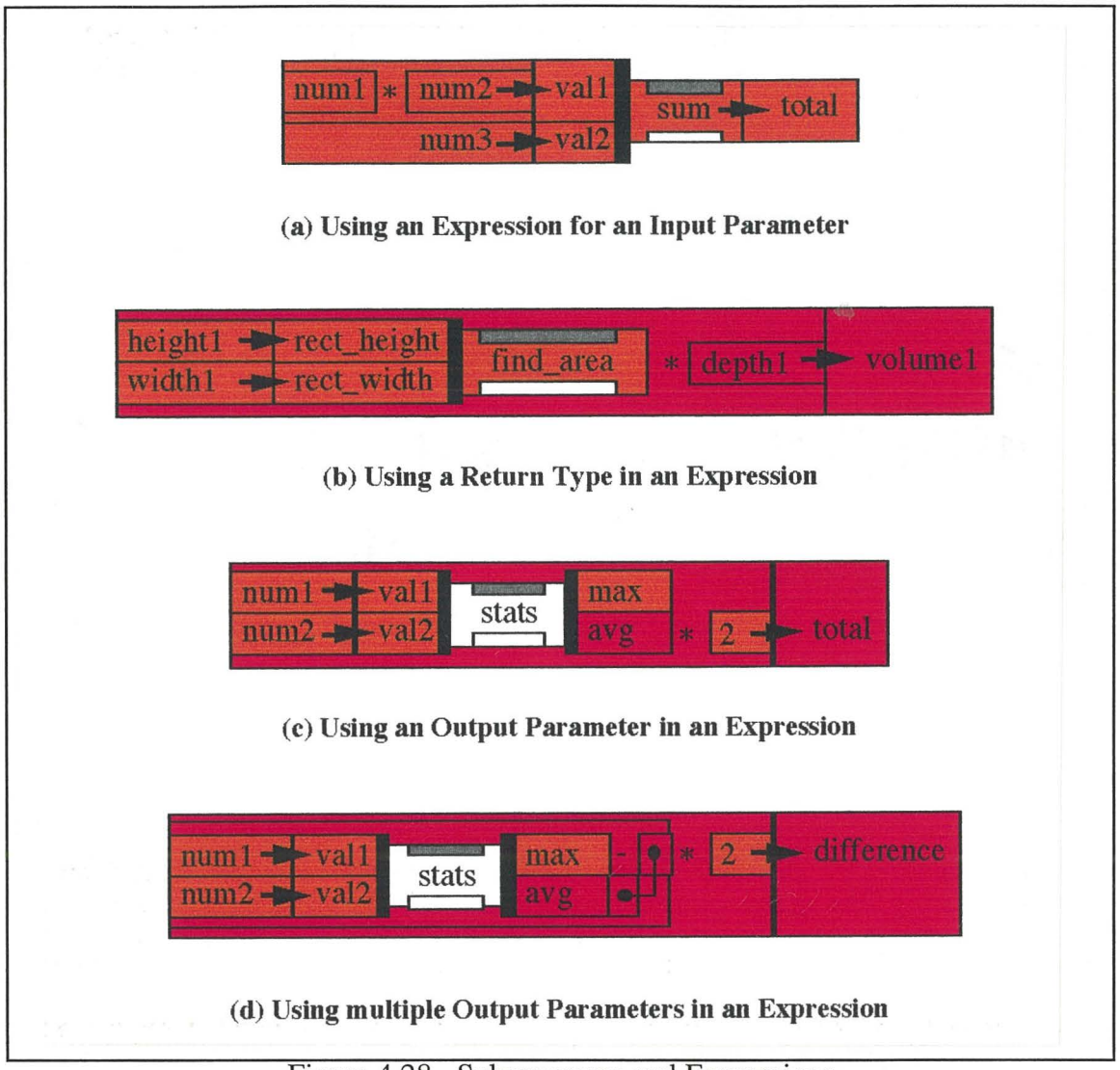


Figure 4.28. Subprograms and Expressions

Figure 4.28(a) shows a subprogram with two input parameters, one of which is assigned the result of an expression. The expression is evaluated, then the result is assigned to the input parameter, and the subprogram invoked. Figure 4.28(b) shows a subprogram with a return type being used in an expression as a value. Figures 4.28(c) and 4.28(d) shows the use of a subprogram with output parameters in an expression. Figure 4.28(c) shows how, when only one output parameter is used in the expression, the expression is aligned with that output parameter. Figure 4.28(d) shows the use of the connected dot notation to enable multiple output parameters of a subprogram to be used in an expression.

Sections of code in programs often manipulate several fields of a particular record. It is tedious and space-consuming to represent the record as well as the desired field(s) in each of the manipulations concerning the field(s). A *with* component in HyperPascal allows the programmer to reference the fields of a record (or pointer to a record) without representing the record each time.

Withs

With components are rectangular, light-purple coloured boxes that contain a reference to a record structure. The *with* component specifies a record name, and all references to identifiers in the children of the *with* component are checked to see if they match the

fields of this record. If an identifier matches a field of the specified record, it is assumed to be that field, but if an identifier doesn't match a field of the record, it is treated as a normal identifier. An action sequence is shown in figure 4.29(a), and its equivalent representation using the 'with' box is shown in figure 4.29(b).

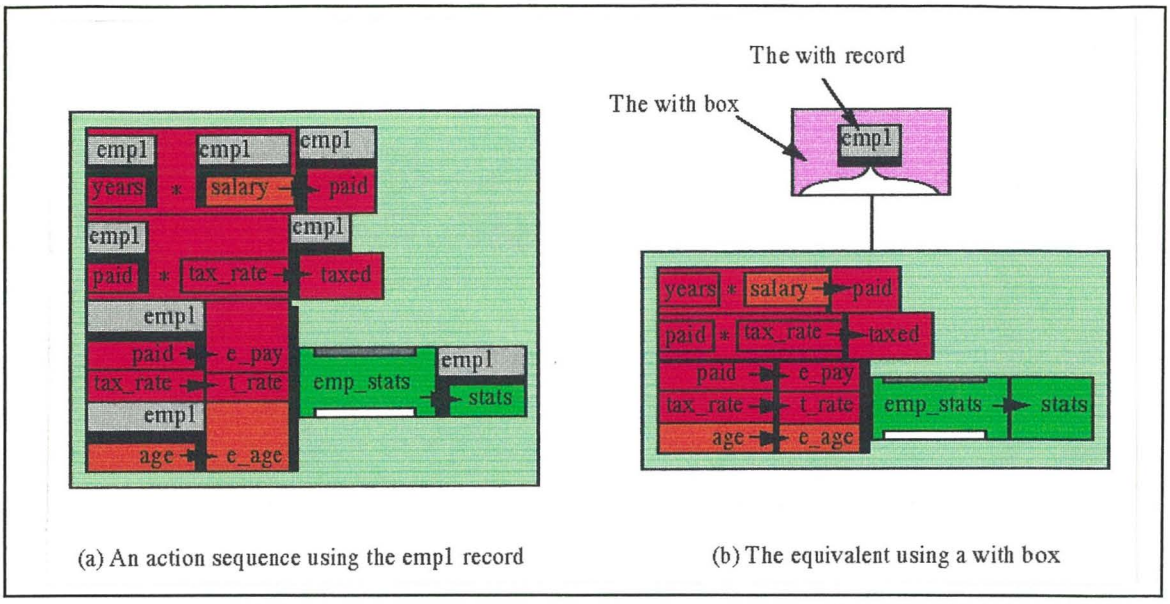


Figure 4.29. The Appearance and use of the With Box

The two action sequences shown in figure 4.29 represent the same functionality. In figure 4.29(b), the assignment actions and expressions using the fields of the "empl" record are vertically reduced in size by approximately half. The overall representations of figure 4.29(a) and 4.29(b) are approximately the same size in this case, but for larger sections of manipulations involving the record's fields the space saving is more apparent.

The components so far discussed include action sequences, which contain a sequence of actions for performing data manipulations and invoking subprograms in HyperPascal, and execute in a strictly sequential manner. Comment boxes and with boxes, have also been discussed. Neither of them perform any data manipulation, and they have no influence on the sequential execution of a program. Since procedural programs typically do not execute in a strictly sequential manner (they typically use conditional execution and repetition), other constructs are needed to impose control on the execution of actions in a program. The first of these constructs discussed is the *choice* construct.

Choices

Choice constructs enable a program to choose whether or not to execute a section of code, or to choose one of two or more sections of code, based on the result of the evaluation of a condition, or conditions.

Choices are hexagonal, light blue coloured boxes that contain a short comment, and one or more condition disks arranged along the bottom of the box. A condition disk is a hidden representation of a condition attached to a sub-tree, and can either contain a valid condition (coloured dark blue), or can contain a default (else) condition (coloured white). Condition disks can be expanded to show (and edit) the conditional expression contained. Figure 4.30 shows the choice box and condition disks in detail.

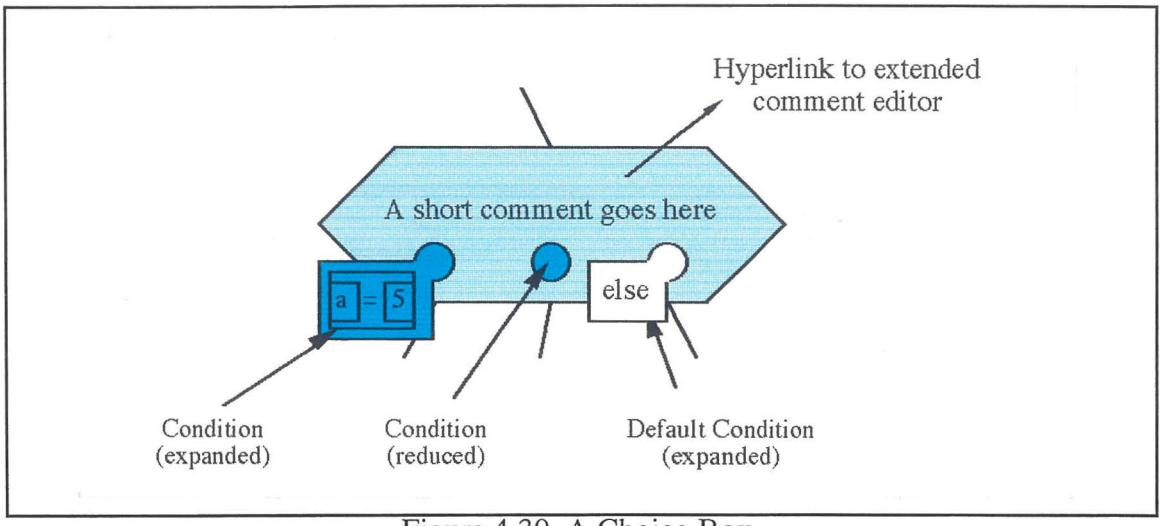


Figure 4.30. A Choice Box

The conditions in a choice box are evaluated in left-to-right order, and the sub-tree executed is that associated with the first condition that evaluates to true (default conditions always evaluate to true). If none of the conditions evaluate to true, none of the sub-trees are executed. Once a condition has evaluated to true, and its sub-tree executed, the conditions to its right are not evaluated. This use of multiple conditions in a choice box allows us to use it to represent the different types of conditional constructs in a program. Figure 4.31 illustrates how some common conditional constructs are represented using the choice box.

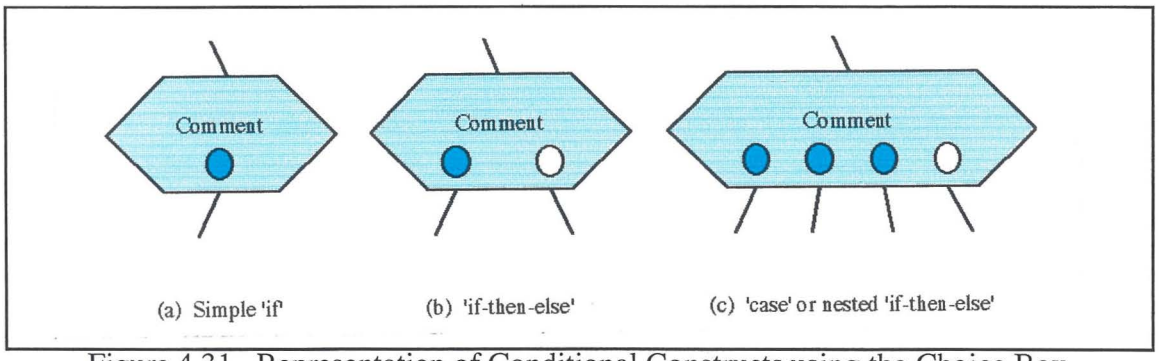


Figure 4.31. Representation of Conditional Constructs using the Choice Box

The HyperPascal equivalent of a simple 'if-then' construct is a choice box with one condition disk containing the condition. An 'if-then-else' construct is represented by a choice box with two condition disks, the left-most containing the 'if' condition, and the right-most containing the default condition. A 'case' (or nested 'if-then-else') construct is represented by a choice box with many condition disks, where an 'otherwise' alternative is represented by having the right-most condition disk containing the default condition.

The choice box in HyperPascal allows the programmer to represent several common conditional constructs using a single programming component. This allows the recognition of conditional constructs at a glance and, according to the number of conditions present, enables the programmer to judge which type of conditional construct it is. The use of the single component (the choice box) to represent the common types of conditional construct also enables one construct to be changed to another easily (by adding or removing conditions).

The choice box allows the specification of conditional execution, but the actions executed are still sequential. Most non-trivial programs involve of repetition of a number of

actions. Repetition could be implemented using recursion (which can be implemented in HyperPascal by having a subprogram call itself, or a subprogram that calls it), but a less complex way of representing repetition is with the use of loop constructs.

Loops

Loop constructs are used in a program to execute a group of actions a number of times. The number of times the actions are executed (or whether they are executed at all) depends both on the type of loop, and the result of a condition associated with the loop. There are three common types of loop: the while-loop, which checks the condition, then loops through the actions if the condition evaluates to true; the repeat-until-loop, which executes the actions then, if the condition is false, loops back to execute the actions again; the for-loop, which has a counter variable, and loops through the actions with the counter variables set to the current count, until the counter variable reaches a set value.

Loop control in HyperPascal is specified using round cornered, light orange coloured boxes. The repeated code is contained in the loop icon's sub-trees. Control loops are generalised by allowing pre-loop and post-loop actions to be associated with the icon, and allowing multiple continue, and terminate loop conditions to be placed anywhere in the execution of the loop. The conditions are also allowed to have pre-test and post-test actions associated with them. The general appearance of a loop box (using reduced representations) is shown in figure 4.32.

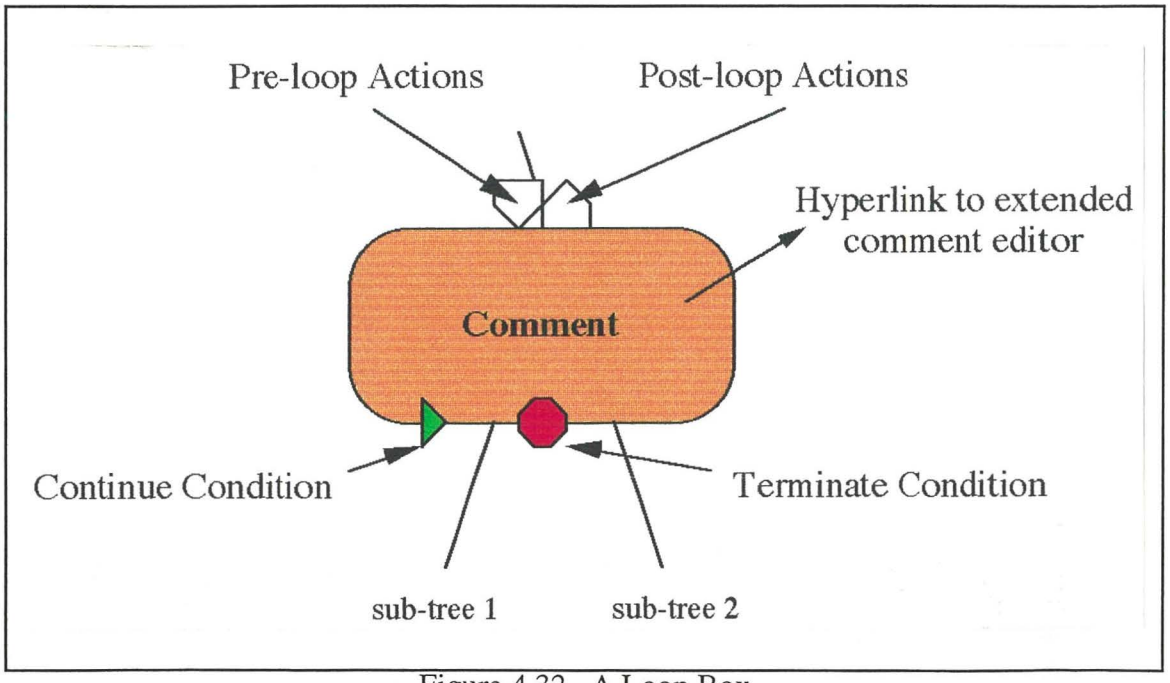


Figure 4.32. A Loop Box

In conventional languages, loops have statements before and after the loop (outside the loop), that are related to the functionality of the loop. Actions typically performed before a loop include the initialisation of variables used in the loop, and pre-reading of data used for loop control. Actions performed after a loop typically include the assigning of loop variables to other variables. These actions are conceptually associated with the loop, and should be physically associated with the loop to enable the whole conceptual construct to be manipulated as one. HyperPascal loops have pre-loop actions that can contain a sequence of actions to be executed on entry to the loop (but not on each loop repetition), and post-loop actions which can contain a sequence of actions that are executed on exit from the loop. The reduced representations of pre-loop and post-loop actions are coloured white if no actions are contained, otherwise they are grey. An expanded representation of pre-loop and post-loop actions is shown in figure 4.33.

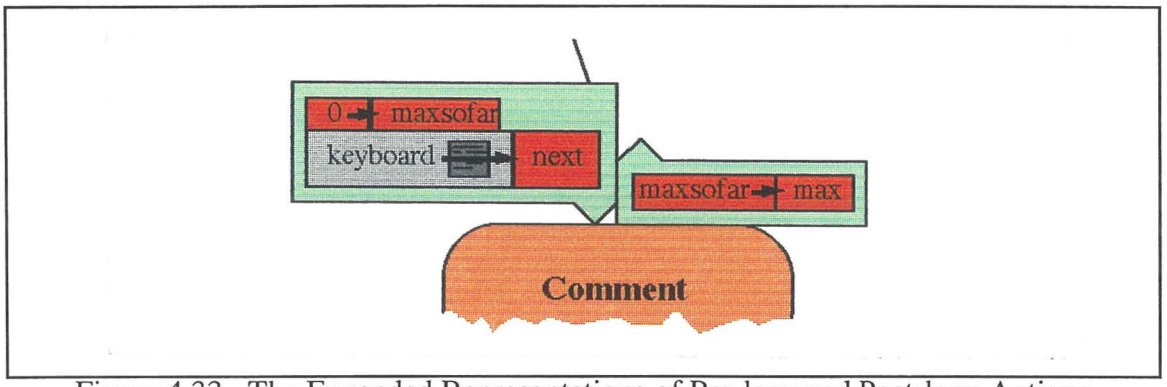


Figure 4.33. The Expanded Representations of Pre-loop and Post-loop Actions

Three types of loop are commonly available in traditional languages; while loops, repeat-until loops, and for loops. These loop constructs either have a continue-looping condition at the very beginning of the loop (while loop), or a finish-looping condition at the very end of the loop (repeat-until loop, and for loop). By allowing the user to position continue-conditions and terminate-conditions anywhere in the sequence of sub-trees attached to the loop, HyperPascal can emulate these three conventional loop constructs, and more, with a single icon. Figure 4.34 shows the representation of the three standard loops, and a non-standard loop.

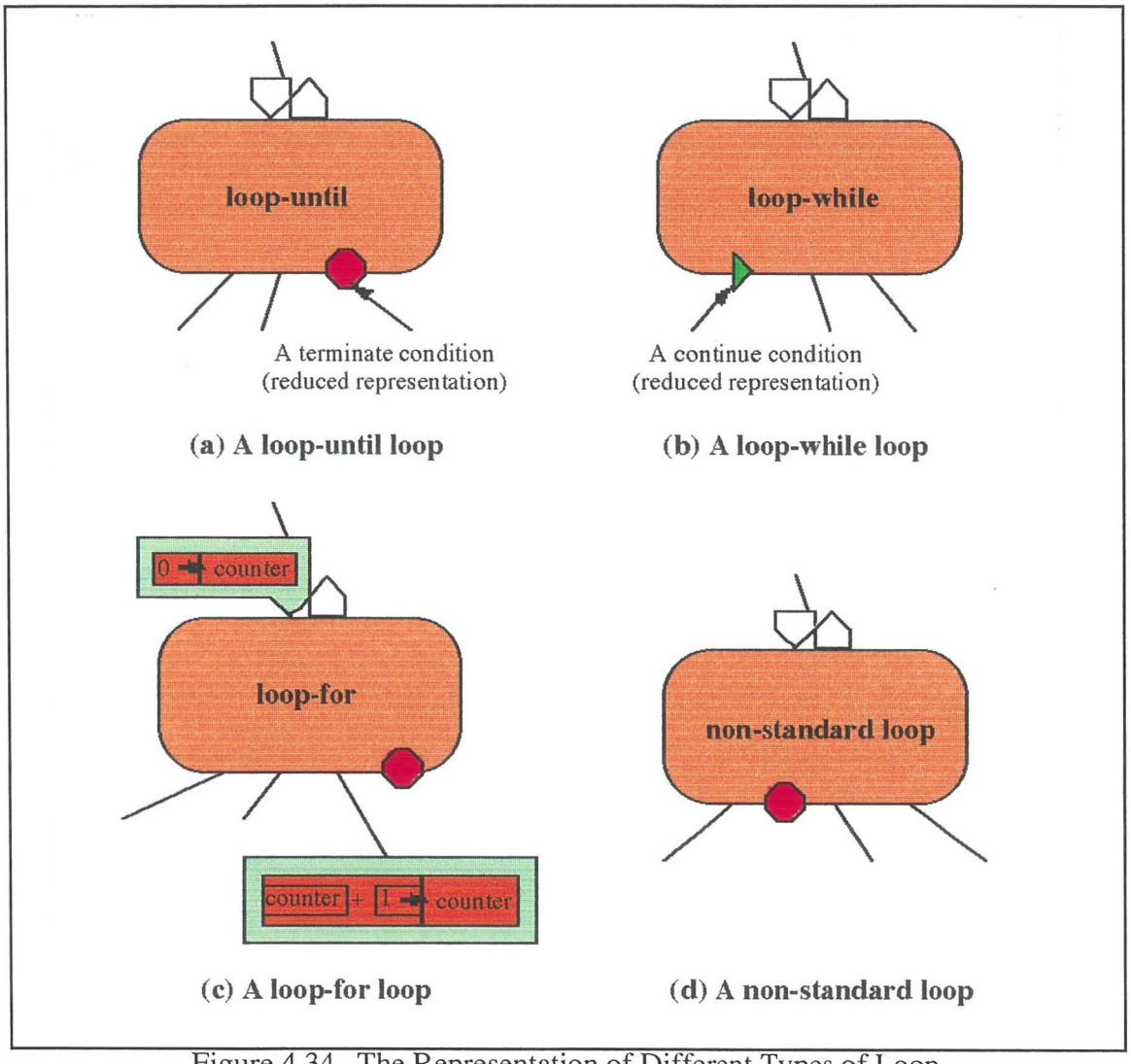


Figure 4.34. The Representation of Different Types of Loop

A repeat-until loop is represented by placing a terminate condition as the right-most child of the loop (figure 4.34(a)). If this condition evaluates to true, the loop will exit at this point (execute the post-loop actions, if present, and return control to the parent). A while loop is represented by placing a continue condition as the left-most child of the loop (figure 4.34(b)). If this condition evaluates to true, the loop continues, otherwise it exits at this point. The implementation of a for loop is a little more complex, and involves using a counter which is initialised on entry to the loop, and is increased (or decreased if the for loop is required to count downwards) before a terminate condition checks whether it is above the maximum value (figure 4.34(c)). The non-standard loop in figure 4.34(d) checks the terminate condition only after the first sub-tree is executed, and can cause loop termination at that point. This means that the first sub-tree will execute once before the rest of the loop is conditionally executed, and then as the last sub-tree executed for every repetition of the loop (this is useful for the representation of pre-read situations such as shown in figures 4.16 and 4.17).

Condition checks often rely on some actions to be performed just prior to the condition checks (such as the "increment counter" action in figure 4.34(c)), or just after the condition check. These actions are conceptually associated with the condition check, and should, therefore, be physically associated with it. HyperPascal allows the association of actions before or after a continue or terminate condition.

A terminate (or continue) condition can be expanded to show that it is represented as an action sequence with a condition inside it (the condition is shown in the sequence with the icon for the reduced representation beside it). The actions which preceed the condition are performed, then the condition is evaluated. If the loop continues (a continue condition evaluates to true, or a terminate condition evaluates to false), the actions after the condition are executed, otherwise they are not. The expanded version of this *condition sequence* is connected to the loop by an arc to the reduced representation of the condition. The for loop shown in figure 4.34(c) is modified to associate the counter increment with the condition, and is shown in figure 4.35.

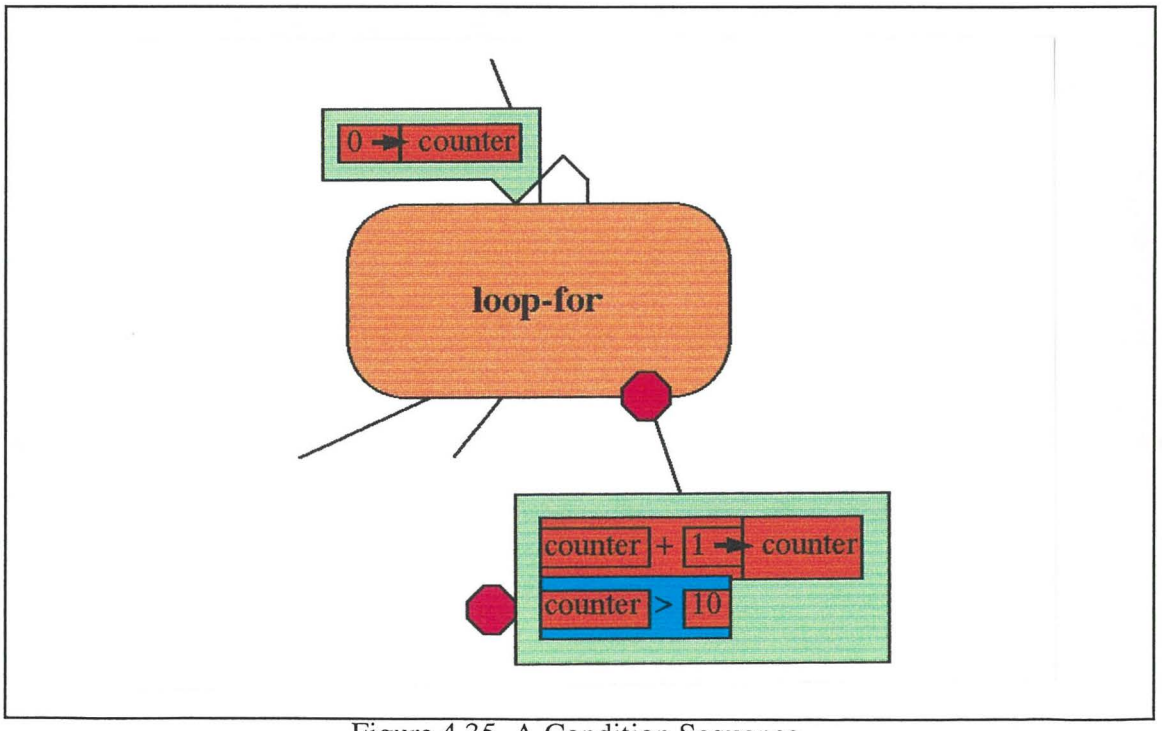


Figure 4.35 A Condition Sequence

The HyperPascal loop box allows the specification of the common types of loop, and non-standard loops, with a single generalised component. This means that loop constructs can be easily recognised, and the type of loop can be altered by modifying, or re-arranging the components of the loop component. The physical association of actions before and after the loop, allows the whole conceptual loop to be treated as a single component. This means that adding, deleting, or moving a loop, is also applied to the associated pre- and post-loop actions. The association of a condition and its related actions, allows it also to be treated as a single component.

This section has so far dealt with the notation of the action tree, but an important part of the representation of a hyperprogram is also the inter-view relationships which allow navigation inside the hyperprogram. These relationships are represented using hyperlinks between the action tree, and the other views. These hyperlinks from the action tree are now discussed.

Hyperlinks from the Action Tree

A subprogram declaration in the scope tree has a corresponding action tree, and it is necessary to travel between these views to fully define a program. It is possible to travel from the scope tree to the action tree by activating a hyperlink from the subprogram header in the scope tree. Likewise, a hyperlink can be activated from the background of the subprogram header, in the action tree, to travel to the scope tree. This hyperlink is shown in figure 4.36. The other hyperlinks from the subprogram header in the action tree are the same as those in the scope tree.

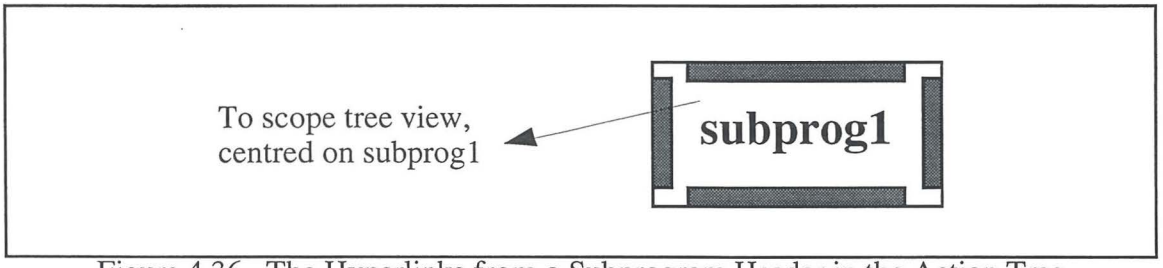


Figure 4.36. The Hyperlinks from a Subprogram Header in the Action Tree

All variables, constants, and parameters in the action tree, have a corresponding type or value (if left undefined, their type is "undefined") associated with their name. There is therefore a relationship between an identifier name, and its corresponding type. The name of an identifier can be modified directly, but the type associated with this identifier can only be changed by activating a hyperlink from the identifier to the type editor. Examples of these hyperlinks are shown in figure 4.37.

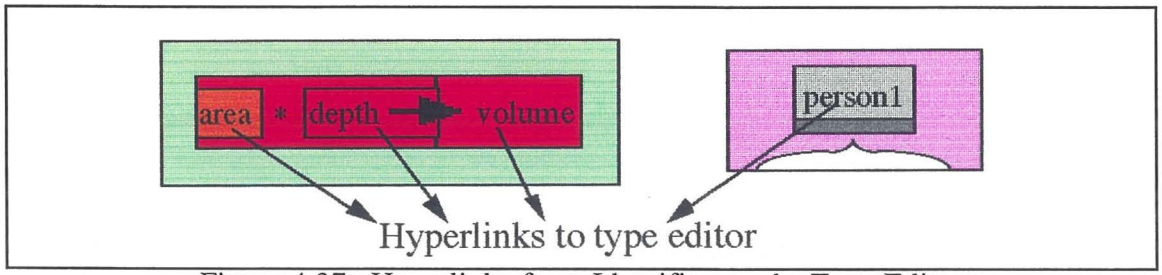


Figure 4.37. Hyperlinks from Identifiers to the Type Editor

Each of the major action tree components can have an extended comment associated with it. The user can edit this extended comment by activating a hyperlink from the background of the component to the extended comment editor. Figure 4.38. shows the hyperlinks from components to the extended comment editor.

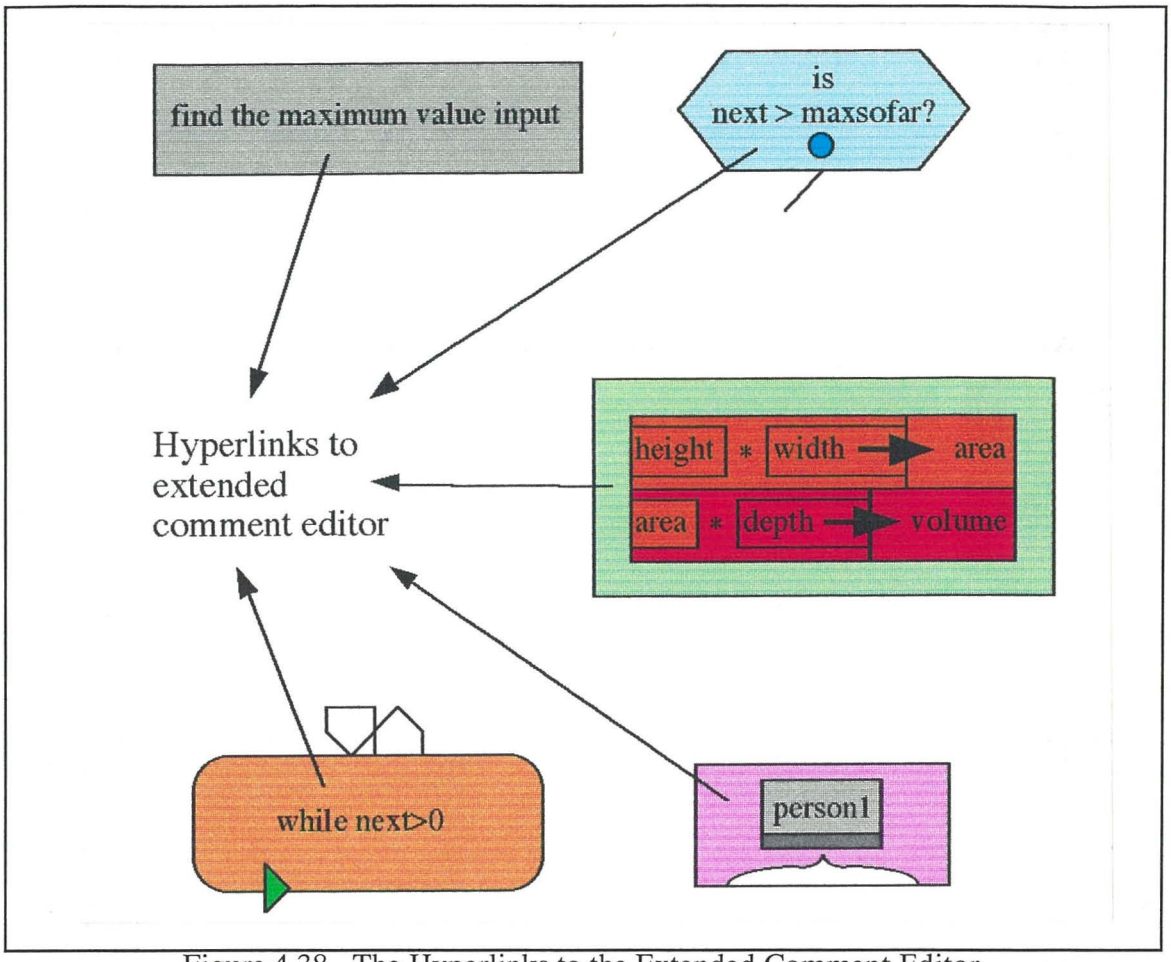


Figure 4.38. The Hyperlinks to the Extended Comment Editor

Actions in HyperPascal can include the invoking of subprograms. These subprogram invocations are used to perform actions that are present in another action tree, related to the subprogram invoked. The invoking of the subprogram is therefore related to the action tree of the subprogram, and this action tree can be travelled to by activating a hyperlink present in the background of the subprogram in the subprogram invocation. The hyperlinks from a subprogram invocation are shown in figure 4.39.

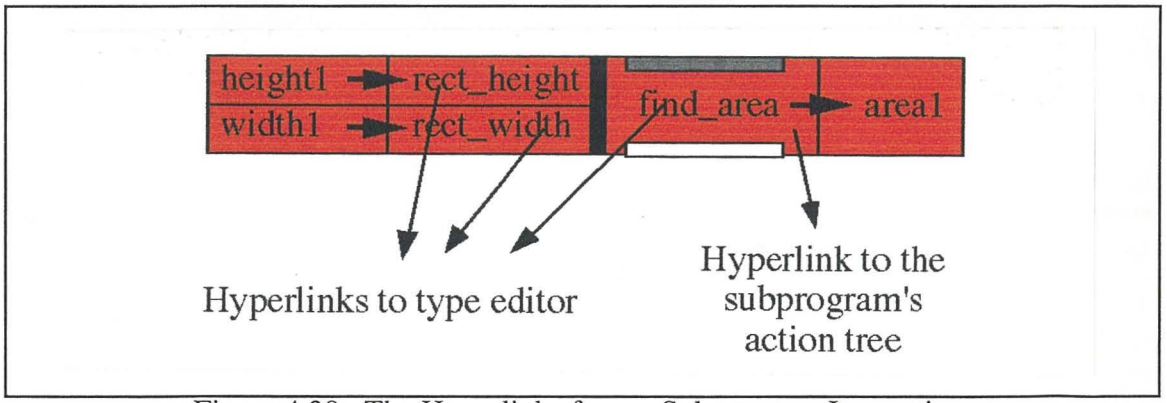


Figure 4.39. The Hyperlinks from a Subprogram Invocation

As well as the hyperlink from the subprogram background to the corresponding action tree, there are other hyperlinks to the type editor. The activation of the hyperlink from the subprogram name takes the user to the type editor to edit the return type of the subprogram, whereas activating a hyperlink from an input or output parameter takes the

user to the type editor to edit the type of that parameter. The hyperlinks from the variables have been shown previously, so are not shown here.

Formatting information is associated with input and output actions in HyperPascal. This formatting information is not shown in the action tree because, if extensive, it could obscure the underlying functionality of the program. It is associated with the input/output statements using a hyperlink to a forms window. Figure 4.40 shows the hyperlinks from input and output actions (again, the hyperlinks from the variables and the constant are not shown).

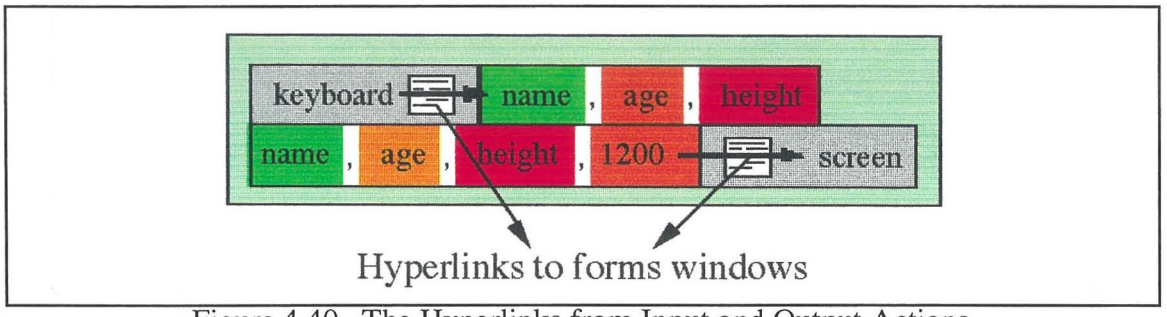


Figure 4.40. The Hyperlinks from Input and Output Actions

A hyperlink from an input or output action can take the user to the forms window that holds the corresponding formatting information. The forms windows are now discussed in detail.

4.4. Forms Windows

The description of the appearance of the input and output of a program is an important part of the programs description because it is through input and output that the user communicates with the program. In a text based input/output scheme, this description consists of prompts for input, formatting of output (the text associated with or explaining the output), and layout information. This version of HyperPascal assumes a text-based input/output scheme, and from here on, the prompt, formatting, and layout information will be referred to as the screen formatting information.

Traditional (textual) languages often include the screen formatting information in the same statement as the reading or writing of the data, or in similar statements. The outputting of this screen formatting information can hide the actual input and output of data taking place, and should therefore either have a different representation to the input and output of data, or be shown separately.

The sequential nature of traditional languages means that screen formatting information is usually dispersed throughout the program so that it is output to the screen at the appropriate point of execution, and in the appropriate place on the screen. This dispersal of formatting information throughout the program can sometimes (by sheer quantity) obscure the actual data manipulation actions of the program. A different representation of the screen formatting information is not enough in this case; the screen formatting information should be shown separately.

Because screen formatting information is traditionally represented as a sequence of outputs to the screen, inserting the input or output of data in the middle of this screen formatting information, can cause it to become fragmented and hard to visualise as a whole. Figure 4.41 demonstrates this by showing an excerpt from part of the input and output appearance of a program, and the Pascal code needed to describe it.

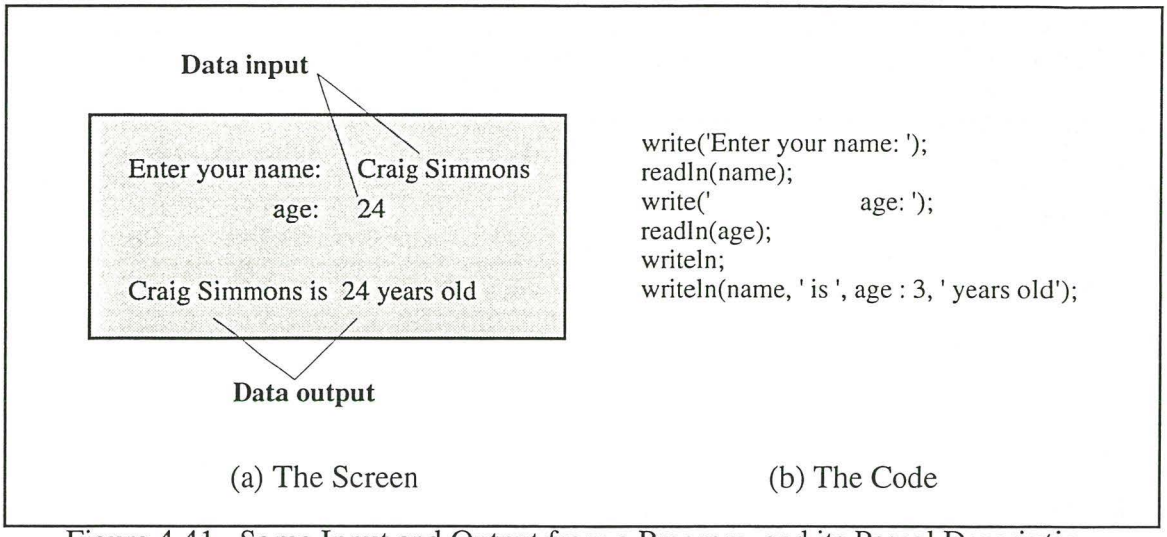


Figure 4.41. Some Input and Output from a Program, and its Pascal Description

The screen shown in figure 4.41(a) is described by the code in figure 4.41(b). However, it is not easy to gain a true appreciation of the screen appearance from the code. This is a simple example, and it is much harder to visualise the screen appearance when looking at a larger segment of code describing a more complex screen appearance. A better idea of the run-time appearance of the screen could be gained by describing the screen formatting information so its appearance mirrors the appearance of the screen at run-time. To enable the actual input and output data to be taken into account, this should be shown with, but distinguished from, the screen formatting data.

HyperPascal describes the screen formatting associated with an input or output action (or multiple inputs or outputs) using the forms window. A forms window is associated, via a hyperlink, to the input or output action, but shows the formatting of the whole action sequence in which the input or output action is contained. The appearance of a simple forms window is shown in figure 4.42.

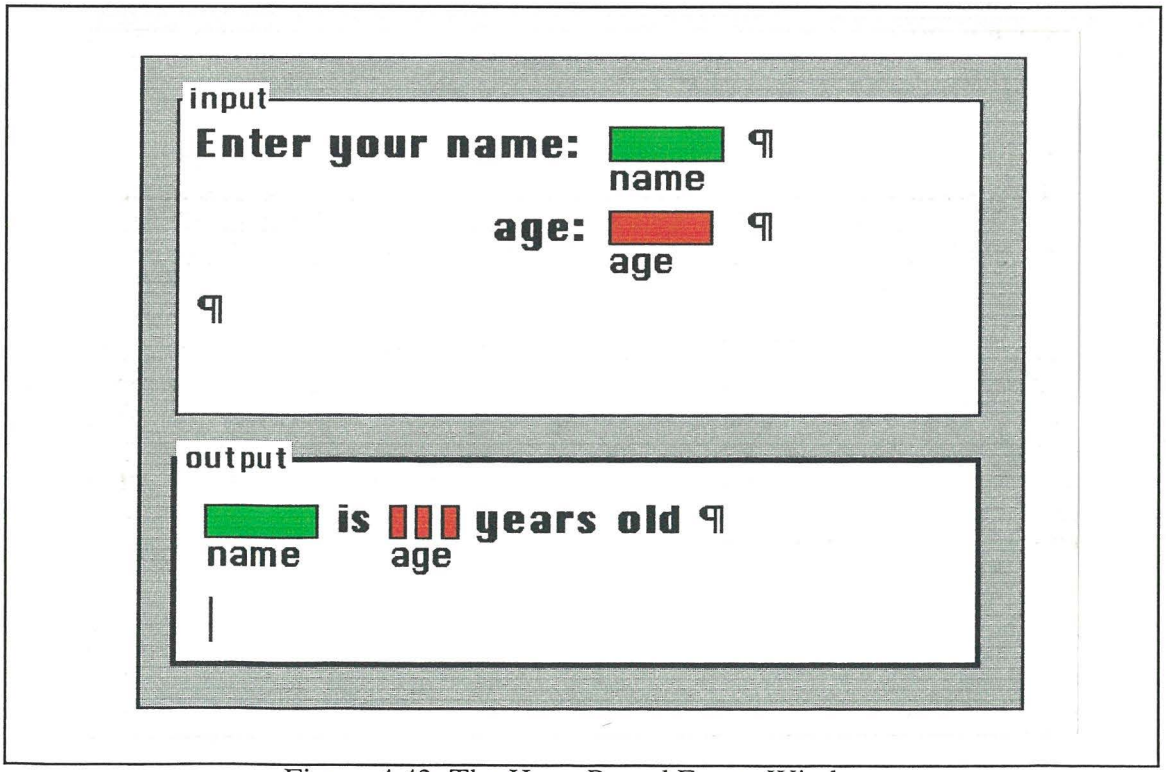


Figure 4.42. The HyperPascal Forms Window

The forms window in figure 4.42, shows the formatting associated with an input action (assigning from the keyboard to two variables), and an output action (assigning two variables to the screen). An input or output variable is shown as a colour-coded (according to the variable's type) field with the name of the variable underneath the field. This use of the variables' names is an edit-time mneumonic, and does not appear in the window when the program is running. The forms window can be used to add additional fields to an input or output, or to add or remove an input or output action from the action sequence.

There are three types of variable field possible in the forms windows; The variable length field (▓▓▓▓▓) represents a variable that may have any length on the screen; The fixed length field (□□□) represents a variable that uses a set number of screen positions (corresponding to the number of small squares) for its display; The fixed length decimal field (□□□□.□□) is similar to the fixed length field except it also allows the specification of the number of screen positions after the decimal point. These different fields are used for different purposes.

Since it is difficult in a textual input/output environment to restrict the user to a certain length when inputting values, all input variables are represented by variable length fields. The output of strings is represented with a variable length field because it is difficult for the programmer to predict an appropriate length of an arbitrary string. Integer variables are output to a field with a default length of five characters. This length may be edited by the programmer. Boolean values are output to a one-character-wide field. Real variables are output to fields with a default size of five characters before the decimal point, and two after. The size of these subfields may be edited by the programmer.

Normal assignment actions in HyperPascal do not affect the screen format, so are not shown in the forms window. Actions that invoke subprograms can, however, affect the screen format, as the subprogram called could write information to the screen. The information this subprogram writes to the screen may not be able to be ascertained at write-time (it may involve conditional or repetitive execution), so it is possible that no meaningful screen formatting information can be shown for it. In the forms window, HyperPascal represents subprogram invocations as grey boxes with the name of the subprogram inside them. An example of this is shown in figure 4.43.

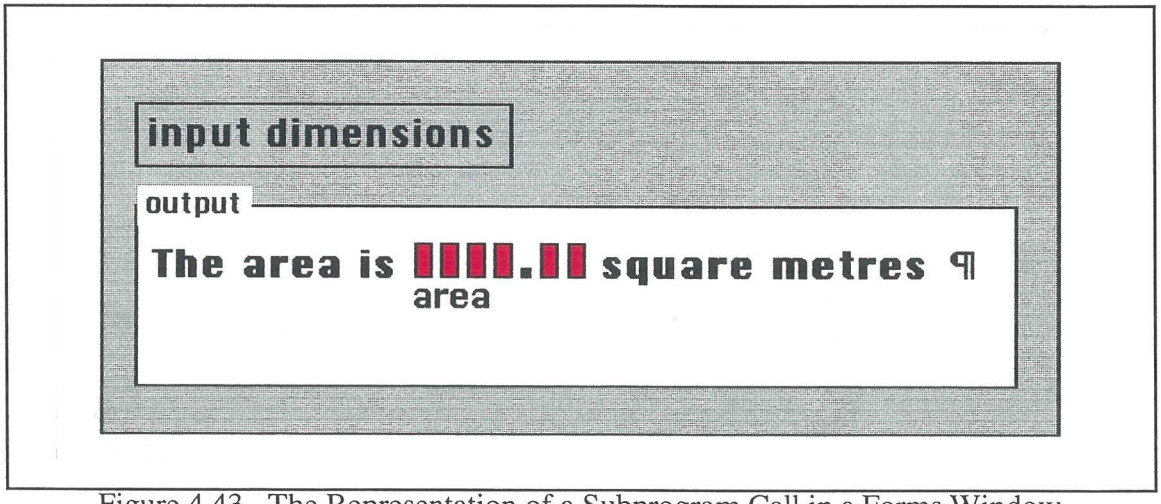


Figure 4.43. The Representation of a Subprogram Call in a Forms Window

The use of the forms windows allows the formatting information associated with input and output actions to be designed as it will appear on the screen, and the removing of the

formatting information from the action tree also stops it from obscuring the underlying functionality of the actions.

The formatting information in the forms windows is found by travelling along the hyperlinks from the input and output actions. It is also desirable to be able to navigate from the formatting information in the forms window to the associated action, or to be able to change the definition of a field. Hyperlinks are provided from the forms windows for these purposes.

Hyperlinks from the Forms Windows

A forms window, may show the formatting information for a number of input and output actions. The formatting information in the forms window is associated with the whole action sequence, but it is divided into smaller sections showing the formatting information for a single input or output action. The formatting information in one of these smaller sections is directly associated with an input or output action, and this association is represented in HyperPascal with a hyperlink. This hyperlink is connected from the border of a formatting section to the action tree with the view centred on the corresponding input or output action (shown in figure 4.44).

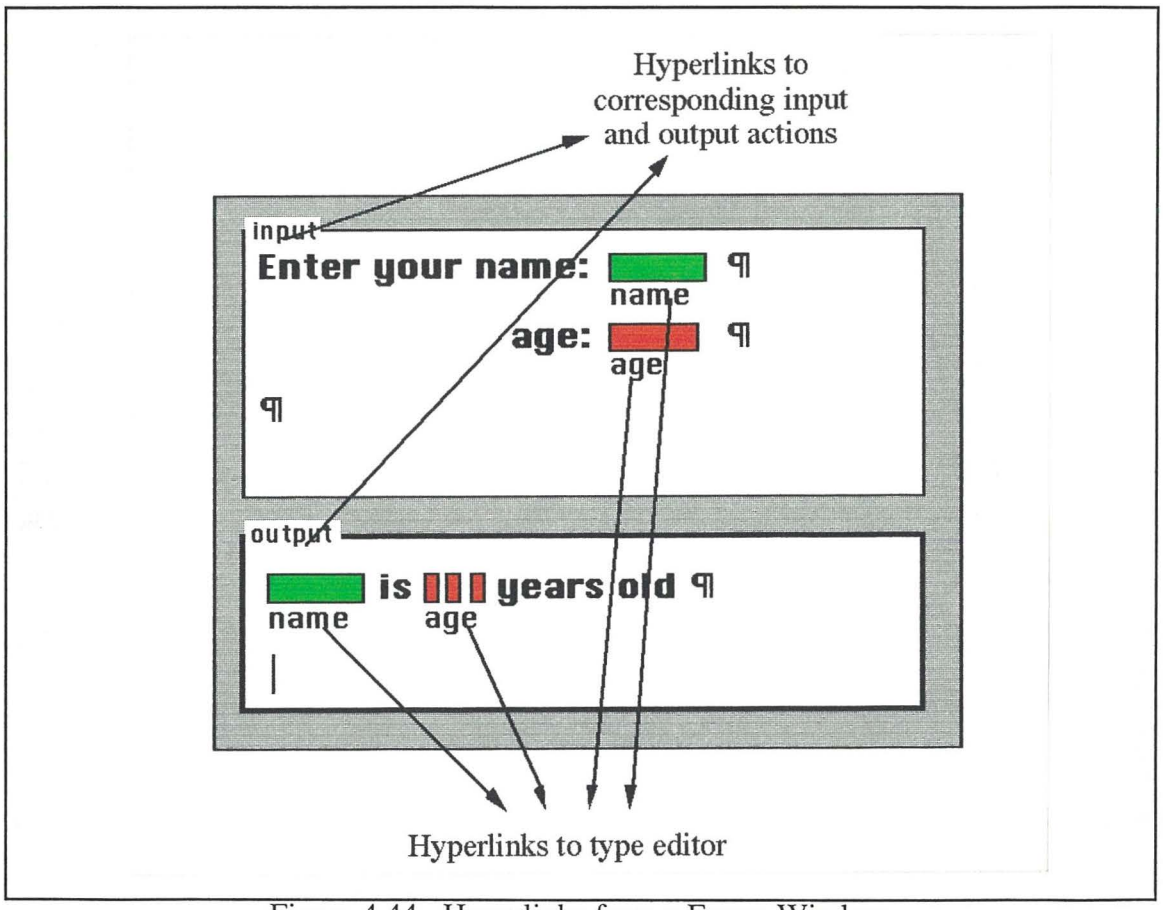


Figure 4.44. Hyperlinks from a Forms Window

Figure 4.44 also shows hyperlinks from the names associated with input and output fields to the type editor. The type associated with this identifier can be changed in this view by activating a hyperlink from the identifier to the type editor.

So far the HyperPascal notation and navigation have been discussed, but little mention has been made about the methods of interacting with the program when creating, moving, or editing components. Users of differing levels of expertise typically prefer different interaction techniques. If a visual programming language is to be appealing to both

novices and experts, these differing techniques should be supported when editing. The interaction techniques supported by HyperPascal when editing a program are discussed now.

4.5. Editing HyperPascal Programs

The creation and editing of HyperPascal programs is performed in the different views of the hyperprogram, possibly using a number of the different interaction styles available. Generally, the creation and editing of a HyperPascal program is allowed using menu's and dialogs, and direct manipulation and editing-in-place techniques. The direct manipulation and editing-in-place techniques described in this section are illustrated using a graphical notation to represent a mouse-down, a mouse-up, a drag with the mouse button down, a single-click, and a double-click. This graphical notation is outlined in figure 4.45.

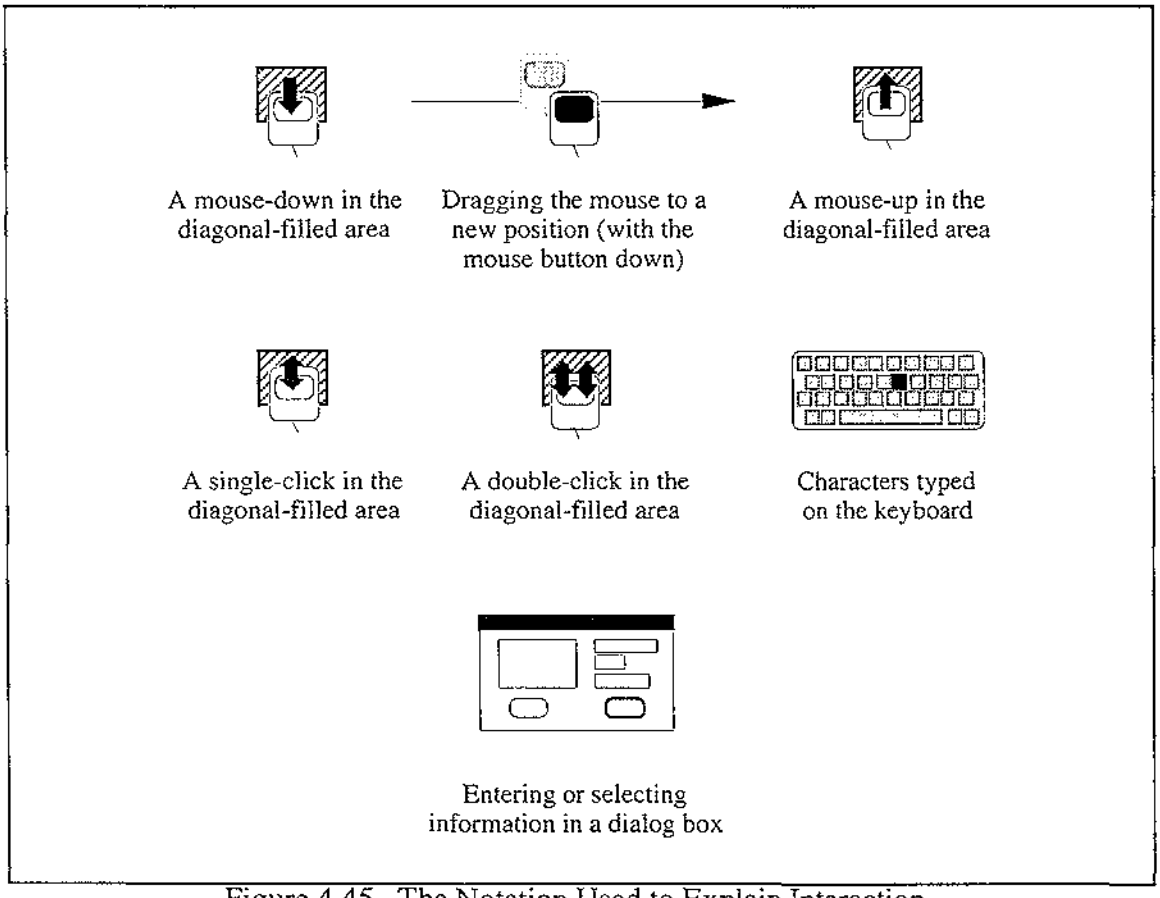


Figure 4.45. The Notation Used to Explain Interaction.

The techniques used to create and modify a HyperPascal program are now discussed in detail.

Creating and Arranging Program Components

HyperPascal programs are generated by creating, modifying, and arranging program components. In order to accommodate many different interaction styles, different methods of manipulating the program are allowed. Program components can be created using menus, hot-keys, and direct manipulation methods.

Adding a program component using a menu or hot-key (when no component is currently selected), results in the corresponding component being added in the centre of the view, and it will not be connected to any other components. If a program component is added while another component is currently selected, the component is added as the right-most child of the selected component. In this case, the parent-child connector is automatically added between the two components.

A direct manipulation approach to creating components allows components to be added as children of another component. Figure 4.46 illustrates the method for adding a child component to an existing program component. When adding a component in this manner in the scope tree, the only component possible to add is the subprogram, so it is added at the mouse-up position. In the action tree however, there are alternatives in the kind of component that can be added, so a choice of component kind must be made before the component is added.

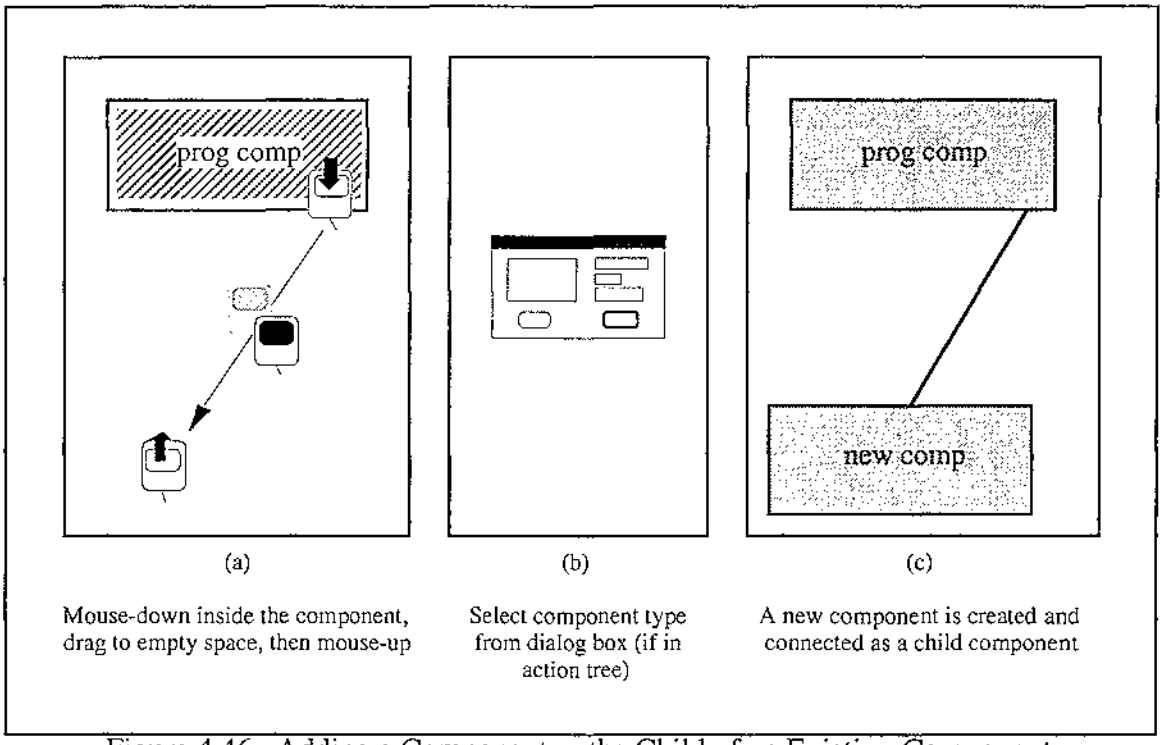


Figure 4.46. Adding a Component as the Child of an Existing Component

Program components may be connected in parent-child relationships either by menu, hot-key, or direct manipulation. To connect two components in a parent-child relationship using the menu or hot-key, both components must be currently selected, and when they are connected, the component located at the highest position will be connected as the parent in the relationship. The direct manipulation method for connecting components in a parent-child relationship is similar to the method for adding a child component (illustrated in figure 4.46), except that the mouse-up occurs over the intended child component, and no dialog appears.

HyperPascal program components are selected by performing a single mouse click on their outer boundary. More than one component may be selected by holding the "shift" key down and selecting a series of components. Multiple components can also be selected using marquee selection, where the user mouse-downs in empty space, which positions one corner of a selection rectangle there, then drags the mouse (and the opposite corner of the rectangle) to a new position and mouse-ups. All components that are completely within the area of the rectangle are selected.

Components, or groups of components may be re-positioned by performing a mouse-down on a component boundary and dragging the component (or group of components) to a new position. A group of components, when dragged to a new position, retain their positions relative to each other.

A selected component(s) may be deleted either by pressing "delete", or by selecting delete or cut from a menu. If the cut option is used, the component(s) may be replaced at the last mouse-click position by selecting paste from a menu. This is another way of re-arranging the components.

The order of execution of a component's children can be modified by re-arranging the order of the connections attached to the bottom of the component. This re-arrangement can be performed either by selecting and deleting a connector, then re-connecting the child component in a different position, or by selecting the connector, then dragging the end of the connector attached to the parent to its new position (as shown in figure 4.47). The child component (or sub-tree) can then be moved to tidy up the diagram.

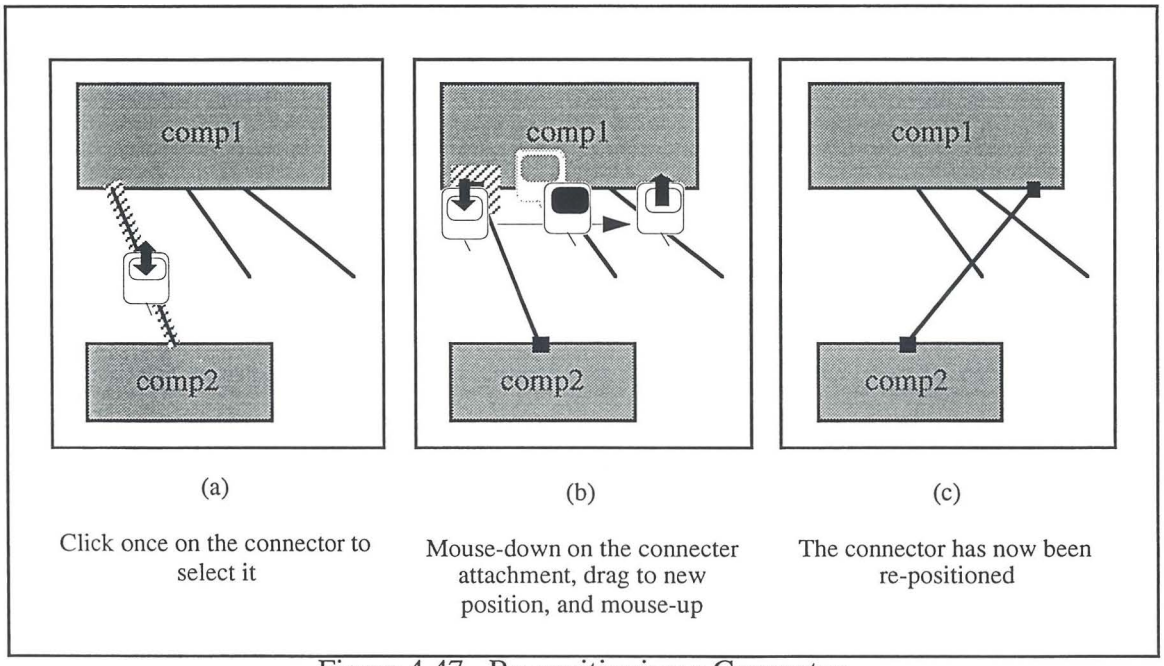


Figure 4.47. Re-positioning a Connector

The HyperPascal approach to adding, arranging, and deleting program components allows these actions to be performed using differing interaction techniques in order to accommodate the preferences of different users. To specify a program however, the program components themselves must be able to be edited. The approach used by HyperPascal in editing program component is discussed below.

Editing Subprogram Declarations

A subprogram declaration includes the declarations of its input and output parameters, variables, constants, and types, as well as the name of the program, and its return type.

Subprogram declarations allow the user-controlled hiding and showing of much of the declaration information associated with them. This is done using collapse bars as a reduced representation of the declaration areas. A collapse bar can be expanded to a declaration area, and a declaration area reduced again to a collapse bar using the method illustrated in figure 4.48.

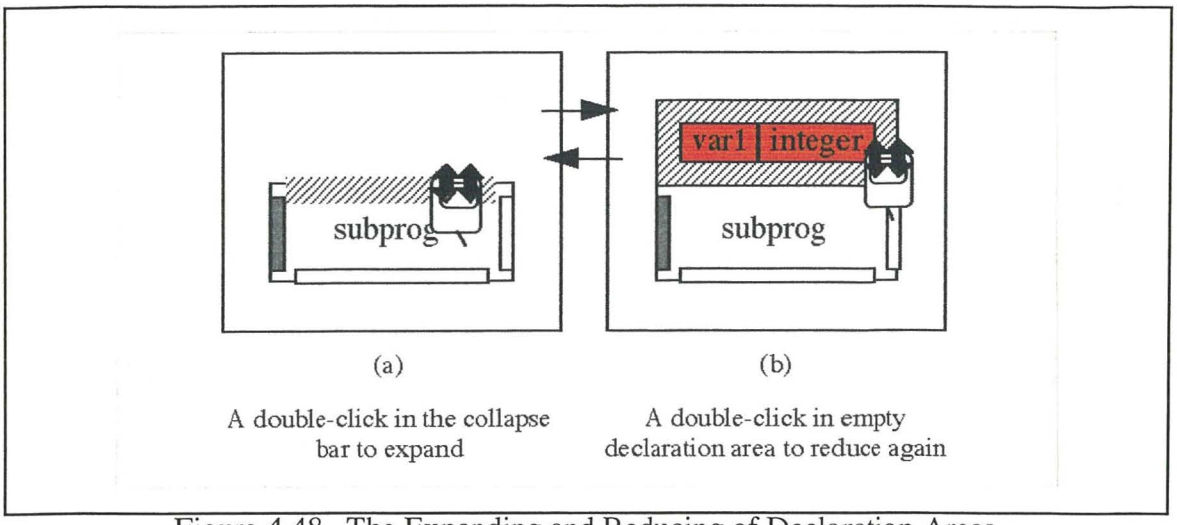


Figure 4.48. The Expanding and Reducing of Declaration Areas

Declarations in a HyperPascal program are composed of two parts, a name, and a description. The name and description of a simple declaration can be edited in place by positioning the cursor, then using the keyboard to modify (or add) the name and simple description consisting of a named type, or a constant value. The process of creating a simple declaration is shown in figure 4.49. The declaration can be left undefined by entering a white space character or return at the stage shown in figure 4.49(c) instead of entering the description.

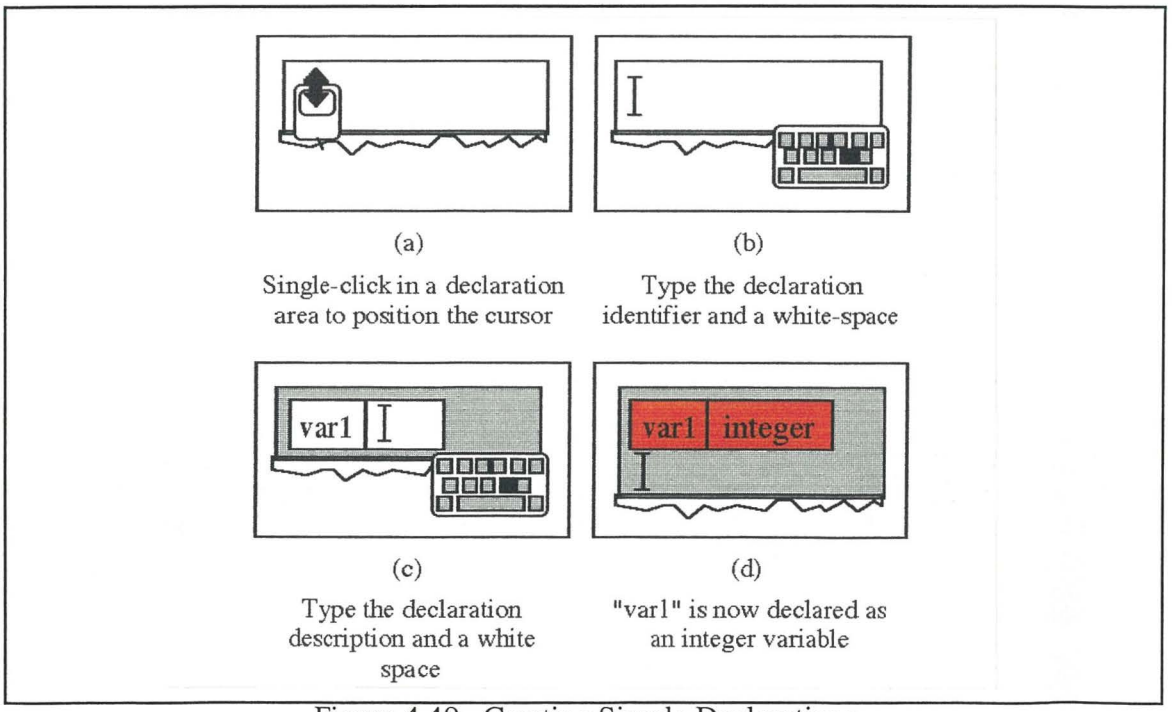


Figure 4.49. Creating Simple Declarations

The description of a complex declaration can be complicated, and sometimes highly structured in nature, which makes the editing in place technique unsuitable for editing them. Simple, and complex declarations can both be defined using the type editor, which is connected by a hyperlink to each declaration. Activating the hyperlink from a declaration, and the appearance of the top level of the type editor is shown in figure 4.50.

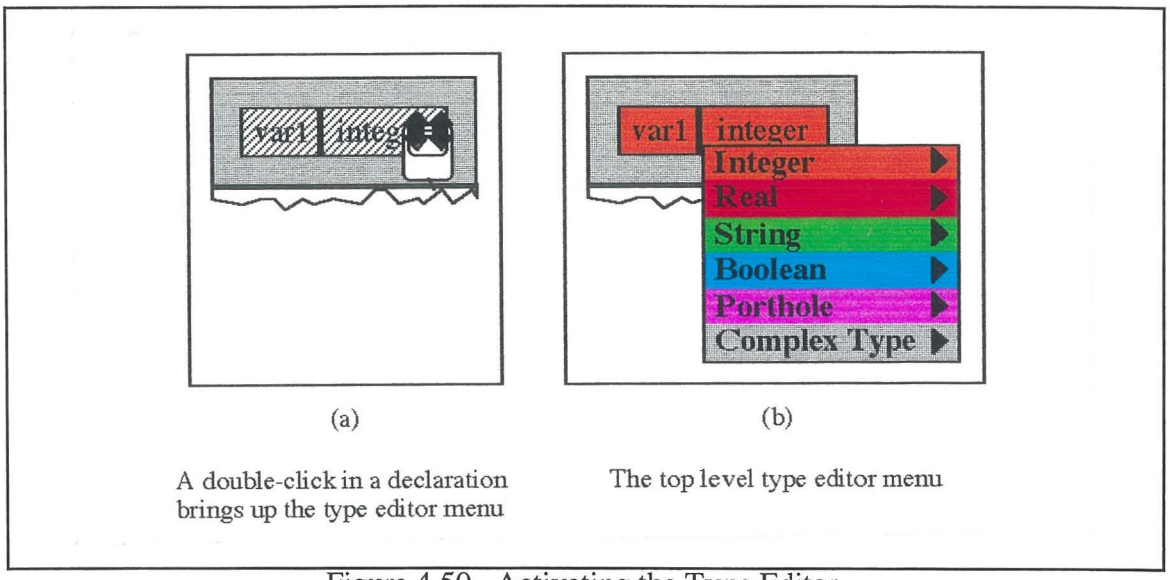


Figure 4.50. Activating the Type Editor

The top level of the type editor appears as a multi-coloured menu which allows the user to select the type of declaration required. Then, if necessary, a transition occurs, to a lower level in the editor for defining the declaration description. If no selection is desired, a single mouse-click outside the area of the menu, or any key press will deactivate the type editor. The type editor menu and sub-menus are illustrated in figure 4.51.

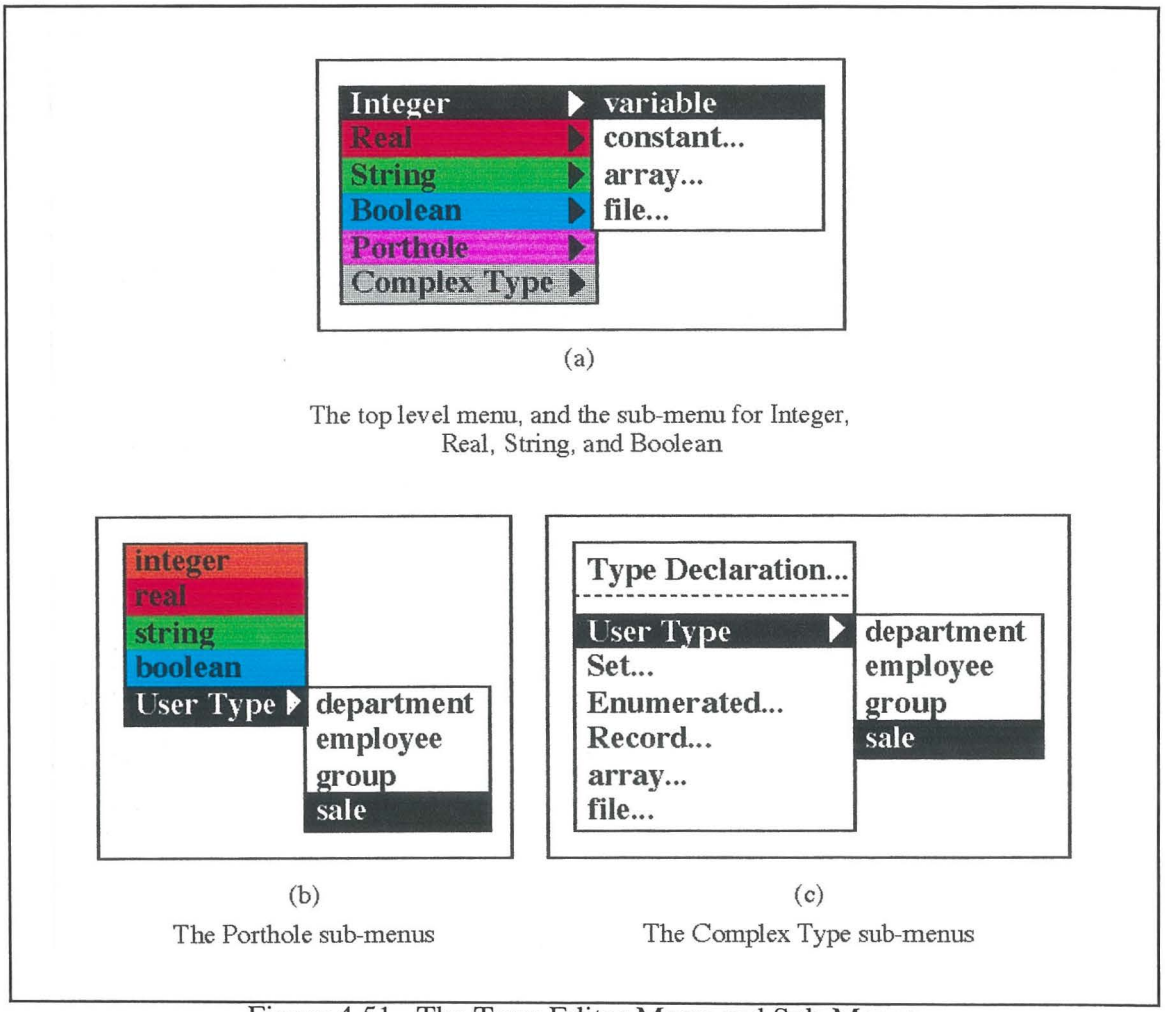


Figure 4.51. The Type Editor Menu and Sub-Menus

Figure 4.51(a) shows the top level menu, and the sub-menu for the "Integer" selection. This sub-menu allows the user to declare the declaration to be a declaration of an integer variable, an integer constant (after entering the constant value), an array of integers (after specifying the number of dimensions, and the upper and lower bounds of the dimensions), and a file of integers. The sub-menus for the top level "Real", "String", and "Boolean" selections allow similar declarations using these base types.

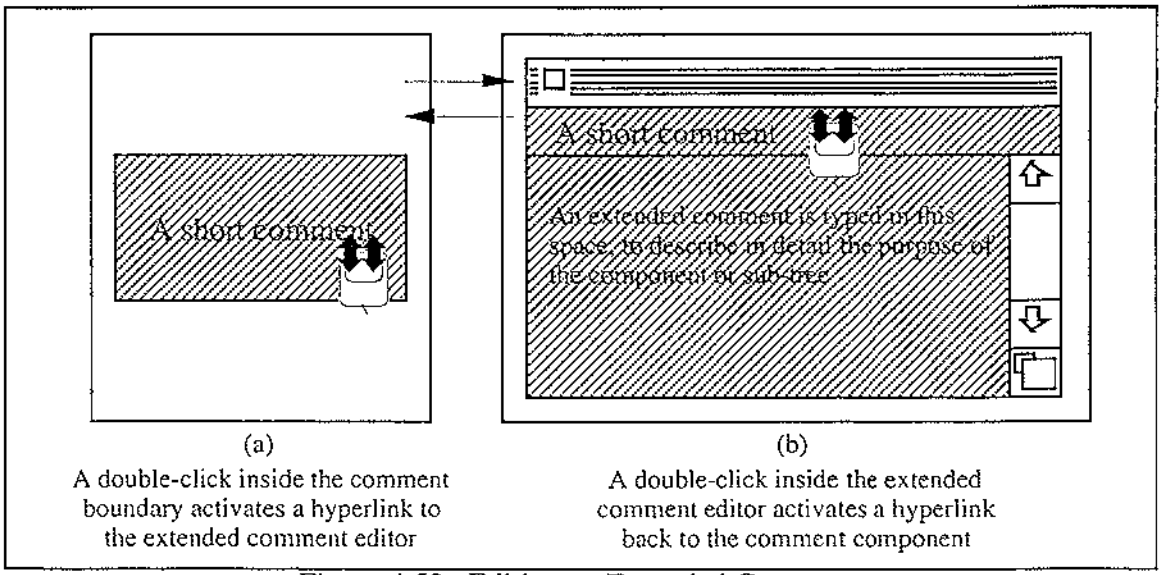
The sub-menu for the top level menu "Porthole" selection (shown in figure 4.51(b)) allows the user to specify a porthole onto one of the base types, or a user defined type. The sub-menu of the "User Type" selection in the sub-menu allows the selection of a type from a list of possible user-defined types (this list only shows the types available at the current scope).

The sub-menu for the top level menu "Complex Type" selection allows both the declaration of a user-defined type, and the declaration of a variable of a user-defined, or complex type. A user-defined type is declared by selecting the "Type Declaration..." option, and then describing the type structure. The "User Type" option allows the user to declare a variable of a pre-defined user type, and the rest of the selections are used to declare a variable of an unnamed complex type.

The type editor can also be accessed via a hyperlink from the name of the subprogram for the purpose of modifying the return type of the subprogram, but the menu options that allow the declaration of new types are disabled. The return type can therefore be specified to be a simple type, or an existing complex type, but can not, however, be declared as returning a type declaration. The name of the subprogram can be modified in the declaration header by editing in place.

Editing Comments

The comment component in HyperPascal is the simplest program component, as it embodies no functionality, and exerts no control over the execution of the program. The comment component has a short string in the comment box, which can be edited in place, and an extended comment associated with it by a hyperlink. The appearance of the extended comment editor, and the navigation between it and the comment component is shown in figure 4.52. Once in the extended comment editor, the user can edit the short comment and extended comment in place.



(a)
A double-click inside the comment boundary activates a hyperlink to the extended comment editor

(b)
A double-click inside the extended comment editor activates a hyperlink back to the comment component

Figure 4.52. Editing an Extended Comment

Editing Action Sequences

An action sequence contains a number of actions, and is associated with an extended comment via a hyperlink. The extended comment is edited using the extended comment editor. The actions, which are generally represented as assignments, can be edited in place, or the type editor can be used to change variables. These assignment statements are composed of a donor field, which donates a value, and an acceptor field which accepts the value. Simple examples of these assignment actions can be edited in place in a similar manner to the declarations in the subprogram header.

Actions can either be edited in place, or the type editor can be used to edit the action by replacing variables in the action by selecting variables from a list of those available. The modification of a simple action using the type editor is shown in figure 4.53.

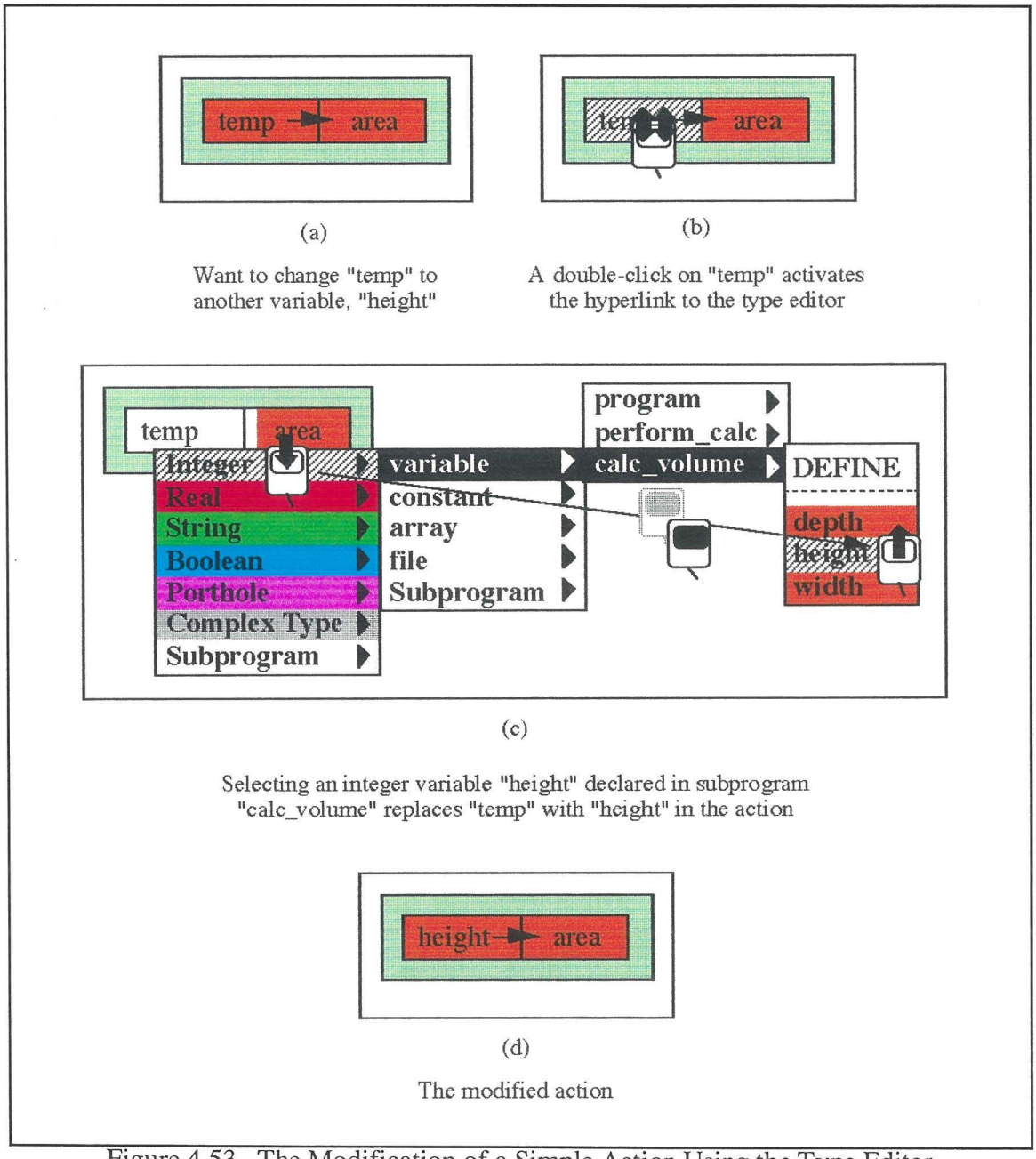


Figure 4.53. The Modification of a Simple Action Using the Type Editor

It would be a useful facility when programming to be able to declare variables and constants when they are used in a program, as this would save the programmer having to move to the appropriate program header in order to declare a variable. Similarly, it would also be useful to enable the programmer re-declare existing variables to be of different types or scope, from an instance of that variable in the action tree. HyperPascal allows this declaring "on the fly" using the type editor, which also allows the user to specify the scope of the declaration. When an undeclared variable is added to an action, the variable's field appears multi-coloured. Variables can be defined or re-defined by choosing "DECLARE" from the variable menu, which declares the variables to be of the type and scope specified in the menu. If the variable has a complex type, additional information may need to be entered in a dialog. When declaring a variable with a complex type to be of the same type and scope, the description information is edited in the dialog instead of having to be fully re-specified.

Editing Expressions

HyperPascal expressions have a nested box format, which specifies the scope of operators, so operators in HyperPascal operators have no precedence, and parentheses are also not needed. Expressions can be edited in place, and the nested boxes are automatically created in accordance with the right-associativity of the operators (see figure 4.54(a)). This right associativity can be overridden by using a left or right arrow key with the shift key pressed, to move the cursor outside the scope of the current box (as shown in figure 4.54(b)). In addition to the use of the keyboard, the mouse may be used to position the cursor at an appropriate location in the expression.

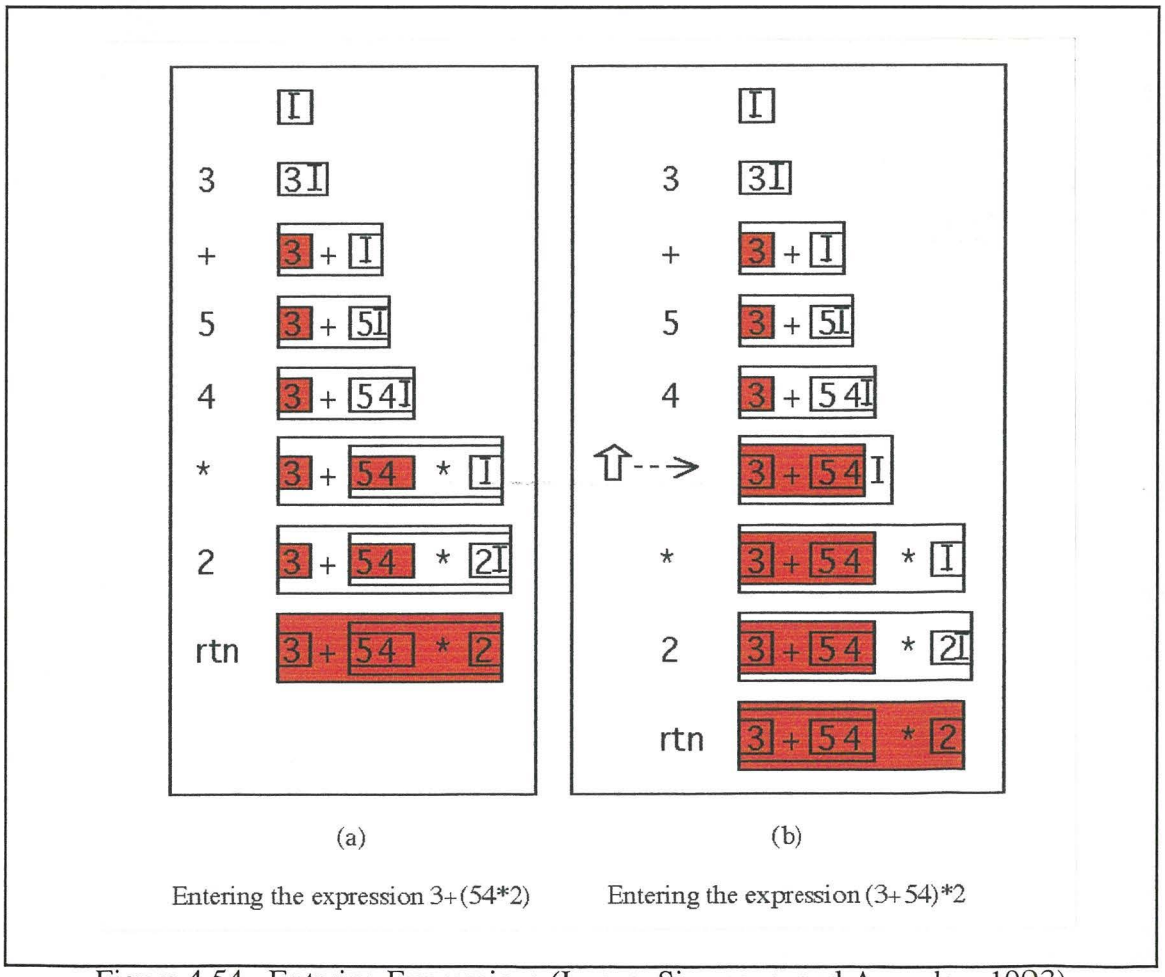


Figure 4.54. Entering Expressions (Lyons, Simmons, and Apperley, 1993)

HyperPascal subprograms are invoked from action sequences. A subprogram invocation can be added using editing-in-place techniques, or by selecting a subprogram from the type editor. When using editing-in-place techniques, the subprogram can be entered first, then the inputs and outputs inserted into the appropriate fields (shown in figure 4.55(a)), or a list of inputs can be entered as donors, then the subprogram name entered as the acceptor, and the inputs assigned to the input parameters of the subprogram (shown in figure 4.55(b)).

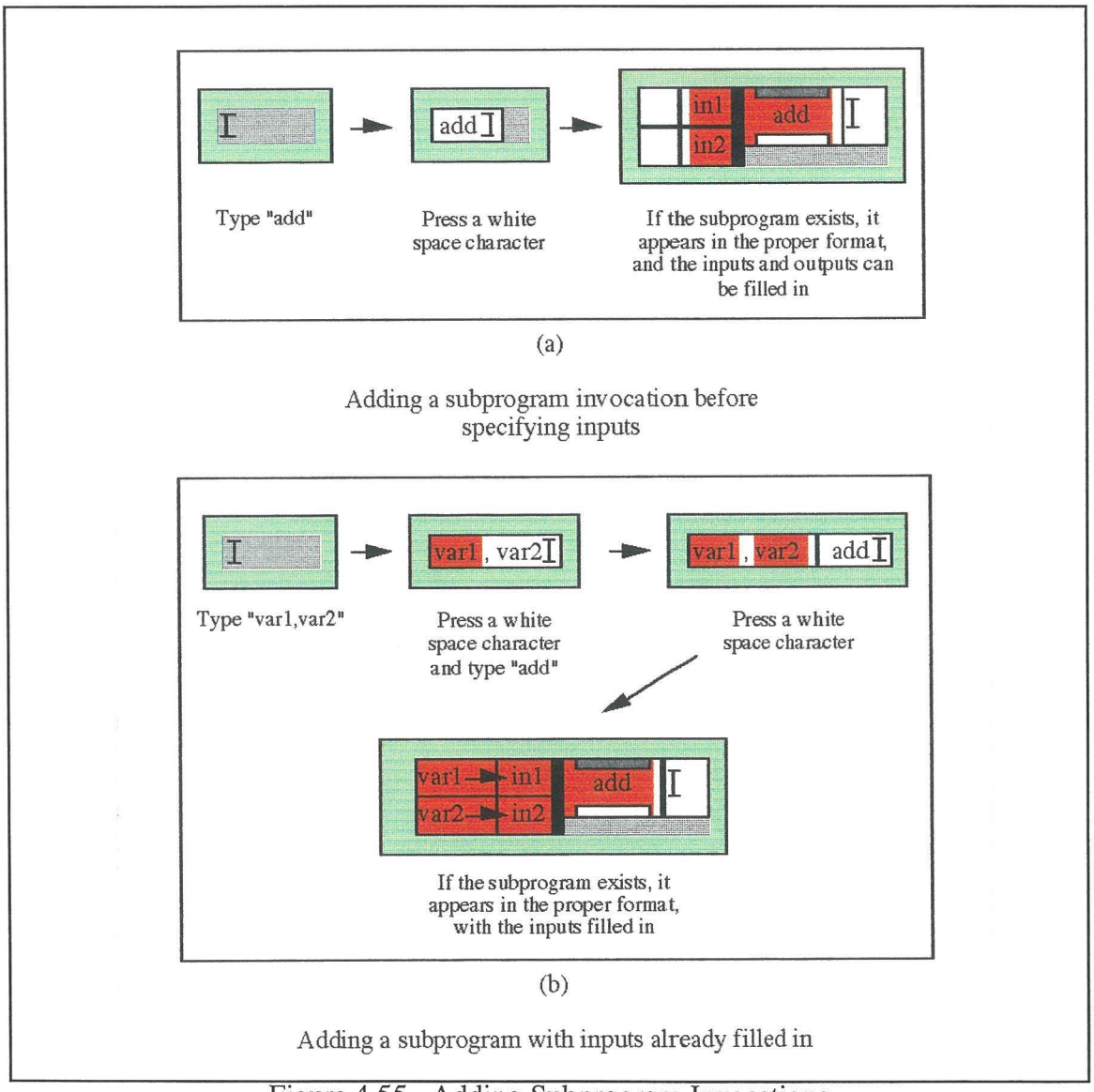


Figure 4.55. Adding Subprogram Invocations

Subprogram invocations can also be added using the type editor. The type editor top level menu, and the sub-menus for "Integer", "Real", "String", and "Boolean" have the extra "Subprogram" option added when used outside of a declaration area. This is to enable the choosing of subprograms, or their declaration "on the fly".

Input and output actions in are used in HyperPascal to get information from the keyboard or a file, and to write information to the screen, or a file. Input actions have the name of the input source in the donor field, and a list of acceptor variables or parameters in the acceptor field. Output actions have a list of value donors in the donor field, and the name of the output sink in the acceptor field. The method of editing an input or output action is

similar to that of editing a simple assignment action, except that in place of the donor or acceptor, the keywords "keyboard" or "screen" are used.

The input and output actions show only the actual data being input and output. This input and output data has formatting information associated with it. This information is viewed and edited by moving (via the formatting hyperlink) to the appropriate forms window. The hyperlink is activated by double-clicking on the formatting icon, as shown in figure 4.56.

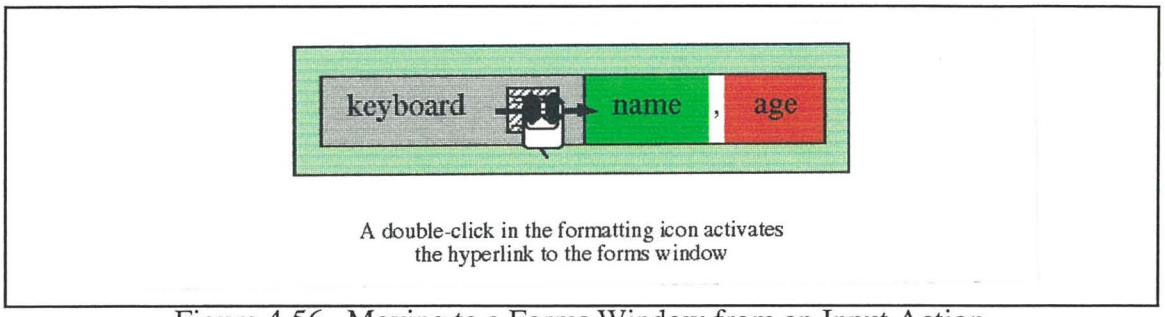


Figure 4.56. Moving to a Forms Window from an Input Action

Editing Forms Windows

In the current version of HyperPascal, input and output are textual , so the formatting of the input and output is also textual. This textual formatting information is represented in the forms window as it will appear on the screen (or in the output file), and can be edited using a simple WYSIWYG text editing approach.

Fields can be added to, modified, and deleted from an input or output action while the current view is the forms window. To delete an input/output field, it is selected by clicking once on the name or format of the field, and then deleted using a menu option or hot-key. To add a new field to an input or output action, the type editor may be used to either choose a variable, or to declare one "on the fly". A field that is removed from or added to the forms window is also removed from, or added at the appropriate position to the input or output action in the action sequence.

Fields in the forms window can have formatting information associated with them regarding their actual appearance on the screen. The format of input fields, and of string output fields, is of variable length because their field lengths are not known until run-time. Boolean output fields have a single screen position allocated to them, and the length of this also cannot be edited. Integer and real output fields, however, can have their formats modified to adjust the number of digits displayed, and in the case of real outputs, the number of digits before and after the decimal place. The editing of these format fields is shown in figure 4.57.

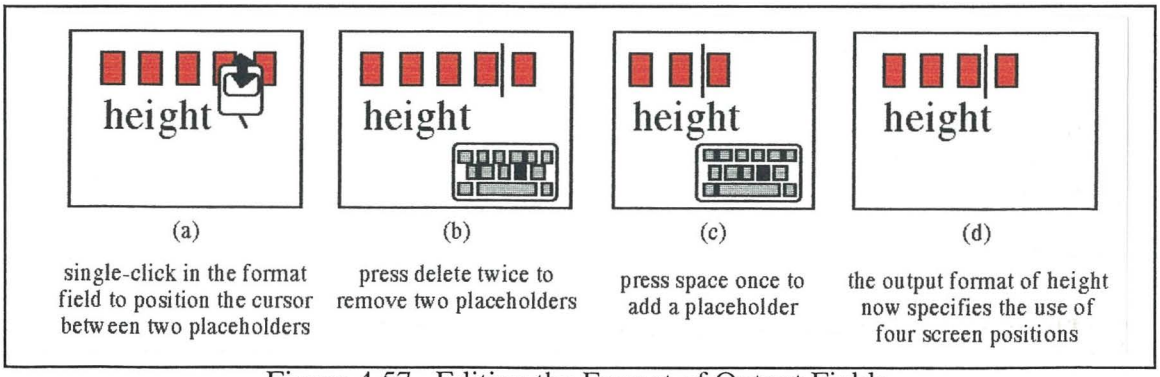


Figure 4.57. Editing the Format of Output Fields

When editing a format field with two or more placeholders, the cursor can be placed in-between two placeholders, and the entry of a space character adds a placeholder (as shown in figure 4.57. However, when the cursor is positioned outside either end of the format field (as is necessary when adding to a field that has been reduce to one placeholder), the entry of a space character can either mean "add a space at this position", or "add a placeholder to the field". To resolve this ambiguity, if the cursor is positioned adjacent to a format field, a placeholder can be added to that field by holding the shift key down while entering the space character.

It is useful to be able to add a complete input or output action to the action sequence while remaining in the forms window. This can be done by positioning the cursor outside of an existing input or output area and either adding an input or output action using a menu, or by typing "input" or "output" directly at that position, and pressing return.

An input or output action that is added to a forms window appears in the action sequence as near the top of the sequence as possible. This means that if an input or output is added as the first in the forms window, it will be the first action in the corresponding action sequence, and if added between two existing input/output actions in the forms window, will be placed directly after the first of these in the corresponding action sequence.

Editing Withs

A with component is used in HyperPascal to enable fields of a record (or a pointer to a record) to be referenced below that node in the tree as though they were simply variables of the appropriate type. The with component contains the record variable (or parameter) being referenced, and is associated with an extended comment via a hyperlink. The extended comment is edited using the extended comment editor as outlined previously, and the record is edited or modified using the type editor, or by editing in place.

Editing Choices

A choice icon has:

- a short comment, always visible, that can be edited in place;
- a hyperlink to an extended comment, editable using the extended comment editor;
- a number of expandable sub-icons, each of which contains a condition, and to each of which may be attached a conditionally executed sub-tree.

The choice component is initially created with one condition present in reduced form (as a small disk). More conditions may be added by a menu or hot-key operation, by adding together with child component (using the same process as shown previously in figure 4.xx), or by duplicating an existing condition (using a menu or hot-key operation). A user may select a condition, then delete it using a menu or hot-key operation. The order of the conditions can be altered by dragging them to different positions on the bottom of the choice component as shown in figure 4.58.

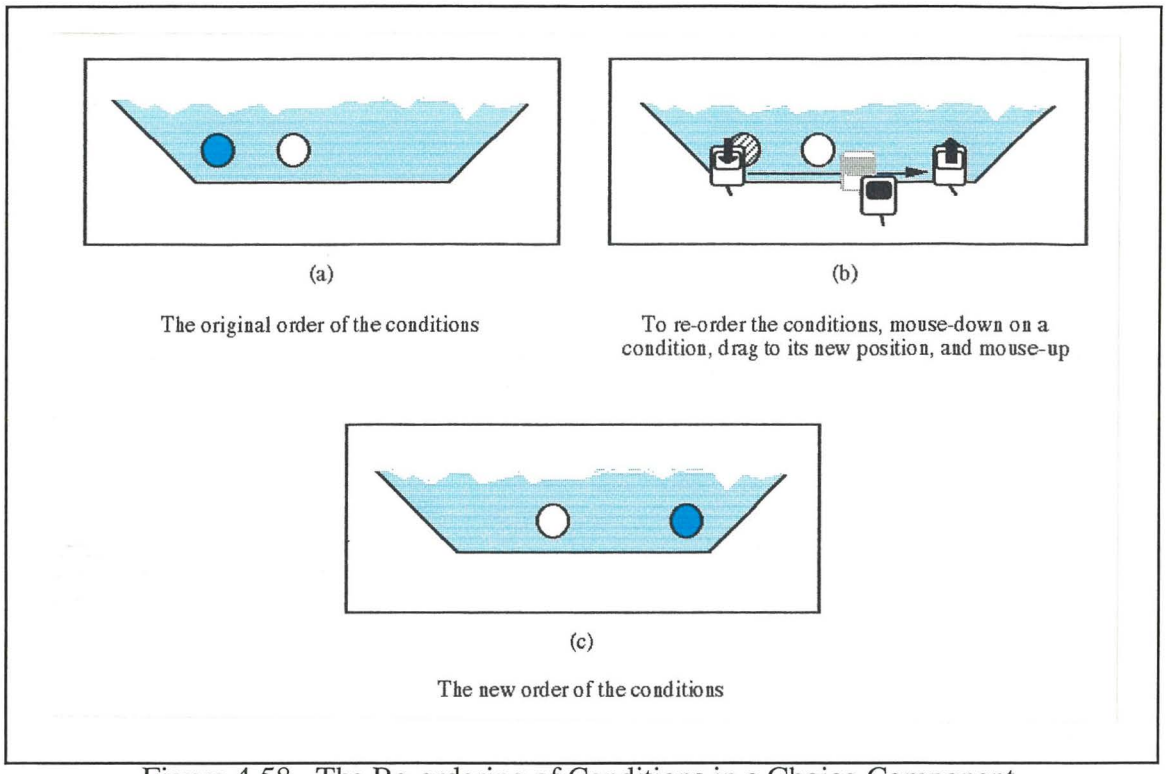


Figure 4.58. The Re-ordering of Conditions in a Choice Component

The user can expand a reduced condition disk by performing a double-click while the pointer is positioned over it, and can then edit the contained condition as though it were an expression in the donor field of an action. The expanded representation of a condition can be reduced again by performing a double-click in the condition area, but not on the condition itself.

A condition can have a sub-tree attached, which will execute when the condition evaluates to true. A child component (or sub-tree) can be attached to an empty space in a choice component, which causes the creation of a new condition with the component (or sub-tree) attached, but a child component (or sub-tree) can also be attached to an existing condition (as shown in figure 4.59).

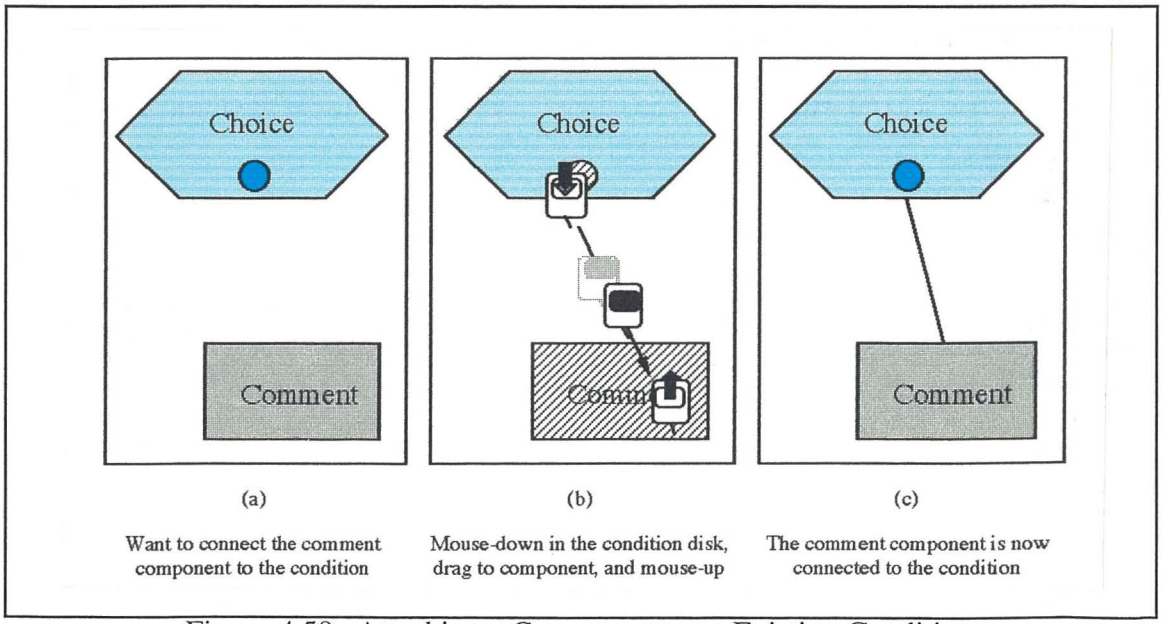


Figure 4.59. Attaching a Component to an Existing Condition

Once a component has a connector attached to a choice condition, the condition can be moved, with the connection attached. The component can be detached from the condition either by deleting the connector (which leaves the condition intact), deleting the condition (which also deletes the connector), moving the end of the connector to another condition (which attaches the connector to that condition, and removes any connectors already attached), or moving the end of the connector to an empty position inside the choice component (which creates a new condition and attaches the connector to that condition).

Editing Loops

A loop icon has:

- a short comment, always visible, that is editable in place;
- a hyperlink to an extended comment, editable by using the extended comment editor;
- attachment positions for:
  - sub-trees which are to be executed repeatedly;
  - conditions which are evaluated to control the number of repetitions of the loop.

The reduced representation of the pre-loop and post-loop action sequences can be expanded by double-clicking while the pointer is positioned over the reduced representation. The expanded representation of the pre-loop and post-loop action sequences are edited similarly to normal action sequences, and can also be associated with a forms window via a hyperlink. The expanded representation of the pre-loop and post-loop action sequences can be reduced again by double-clicking just inside their boundary. The expanding and reducing of a pre-loop action sequence is shown in figure 4.60.

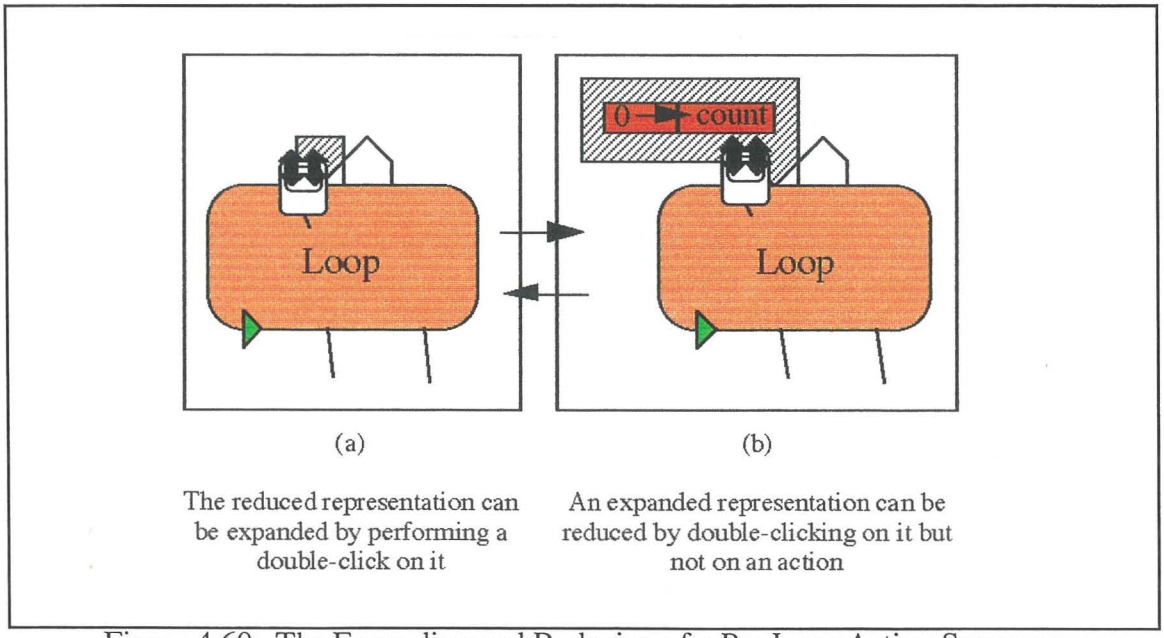


Figure 4.60. The Expanding and Reducing of a Pre-Loop Action Sequence

Condition sequences can be attached to a loop component anywhere in the sequence of sub-trees that it controls. These condition sequences are used to control the execution of

the loop, and also enable the association of actions with the condition check. Condition sequences can be added to a loop in a similar manner to the adding of children to a loop.

If a loop component is currently selected, it is possible to add a condition sequence to the loop using menus or hot-keys. If a continue condition sequence is added in this manner, it is placed as the left-most child of the loop (first in the loop's execution), and if a terminate condition sequence is added, it is placed right-most. A condition sequence can be added in an arbitrary position on the bottom of the loop using the direct manipulation method, previously described for adding a normal child component and connector at the same time. When adding a child to a loop component, the menu for the selection of components also includes entries for a continue condition sequence and a terminate condition sequence.

A condition sequence has a reduced representation in the form of a small icon on the bottom edge of the loop component. The reduced representation of a condition sequence can be expanded by performing a double-click while the pointer is positioned on the reduced icon. The expanded representation of a condition sequence has a representation similar to that of an action sequence, and is attached to the reduced representation icon with a connector. The condition sequence can be reduced again by a performing a double-click on the connector, on the reduced representation next to the condition (the condition indicator), or on the reduced representation on the bottom of the loop. The methods of expanding and reducing condition sequences are summarised in figure 4.61.

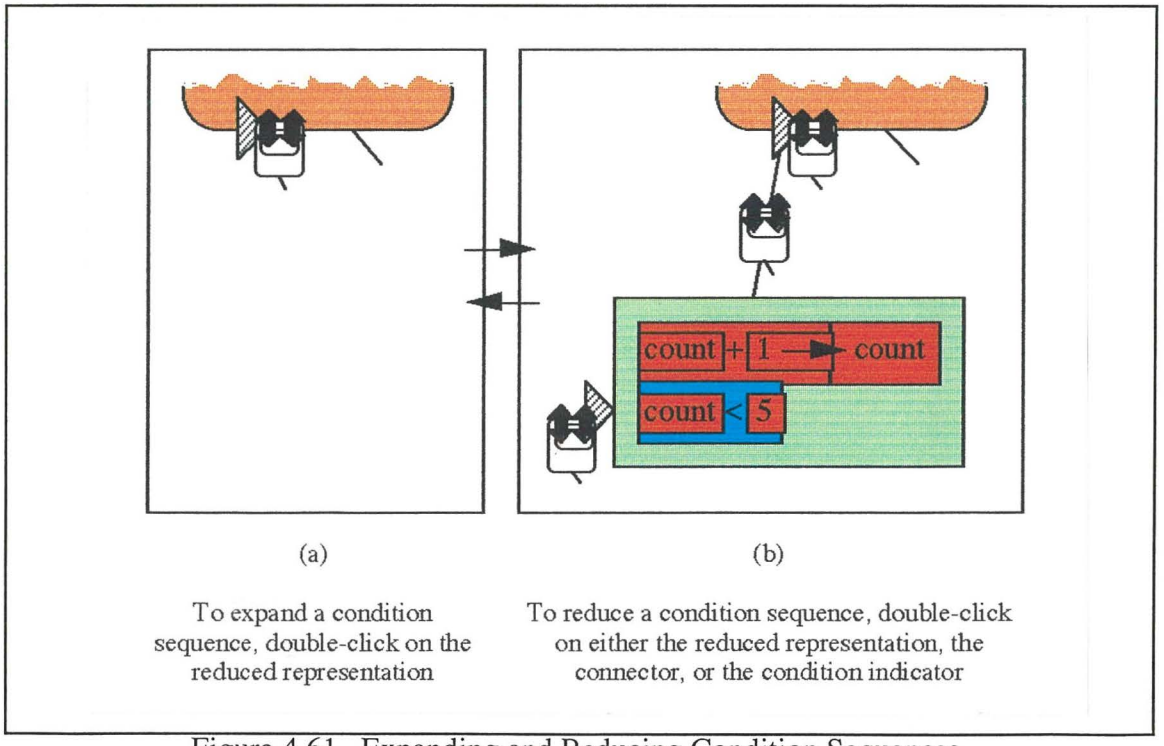


Figure 4.61. Expanding and Reducing Condition Sequences

Condition sequences can contain actions as well as a condition. These actions are created and edited using the same approach as that for action sequences. Condition sequences also contain a condition, which has an icon next to it indicating the condition type. The condition is edited the same way as expressions.

Chapter 5

The Implementation

The previous chapter described the notation and interaction with the HyperPascal hyperprogramming language. This program development environment for the language uses a single data structure to represent the program but editing operations take place on discrete subsets of this data structure, called views. The ability of this language to specify an executable program is used as a test of the validity of the hyperprogramming approach, so a prototype implementation of HyperPascal was required. This chapter describes the implementation of the HyperPascal prototype.

Since it was not the aim of this project to design and implement of a run-time environment is, HyperPascal programs are converted to Pascal source code for later compilation and execution using a Pascal programming environment. If HyperPascal hyperprograms can be converted to correct Pascal source code in this manner, the ability of HyperPascal to specify an executable program is confirmed.

A number of different programming approaches and languages were investigated for implementing the HyperPascal prototype. An object-oriented approach was chosen because of the ease with which object-oriented systems can be modified without the propagation of changes, and since a prototype implementation is required to be easily changeable. A number of object-oriented languages were investigated for the implementation of the prototype, including Think Pascal v4.0 (Symantec Corporation, 1990) and Think C v5.0 (Symantec, 1991a, 1991b).

During the course of these investigations, it transpired that John Grundy and John Hosking were developing a multiple graphical and textual view programming environment called SPE (Grundy and Hosking, 1993). A product of this research was an object-oriented framework for the development of programming environments that utilise multiple graphical and textual views, called MViews (Grundy, 1993).

It was decided to use the MViews framework for the implementation of the HyperPascal prototype, as it combines three desirable features, a library of standardised window-oriented graphics routines, object-orientation, and - most importantly - the automatic handling of between-view consistency checking and updating.

5.1. MViews

The HyperPascal software currently comprises a number of specialised editors for capturing the various components of a HyperPascal program, and a translator which uses

the information these editors have captured to produce Pascal source code which can be used as input data for a conventional Pascal compiler.

The system is ultimately written in LPA MacProlog v. 5.4 (LPA, 1992), but it makes use of two intermediate layers of software: Snart (Grundy, 1993) extends Prolog to support object-orientation, and MViews (Grundy, 1993), which is written in Snart, provides support for multiple-window graphical programming environments.

Overview

The MViews framework allows the construction of software development environments which support multiple views of a central program representation. MViews allows components present in the central representation of the program to be present (in part, or in whole) in more than one of the display views, and to be represented differently (if desired) in each of these views.

The display views in an environment constructed using MViews can be either textual or graphical, and are represented using the LPA MacProlog text and graphics windows. The graphics windows use a tool-based approach to graphical interaction, and since the MViews framework uses these windows, MViews also uses a modal approach to graphical interaction based on tool selection then use. The MViews framework also provides built-in high level support for interaction with these views, including the monitoring and updating of the current view and the currently selected objects in a view.

One of the most important features of MViews is that it handles consistency management between views automatically. When a component (or an aspect of a component) is modified in one view, this modification is automatically propagated to the central representation of the program, and then to the other views containing the component. This means that the underlying representation of the program, and the representation in a view being edited are consistent at all times (the other display representations of a component may, however, not be updated until their display view is made the current view).

The MViews framework also has built-in support for other, more general facilities desirable for a programming environment, such as the saving and loading of programs, and the assigning of unique identifiers to program components.

In summary, the MViews framework can be specialised to create an environment which utilises multiple views with automatic updating between views. It also allows the specification and management of a multiple window graphical interface to be performed easily.

Inside MViews

MViews represents programs as a collection of directed graphs, each of which is grouped into *views*. The MViews framework uses three types of view:

- *The Base View* -Every structure in an MViews environment has one, and only one, base view which contains the complete program graph.

- *Subset Views* - A program can have a number of subset views, each containing a representation of a subset of the components present in the base view. A component in the base view may be represented in more than one subset view.

- *Display Views* - The display views are subset views that also define how the subset representation of components is to be displayed and interacted with by the user.

An update of a component in one view is automatically propagated to occurrences of that component in any other views.

Figure 5.1 shows the MViews object-oriented inheritance hierarchy. Specific software development environments are implemented by specialising this class hierarchy to suit the needs of the environment. The classes that make up the hierarchy are now discussed in more detail.

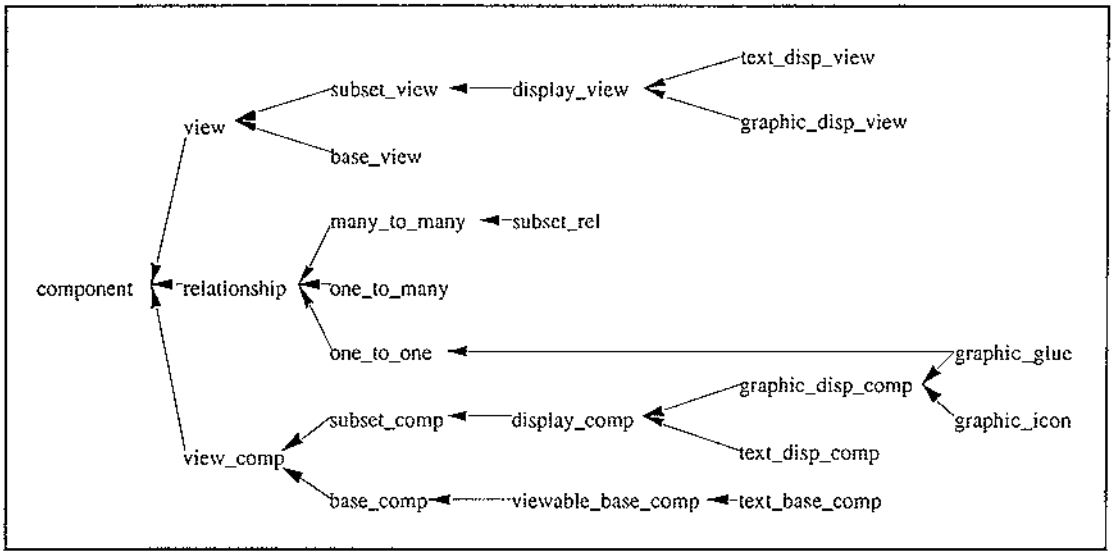


Figure 5.1. The MViews Framework Inheritance Hierarchy

The identification of components and the handling of updates, are common to all classes in the hierarchy, so the necessary state information and operations to supply this functionality are implemented in the *component* class.

There are three broad classes of component in MViews: *views*, which provide common methods for adding, removing and managing visual components, and which contain representations of a subset of the base components; *relationships*, which link components and are specialised to three subclasses, *one-to-one*, *one-to-many*, and *many-to-many*; and *view_comp*(onents), which are used to represent the actual components (*base_comp*(onents)), and their representations (*display_comp*(onents)).

The *base_comp* class defines the functionality and state that is common to components present in the base view, which includes the ability to "own" views, and keep a record of its subset representations, and the views in which they occur. Instances of the *base_comp* class (and classes that inherit from it) are used to form the underlying base view of a program.

The state and operation semantics common to all components in subset views is implemented in the *subset_comp* class. The *display_comp* class inherits the subset component's state and operation semantics from the *subset_comp* class, and is used to define the semantics for displaying and editing a subset representation of the component.

A component may be represented either textually, or a graphically. The state and methods necessary for displaying a component as text are defined in the *text_disp_comp*

class. Likewise, the state and methods necessary to display a component in a graphical form are defined in the *graphic_disp_comp* class. Components displayed in a graphical form can either be members of the *graphic_icon* class, or of the *graphic_glue* class. Graphic icons are used to represent nodes in a graphical representation, while graphic glue components are used to graphically represent the relationships between the nodes. The *graphic_glue* class also inherits from the *one-to-one* relationship class, and therefore shares the behaviour and state information of a one-to-one relationship.

Relationships are used for linking two or more MViews components together, and can be one-to-one, many-to-one, or many-to-many. The common state and methods of these three types of relationship are implemented by the *relationship* class, and those specific to each type are implemented in the *one-to-one*, *one-to-many*, and *many-to-many* classes respectively.

The *view* class provides common methods for adding, managing and removing components in a view, and maintains a one-to-many relationship between a view and the components it contains. The *view* class is specialised to the *base_view* and *subset_view* classes. The *base_view* class implements additional program management operations such as the allocation of unique identifiers to components and views, and keeping track of the current view. The *subset_view* class groups subset components to form a partial copy of the base view of the program, and includes a "focus" component, which "owns" the view and cannot be deleted (unless the focus is shifted to another component).

The *display_view* class is a specialisation of the *subset_view* class, and defines the general information associated with the display and interactive editing of components, and the management of the actual window in which the components are displayed. The *display_view* class is further specialised into the *graphic_disp_view* and *textual_disp_view* classes which provide extra state and operations specific to each kind of displaying and editing approach.

The prototype implementation of HyperPascal is specialised from these classes.

5.2. The Prototype Version of HyperPascal

The experimental verification of a new programming paradigm is the generation of a complete and consistent language based on that paradigm. In order to obtain such verification we designed HyperPascal, and an editing/parsing environment to provide further substantiation of its consistency and generality. The entire design of HyperPascal does not need to be implemented in order to satisfy this objective, and the prototype therefore implements a "cut-down" version of HyperPascal, as detailed briefly below.

The HyperPascal prototype only allows the declaration of variables and parameters of simple types (integer, real, Boolean, and string), or one dimensional array variables of these simple types. The arrays can be of any length, and HyperPascal allows the programmer to define the upper and lower indexes of the array.

The graphical representation of declarations, expressions, and actions is simplified in the prototype implementation. The declarations of a subprogram's simple variables and parameters are colour-coded in the subprogram header according to their type, but array variables have the same appearance as simple variables, except that they are colour-coded grey. The appearance of variable and parameter declarations can be seen in the screen dump of an action tree from the prototype, shown in figure 5.2.

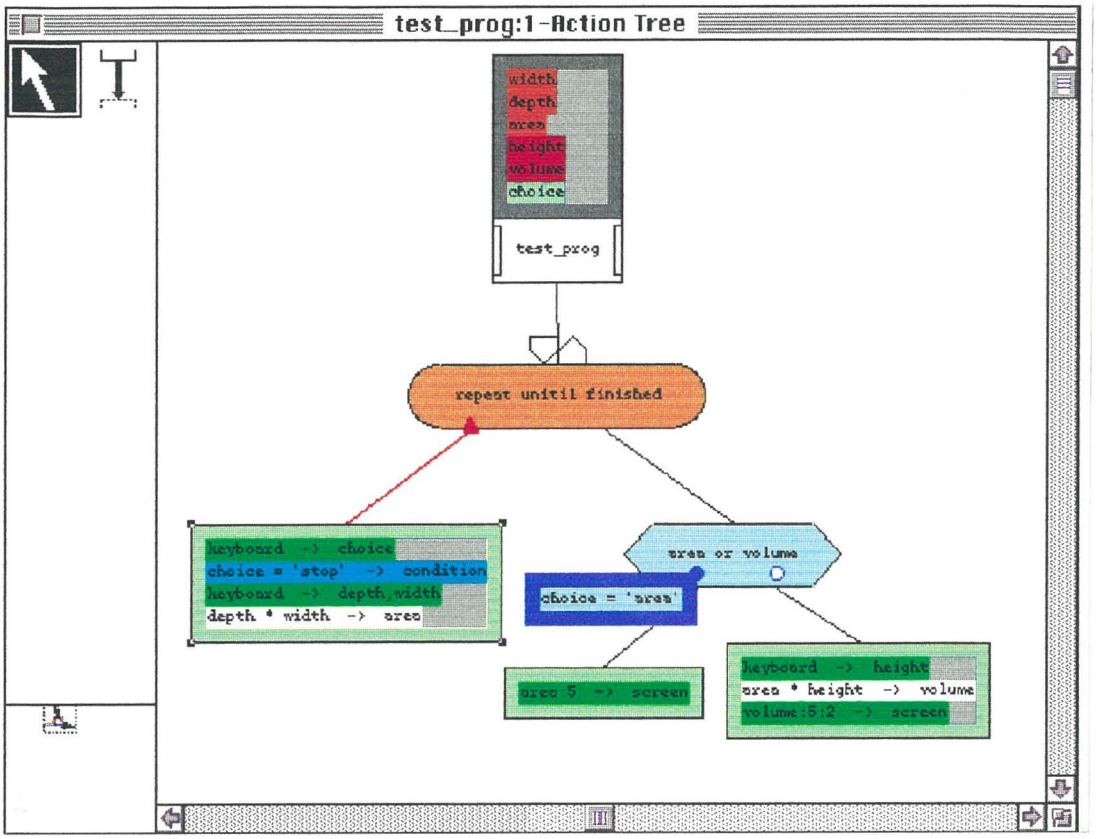


Figure 5.2. Screen Dump of an Action Tree from the HyperPascal Prototype

Expressions are represented as normal text in the prototype language, and no check is performed on the types of the variables used in the expression (hence the lack of colour coding in this prototype example). Complex expressions, such as those involving the referencing of an array variable or a subprogram invocation, are allowed, but follow the syntactic rules of the same expression in traditional Pascal. In subprogram invocations with multiple input and output parameters, the order of parameters is input-only parameters first, input-output parameters next, then the output-only parameters last. The order among parameters of a particular type remains as shown in the subprogram header.

Actions also have a simplified textual representation in the prototype language. They do, however, retain the donor-acceptor assignment format, with an expression in the donor field, and a variable or output parameter in the acceptor field. Values can be returned by a subprogram with a return type, by having the subprogram invocation in the donor field, and a variable or parameter of the correct return type in the acceptor field. The value is returned from within the subprogram's action tree by assigning it to the subprogram name (analogous to a Pascal function). The invocation of a subprogram without a return type (analogous to a Pascal procedure) is performed by having the subprogram invocation in the donor field, and leaving the acceptor field empty.

The prototype implementation omits many of the editing-in-place techniques described in chapter 4 (in the graphical views), and instead makes use of dialogs to enter and modify information. The editing of a declaration or action is performed using dialogs that appear when the user double-clicks on a group of actions or declarations. These dialogs can be used to add to, remove from, or to modify the elements present in the list of declarations or actions in the declaration area, or action sequence. The current system does not recognise a double-click on a single action, so the forms window for an action sequence is found by double-clicking on the area in an action sequence surrounding the actions. In the prototype implementation, the forms windows are simple text windows, and the input and output parameters are shown in normal text, surrounded by double-angle brackets to signify that they are input and output parameters. Individual input and output actions are

also shown using normal text surrounded by special characters. The appearance of a forms window is shown in figure 5.3.

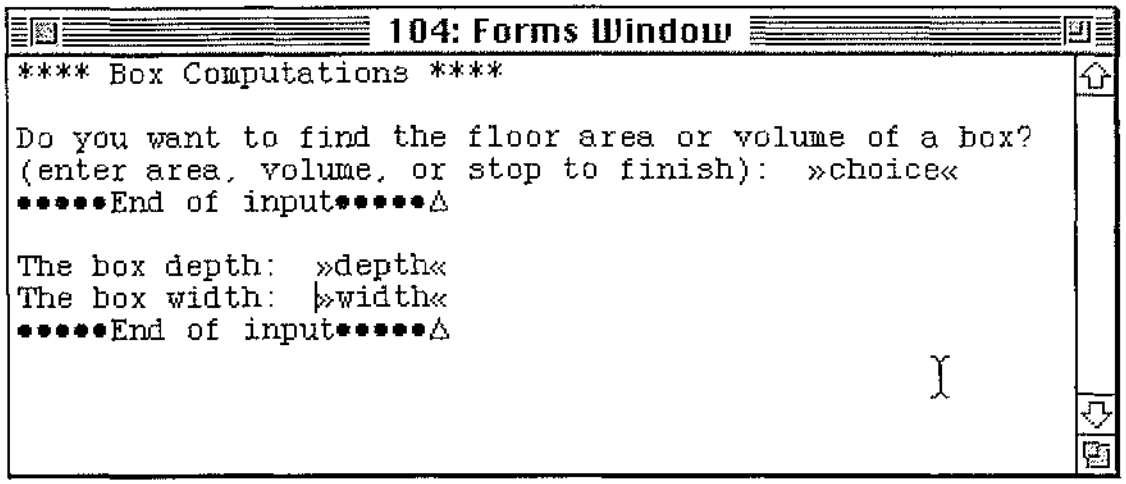


Figure 5.3. A Forms Window in the Prototype Implementation of HyperPascal

5.3. Inside the HyperPascal Prototype

The underlying representation of a HyperPascal program is composed of base components and the connectors between these components, which combine to form a directed graph representation of the program. A number of different types of base components and connectors are used in HyperPascal to represent the different program components and the relationships between them. The different classes of components and connectors in HyperPascal, and their inheritance from the MViews framework, are shown in figure 5.4.

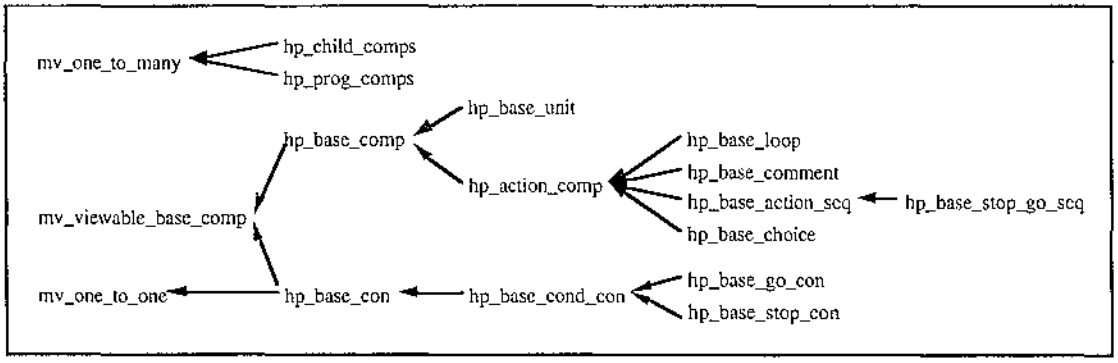


Figure 5.4. The Base Components and Connectors

The base components shown in figure 5.4 are contained by the overall view of the program, which is called the base view. The base view of the program cannot be viewed directly by the user, but can be viewed using the scope tree and action tree graphical display views, and the forms window and extended comment window textual display views. Each of these display views allows the viewing and editing of a subset of the components and connectors from the base view. These different views, and their inheritance from the MViews framework are shown in figure 5.5.

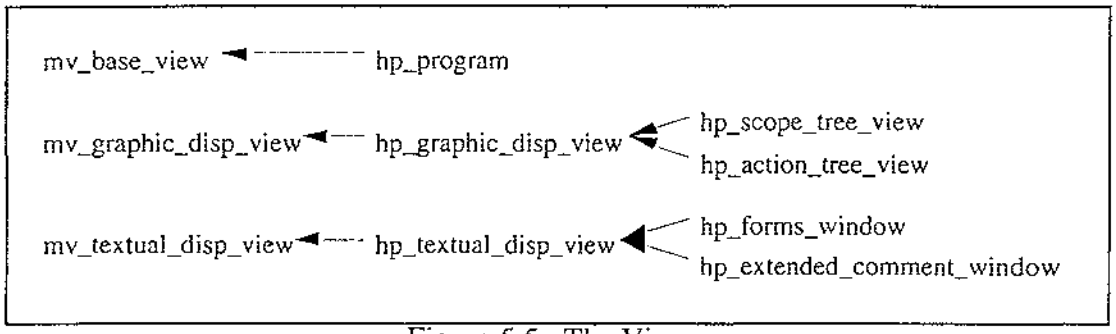


Figure 5.5. The Views

The components and connectors of a single base view can be viewed and edited in many display views. Each of these views may show a different subset of the attributes of the same base component or connector, and may also represent the component or connector in a different manner. Instead of the base component storing the information regarding the different viewable attributes, and the different representations of a component which can appear in different views, the base view simply has a relationship with a number of display components which hold this information.

A display component can only be present in one view, and contains a subset of the state information of the base component. The state information in a base component and its display components is kept consistent by the passing of update records between the components whenever one is updated. The display component also takes complete responsibility for the representation of, and interaction with that component. The display components, and their inheritance from the MViews framework are shown in figure 5.6.

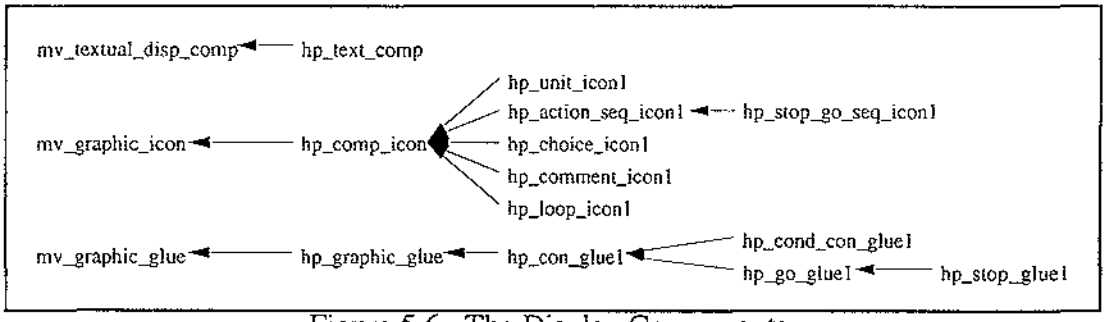


Figure 5.6. The Display Components

In summary, the complete representation of a program is present in the base view, but can only be viewed and edited using the action tree, scope tree, forms window, and extended comment window display views. The components viewed and interacted with in these views are display components, which contain a subset of the state of the corresponding base component, and automatically pass updates to the base component (which passes it on to other display components) if their state is modified.

The implementation of the HyperPascal prototype is now discussed in more detail.

The Base View and Program Parsing

The base view of a HyperPascal program contains full representations of all the program components and the relationships between them. These base components and relationships are used to hold the state of the program, and are the elements that are used to create the Pascal source code that can be later compiled and executed.

Base components are used to contain the information specifying the subprogram declaration and each of the programming constructs, whereas the base connectors are used to represent the relationships (and any attributes of these relationships) between constructs. The base components include both the different types of base action components (the comment component, the action sequence component and condition sequence, the choice component, and the loop component), and the one type of base declaration component (the program unit component). The connectors between these components can be simple connectors, continue or terminate connectors (which indicate the kind of condition sequence attached to a loop), or conditional connectors.

The first sections of a HyperPascal program to be parsed are the declarations in the root program unit component (the program). The program unit components, and the action components of each program unit are parsed in a depth first manner. The Pascal source code is written to an output window as the HyperPascal program is parsed.

A program unit component contains attributes holding the visible text and return type of the subprogram, a list of variable declarations, a list of input parameter declarations, and a list of output parameter declarations. In the prototype version of HyperPascal, the program header (the root program unit) has a name, but its return type, and input and output parameters are not parsed.

A subprogram can be declared as a procedure or a function in Pascal, and the classification of a subprogram into one of these types is dependent on the subprogram's return type. If it has a valid return type, it is declared as a function with that return type, otherwise it is declared as a procedure. The name of the declared function or procedure is held in the visible text attribute.

A HyperPascal program unit for a subprogram is allowed to have multiple input and output parameters, each comprising a name and a type. The program unit component holds a list of these input and output parameters, and these are parsed into the corresponding Pascal function or procedure parameters. Parameters present in the input parameter list are declared to be normal parameters, and those from the output parameter list are declared as variable parameters. It is also possible for an identifier to be declared as an input/output parameter, and these are also declared as variable parameters. The parameters of a HyperPascal program unit are converted to function or procedure parameters. HyperPascal parameters which occur in the output list are converted to Pascal *var* parameters, and HyperPascal parameters which occur in only the input list are converted to Pascal non-*var* parameters. A HyperPascal subprogram declaration, and its Pascal equivalent are shown in figure 5.7.

The HyperPascal Data Structure		The Pascal Equivalent
subprogram name:	inout_test	**procedure** inout_test(input1:integer;
		input2:string;
return type:	None	**var** inout:real;
		input3:boolean;
inputs:	input1 - Integer	**var** output1:real;
	input2 - String	**var** output2:string);
	inout - Real	
	input3 - Boolean	
outputs:	output1 - Real	
	inout - Real	
	output2 - String	

Figure 5.7. The Pascal Equivalent of Subprogram Parameters

A program unit component contains a list of variable declarations for the sub(program), each comprising a name and a type. The variable declarations for a Pascal function or procedure are also a list of declarations, each comprising a name and a type. Since the different declaration types in HyperPascal are a subset of those available in Pascal, and the format of the declarations is so similar, the converting of the HyperPascal declarations is done by positioning the names and types of the declarations (interspersed with some formatting to make them conform to Pascal syntax) after the function or procedure header in the Pascal source code.

A program unit component contains a one-to-many parent-child relationship with other program units, and this relationship is used to represent the scope of declaration of subprograms, with the children in the relationship being declared within the parent. In HyperPascal, the scope of subprogram declarations is important, but their order of declaration within that scope is not. This is not true of Pascal, which does not allow a function or procedure to be used before it has been defined in the text. To allow the use of a function or procedure that is within the same scope, but later in the text of the program, Pascal allows forward declarations of functions and procedures. These forward declarations declare the function or procedure name, parameters, and (if declaring a function) the return type of the function or procedure before it is used.

Since a HyperPascal program is converted to a linear Pascal source document, this forward referencing problem can be solved simply by forward declaring all subprograms. The parsing of the root program unit declares the program header and the program's variables, then its program_units relationship is searched, and the return type, and parameter information forward declared for each of the subprograms contained. The program_units relationship is then parsed in a depth-first manner using a similar method to the parsing of the root program unit (except that the parameters and return type of a program unit are forward declared, so are not declared in the program unit declaration).

A program unit also contains a one-to-many parent-child relationship with a number of actions. After all of the children in a program unit's parent-child relationship have been parsed (or if a program unit has no child program units), the actions relationship of the program unit is parsed before the traversal returns to the parent program unit. This traversal method results in complete subprograms (their declarations and actions) being contained within the scope of other subprograms. A simple HyperPascal program unit structure and the structure of its corresponding Pascal source code are shown in figure 5.8.

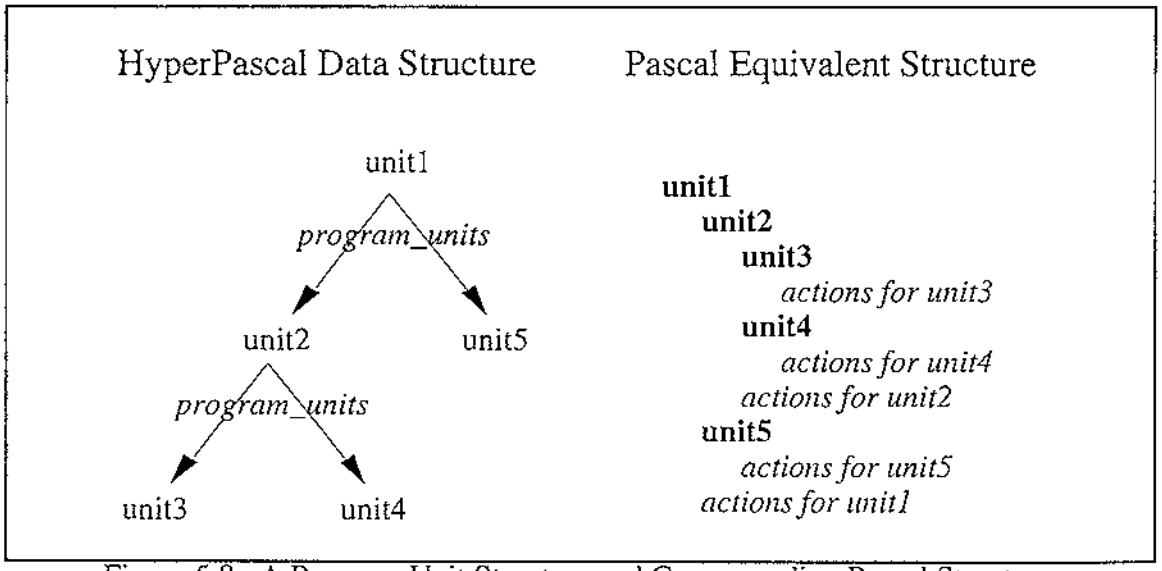


Figure 5.8. A Program Unit Structure and Corresponding Pascal Structure

Since forward declarations are generated for subprograms, the order of the subprograms in a program_units relationship is not important. The actions of a subprogram, however, have a fixed order, and this order must be maintained in order for the program to be parsed correctly. The program unit's actions relationship must therefore be sorted according to the required order of the program unit's actions.

The actions of a program unit are obtained by a depth-first traversal of the actions relationship of the program unit. The actions relationship contains an ordered list of connectors which are attached to base action sub-trees. The base action components that form these sub-trees are now discussed.

The comment component is the simplest of the base action components, containing only the visible text attribute. This visible text attribute of the comment is converted to a short comment in the program text at the current position in the program being generated. The comment component also has an extended comment associated with it, but as the Pascal source is used merely as an intermediate step in the execution of a HyperPascal program, including the extended comment in the Pascal source would slow the parsing process too much to make it worthwhile (the extended comment is discussed in more detail later).

The comment component also contains a one-to-many relationship with a number of other action sub-trees. After the visible text attribute of the comment component has been converted to a comment in the source code, the action sub-trees in this relationship are parsed with a depth-first left to right traversal.

The action sequence component contains a list of actions, each of which has a donor attribute, and an acceptor attribute. These actions are parsed in order from the beginning of the list, to the end of the list, and form the actual data manipulation actions of the program. In HyperPascal there is a convention that information travels from left to right, and the assignment action follows this convention. That is, the value to be assigned appears on the left hand side of the assignment action, and the variable receiving the value appears on the right hand side. This is the reverse of the situation that applies in Pascal, so when a HyperPascal assignment is converted to Pascal, the order is reversed.

Complex assignment actions such as the assignment of a value to an array variable, or the invocation a function, must have the same format as the Pascal equivalent, as they are written directly to the Pascal source code. The parameter order of a subprogram invocation is as specified previously, with the input-only and input-output parameters first, and the output parameters last. Procedure invocations are produced from HyperPascal assignment actions in which the donor field is occupied by the procedure invocation, but the acceptor field is left empty.

Input actions are recognised by the parser because they have the keywords "screen" or "keyboard" in either their donor or acceptor attribute. An input action contains a comma-separated list of variables in the acceptor attribute, and "keyboard" in the donor attribute, whereas an output action contains the list of variables in the donor attribute, and "screen" in the acceptor attribute. The action sequence can also "own" a forms window view, which contains the formatting information for the input and output actions in the view. If no forms window view is owned, a default format is used.

The parsing of an output action starts with a check to see if the action sequence "owns" a forms window view. If a forms window view is owned, it is searched to find the formatting information for the output action. The first output parameter is then found, and the action's formatting information (up to the first output parameter) is converted into *write* and *writeln* statements in the Pascal source window. After this, or if no forms window is owned, the first output parameter is converted into a *write* statement. The parsing continues, generating the formatting information between occurrences of output

parameters, then the output parameter, until the last output parameter is generated. After the last output parameter has been generated, the formatting information between it, and the end of the output action, is parsed. The parsing of an input action is similar to that of the output action except that the parameters are converted to Pascal *readln* statements. The forms window view is discussed in more detail later.

An action sequence does not have a parent-child relationship (it is a terminal node), so after the actions have been parsed, the traversal continues in its parent.

A choice component has a visible text attribute, which holds a short comment about the purpose of the choice, and can have an extended comment associated with it. The main functionality of the choice component is contained in its one-to-many relationship with the connectors to its sub-trees. These connectors (conditional connectors) contain the condition that must evaluate to true for the attached sub-tree to execute. This condition must either be a valid Boolean expression, or "else" for a default condition.

When a choice component is being parsed, the visible text attribute is first written to the source output window as a comment, then the rest of choice component is parsed. A different number of child connectors (and their conditions) results in a different Pascal *If* structure. If the choice component has one condition connector attached (with a valid expression in its condition attribute), it is converted into a simple *If* statement with its condition being the expression in the connector's condition attribute, and no *else* clause. The sub-tree the connecter is attached to is converted into the actions that will be executed if the *If* condition evaluates to true.

If the choice component has more than one condition connector attached, it is converted into a nested *If-Then-Else* structure. In this structure, the *If* conditions correspond to the conditions contained in the conditional connectors, and the actions executed if a condition evaluates to true are taken from the sub-tree that is attached to the corresponding conditional connector. A condition connector with the default condition must always be attached to the choice component as the right-most child connector. Some different choice structures and their equivalent Pascal source code are shown in figure 5.9.

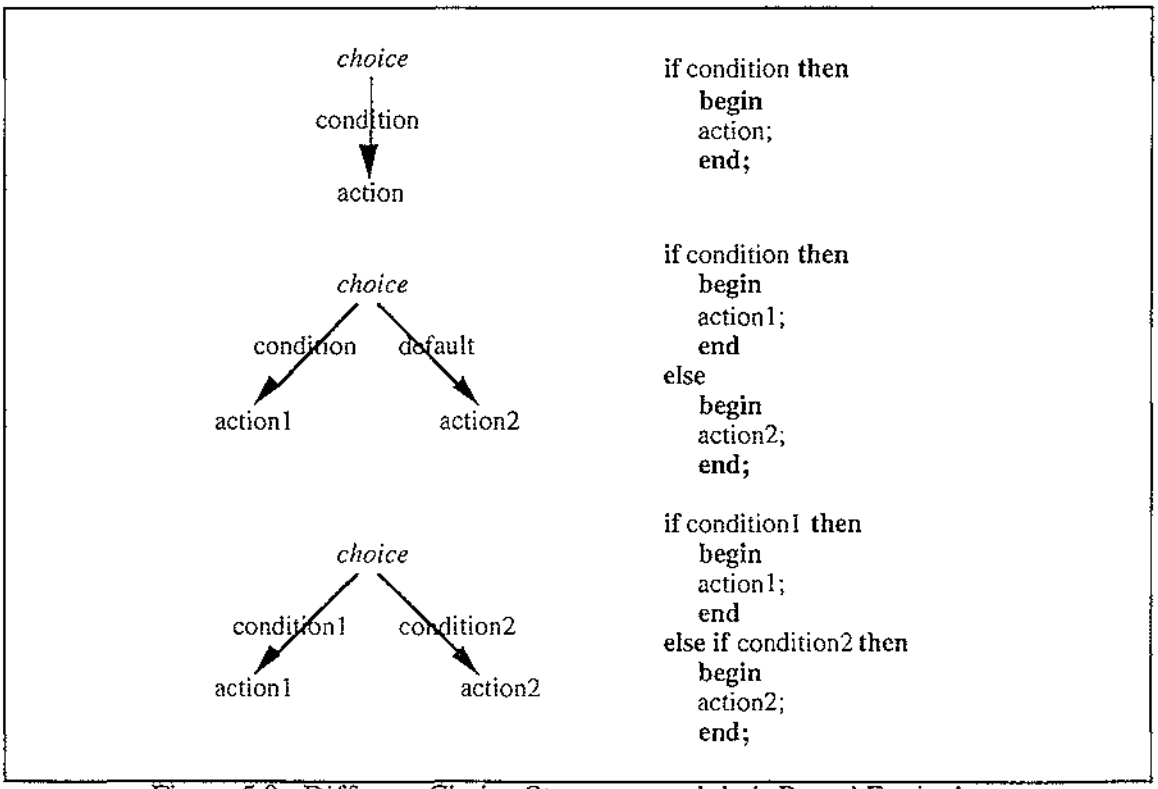


Figure 5.9. Different Choice Structures and their Pascal Equivalent

The loop component has a visible text attribute, which is used to hold a short comment, and can be associated with an extended. A loop component also contains lists of pre-loop actions and post-loop actions, which are parsed in the same way as action sequences. The visible text attribute is converted to a Pascal comment in the source window, then the pre-loop actions are parsed. The children of the loop component are then parsed (as discussed below), and finally, the post-loop actions are parsed before the traversal returns to the loop component's parent.

All HyperPascal loops are parsed into Pascal *While* loops. To enable the checking of multiple conditions in the loop, the underlying loop condition is a flag variable that is declared by HyperPascal in the Pascal source code. The name of this flag is the unique identifier of the loop component prefixed with the string "h_c_", so is not duplicated in the same program, and is not likely to be duplicated by accident as one of the user's variable declarations. This condition flag is initially set to true, so the loop will always be entered at least once (but no user actions are necessarily executed).

The loop component has a one-to-many relationship with a number of connectors to action sub-trees. These connectors can either be normal connectors, in which case the sub-tree is parsed normally, or they can be continue or terminate connectors, which connect the loop with a condition action sequence. A condition action sequence is an action sequence that can contain a condition as well as other actions. The condition is specified as an action with a Boolean expression in the donor, and "condition" in the acceptor.

When a condition is reached in the traversal of the condition sequence, an assignment to the loop condition flag is written to the Pascal source code. If the condition sequence is connected to the loop using a go connector, the condition's donor expression is assigned to the loop flag. If it is connected using a stop connector, the donor expression is negated in the assignment to the loop condition flag. The execution of the rest of the actions in the loop is dependent on the result of this expression, so a Pascal *If* statement is inserted into the source code after the assignment to the condition flag. The same expression assigned to the loop condition is used as the condition in this *If* statement, which controls the execution of the subsequent actions in the loop. A record is kept of the number of these open-ended *If*s so that they can be closed off after the parsing of the right-most action sub-tree of the loop. A loop component and its corresponding Pascal source code is shown in figure 5.10.

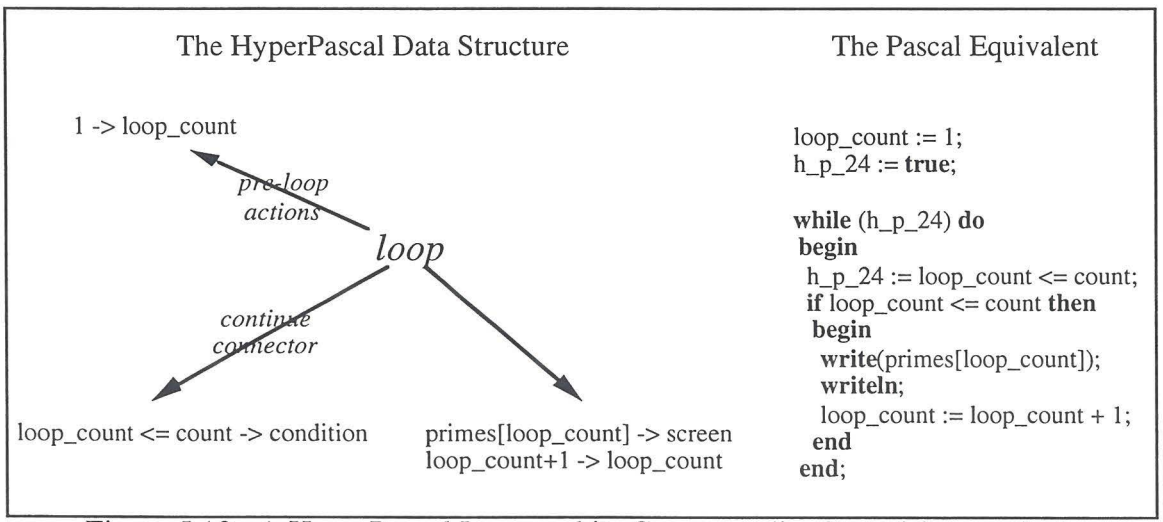


Figure 5.10. A HyperPascal Loop and its Corresponding Pascal Source Code

The components discussed so far have been the base components, which form a full representation of the HyperPascal program. These components are present in the base

view, which is not displayed, and therefore the user cannot see and interact with the base components. Viewing and interacting with a HyperPascal program are performed using other views of the program called display views. These display views present, and allow the user to interact with, different aspects of the underlying representation.

The different display views are the scope tree view (shows the program units, and their scope), the action tree view (shows the control structure and actions of a program unit), the forms window view (shows the formatting information associated with inputs and outputs), and the extended comment view (shows the extended comment associated with a component). The different display views implemented in the HyperPascal prototype are now discussed.

The Display Views

The scope tree view of a program shows representations of the base program unit components (displayed as unit icons), and the scoping relationships between them (displayed as graphical connecters). These display elements contain a subset of the data present in the base elements, and any modification of the data in a display element is automatically updated in the corresponding base element. This automatic updating allows the user to modify the base representation of a program by modifying a displayed representation. An example of a scope tree view showing the unit icons, and graphical connectors, is shown in figure 5.11.

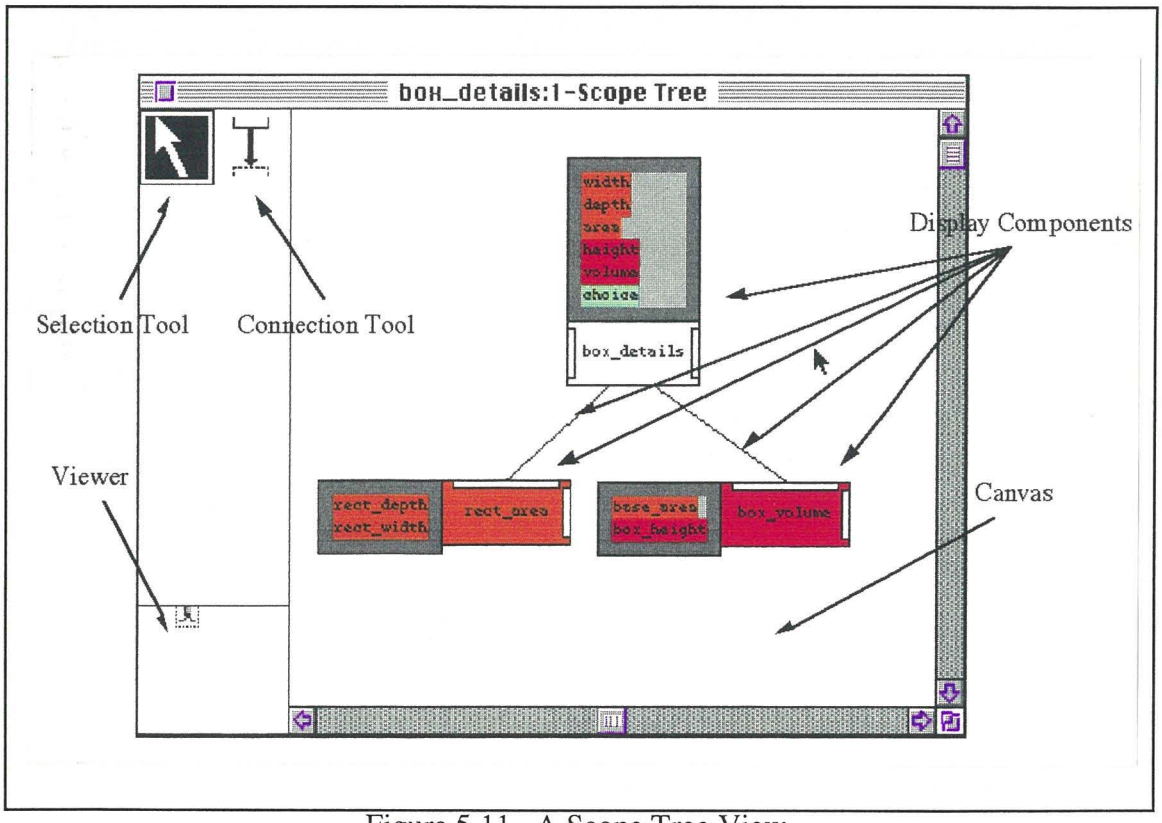


Figure 5.11. A Scope Tree View

Figure 5.xx points out the different parts of the scope tree view (many of which are also common to the action tree views - explained later). These different parts include the selection tool, the connection tool, the viewer, the canvas, and the display components (the unit icons and graphical connectors). These different parts of the scope tree view are now discussed.

The canvas is the area of the scope tree on which the display elements are created and modified. The canvas only shows a small portion of the complete view, while the viewer shows the complete view in a reduced version, and also shows the area of the view that the canvas is displaying. The representation of the view can be reduced in scale so that the whole view fits on the canvas by selecting "Toggle Zoom" from a menu. The representation can be expanded to original size by selecting "Toggle Zoom" again.

The selection tool and the connection tool are selectors and indicators of the current editing mode. When the connection tool is selected, the mouse pointer changes to a cross-hair, and the user is able to add connections between existing components, or add new components connected as the children of existing components.

A graphical connector can be added when the connection tool is selected by performing a mouse-down on the desired parent unit icon and dragging to either the desired child unit icon, or to unoccupied space on the canvas, before performing the mouse-up. If the mouse-up occurred on another unit icon, a graphical connector is added between the two icons, and a connector is added between the two corresponding base components. If the mouse-up occurred in an unoccupied space on the canvas, a dialog is activated for entering the name and return type of the new program unit, a new unit icon appears on the canvas with a graphical connector from it to the parent unit icon, and a corresponding program unit is added to the base view with a connector to the corresponding parent program unit.

When a graphical connector is added, it is shown connected on the bottom edge of the parent unit icon, with its horizontal position determined by the horizontal coordinate of the mouse-down inside the parent component. The graphical connector is connected on the top edge of the child unit icon, and in the horizontal centre of the icon. These connection points can be modified using the selection tool.

When the selection tool is selected, the pointer is an arrow shape, and the user is able to select, re-arrange, and modify the display elements. A display element is selected by clicking once inside the rectangle formed by its maximum and minimum horizontal and vertical bounds. The selection of an element moves it to the front level of the display. Re-arrangement of the display elements is performed by either clicking within a unit icon's area and dragging to a new position, or clicking on one end of a graphical connector, and dragging to a new horizontal coordinate within the same unit icon.

Modifications of the unit icons are performed using a double click to expand or reduce declaration areas, and a double click on a declaration (or group of declarations) to activate a dialog for its (their) modification. When a unit icon is double-clicked on, a check is made to find what part of the icon the pointer was on when the double-click occurred, and the appropriate action performed (as described below).

If the pointer was over a collapse bar, the collapse bar is expanded into a declaration area, and the contained declarations are displayed. If the pointer was inside a declaration area, a check is made to find if it was on the contained declarations. If the pointer was on the unit icon's declarations, a dialog is activated for the editing of those declarations (resulting also in the update of the base program unit's declarations). Otherwise the declaration area is reduced to a collapse bar. If the pointer was over the main body of the unit icon a check is made to find if it was on the name of the unit icon. If the pointer was on the name of the unit icon, a dialog is activated for the modification of the unit icon's name and return type (resulting also in the modification of the corresponding attributes in the base program unit), otherwise, the action tree view for the program unit is activated for editing.

The action tree view shares the same overall format as the scope tree view, except that instead of representing the base program units and relationships between them, it represents (using icons and graphic connectors) one base program unit, the base actions

of the program unit, the relationships between the program unit and the actions, and the relationships between the base actions and other base actions. The base program unit is represented by the unit icon, the base comment by the comment icon, the base action sequence by the action sequence icon, the base choice by the choice icon, and the base loop by the loop icon. The relationships between the base components are represented using different graphic connectors between the icons. The different graphic connectors are the graphic connector, the terminate graphic connector, the continue graphic connector, and the conditional graphic connector. Many of these graphical elements are shown in the action tree in figure 5.12.

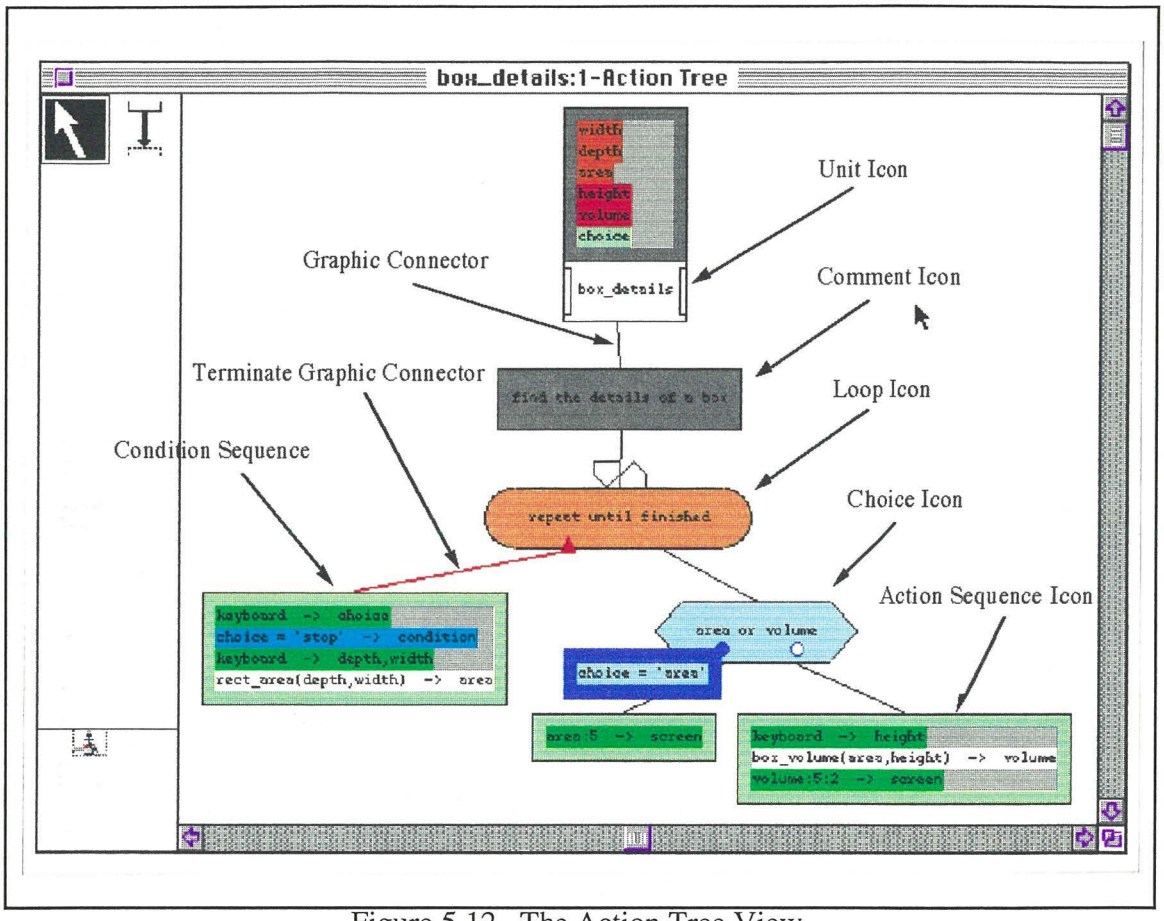


Figure 5.12. The Action Tree View

The operation of the connection tool in the action tree view is similar to that in the scope tree view except that, since more than one type of component can be added to the view, when a child component is being added, the user must choose the type of component to be added.

The selection tool is used for selection and re-arranging icons in the action tree view in the same as it is in the scope tree view. The method used to modify the state of the icons is dependent on the type of icon, but the method is essentially the same as that for modifying the icons in the scope tree. The differences are explained while discussing the icons, below.

The unit icon has the same representation in the action tree view and in the scope tree view. Modification of the unit icon is performed in the same way in the action tree as in the scope tree, but the area used, in the scope tree for activating the action tree, now activates the scope tree. Any modifications of the state of the unit icon cause the state of the corresponding base program unit, and also the unit icon in the scope tree, to be automatically updated (modifying the unit icon in the scope tree view also causes an update in the unit icon in the action tree view).

The comment icon is a graphical representation of the base comment component, and is used both to graphically display the short comment and to enable access to the extended comment. When the user selects the selection tool and then double-clicked on the icon, the system checks to find which region the pointer was over when the double-click occurred. If the double-click was over the text in the icon, a dialog is activated for editing the short comment text in the icon (the visible text attribute of the base comment is automatically updated). If the double click was inside the icon but not over the text, the extended comment window view for the base comment is activated or, if the base comment does not have an extended comment already associated, one is created then associated with the base comment. A comment icon is shown in figure 5.13.

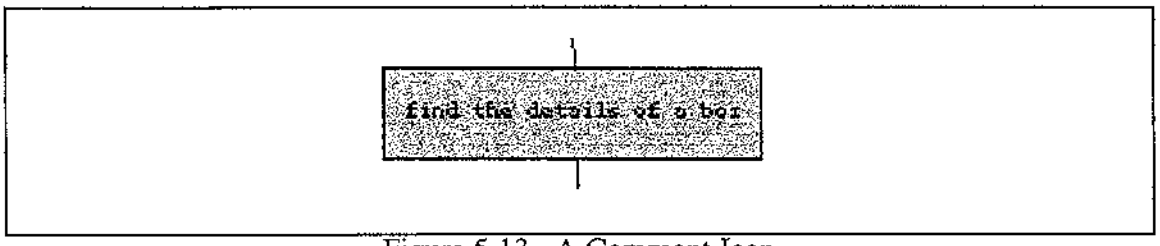


Figure 5.13. A Comment Icon

An extended comment window view is a textual view that allows the free-editing of text. The view is initially created without any text inside it, and with a window name based on both the base component's unique name, and the component's visible text. This window name ensures that no other window has the same name, and the windows are identifiable easily with the associated component. Since the name of the window uses the visible text attribute of the base component as part of its name, any modification to this attribute also updates the name of the corresponding extended comment window view . An extended comment window view (with extended comment text added) is shown in figure 5.14.

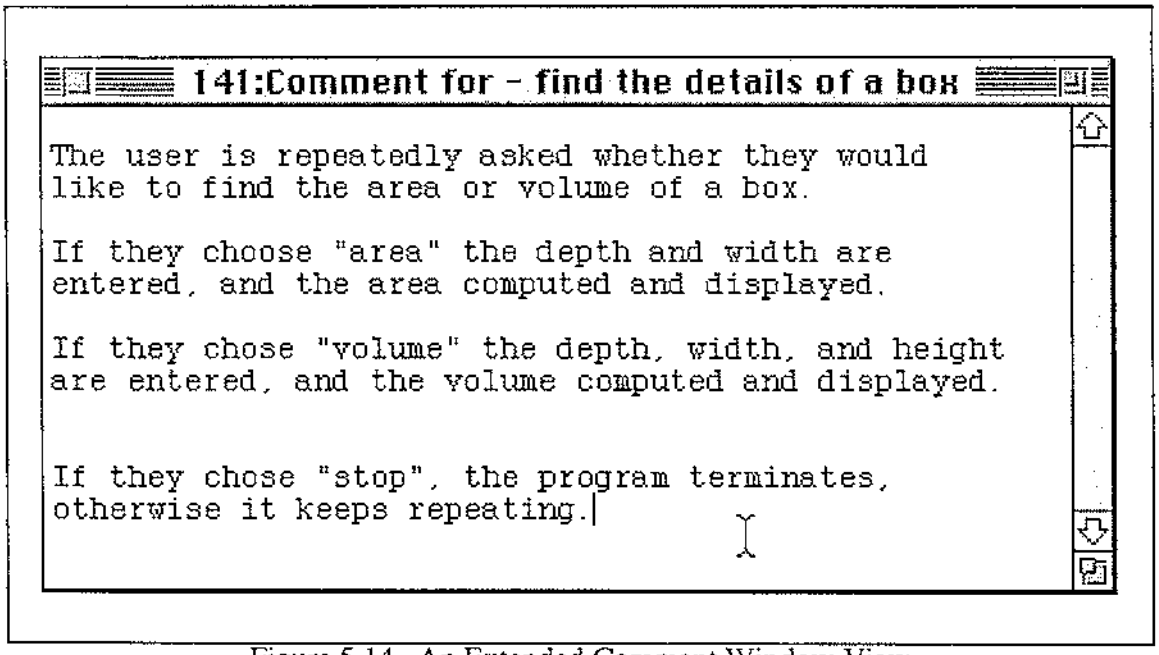


Figure 5.14. An Extended Comment Window View

The data manipulation actions of a HyperPascal program are represented in the action tree view using an action sequence icon. This icon contains a list of actions (corresponding to the list of actions in the base action sequence) that are displayed as coloured text fields. The list of actions in the action sequence icon can contain fields of two colours, white, and green. Green coloured text fields represent input and output actions, and white fields represent other actions. An action sequence icon is shown in figure 5.15.

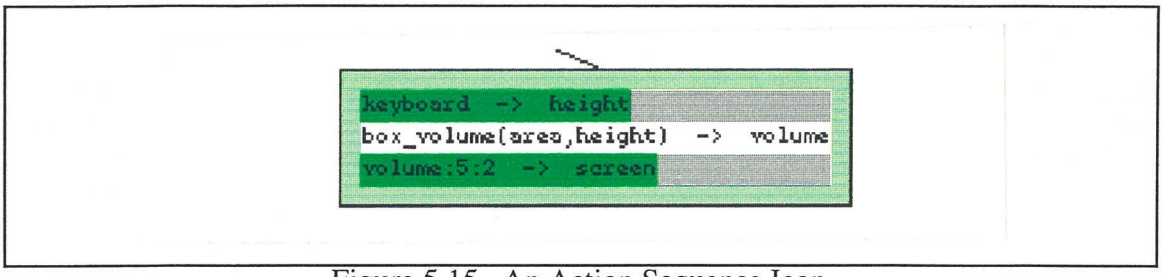


Figure 5.15. An Action Sequence Icon

When the selection tool is activated, and a double-click is performed on the list of actions in the icon, a dialog is activated for adding, deleting, and modifying the actions in the list. The list of actions in the base action sequence component is automatically updated when the actions in the action sequence icon are modified. If the double-click was inside the icon, but not on the representation of the list of actions, the associated forms window view for the base action sequence is activated (or a forms window view is created, then associated with the base action sequence).

The forms window view is a textual view that allows the free-editing of the formatting information associated with inputs and outputs. The forms window view for an action sequence is created containing entries (in the form of specially formatted text) for each input and output action, and the variables being input or output. Input and output format text can be inserted before, between, and after the input and output fields, as it is to appear at run-time. An action sequence with inputs and outputs, and its associated forms window (after formatting text has been inserted) is shown in figure 5.16.

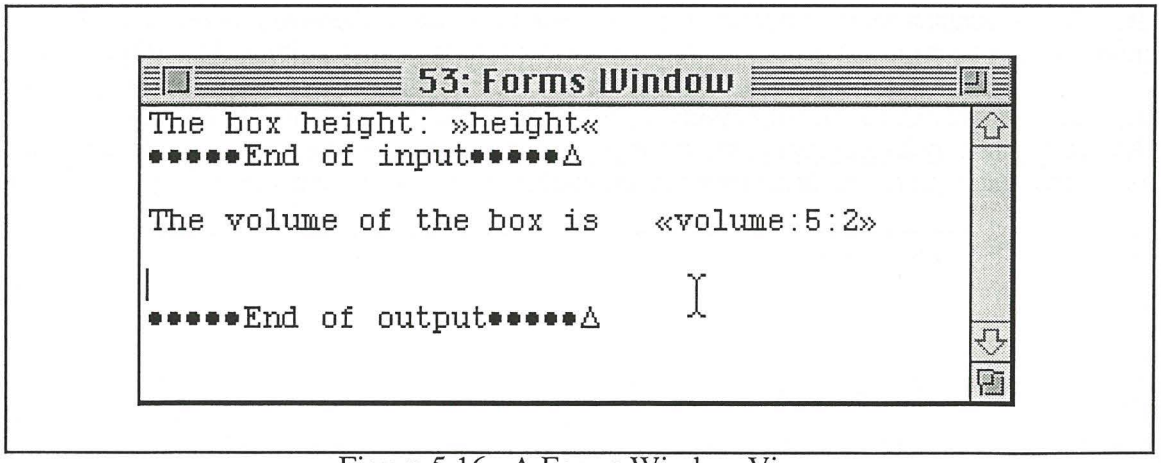


Figure 5.16. A Forms Window View

A modification of the input and output actions of an action sequence will cause an update of the associated forms window to also reflect this modification. The modification to the forms window is in the form of a simple substitution of the inputs and outputs in the action sequence with the inputs and outputs in the forms windows. The modification of the input and output fields in the forms window is not updated back to the base action sequence.

Base action sequences are terminal nodes (they have no child relationships), and since action sequence icons graphically represent base action sequences, they are also terminal nodes. The use of the connection tool to connect an action sequence icon as the parent in a relationship with another icon is, therefore, not allowed.

The choice icon is a graphical representation of the base choice component. It contains a short comment, can be associated with an extended comment window view, and also has

a one-to-many relationship with the graphical connectors to a number of other action tree icons.

The modification of a choice icon is very similar to the modification of a comment icon, with a double-click on the icon's text enabling the modification of the text, and a double-click on the area surrounding the text activating (or creating) an extended comment window view. When the connection tool is used with the choice icon as the parent, however, the graphical connectors created are conditional graphical connectors.

A conditional graphical connector is a graphical representation of a base condition connector, and has an attribute which contains the condition that is associated with that branch. When the conditional graphic connector is added in the action view (and the corresponding base conditional connector added to the base view), the value in the condition attribute is the default condition. The representation of the conditional graphic connector is a graphic connector, and the representation of the condition can either be the reduced representation of a condition disk above its connection point with the choice icon, or an expanded field showing the condition expression. If the condition attribute contains the default condition, the reduced representation disk is filled white, otherwise it is filled blue. The conditional graphic connector is manipulated, and the condition modified using the select tool.

When the select tool is selected, the order of the conditional graphic connectors in the choice icon can be modified by clicking on the end of the connector or the condition representation, and dragging to its new connection point within the same choice component. When the user double-clicks on the conditional graphic connector (either the graphic connector part or the condition part) the system checks to see whether the condition is currently shown in a reduced or expanded representation. If the condition is shown in a reduced representation, it is changed to the expanded representation. If the condition is shown in the expanded representation, the system checks to determine if the double-click occurred on the condition expressions text. If it was on the expressions text, a dialog is activated for the modification of the condition, otherwise the condition is changed to the reduced representation. A choice icon, with both a reduced and an expanded conditional graphic connecter attached, is shown in figure 5.17.

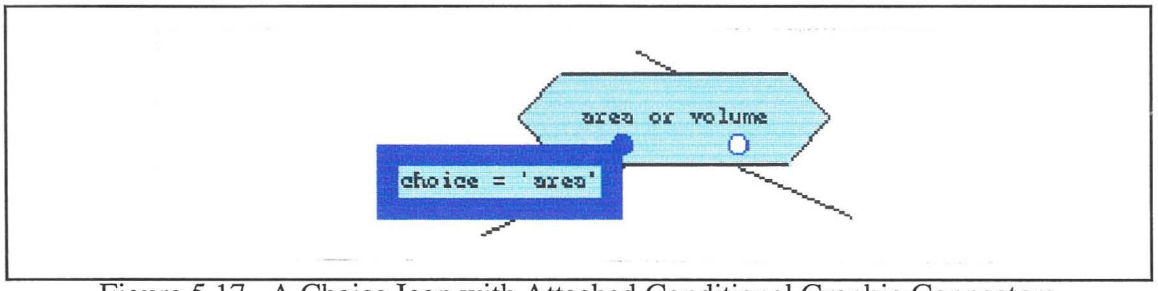


Figure 5.17. A Choice Icon with Attached Conditional Graphic Connectors

A loop icon is a graphical representation of a base loop component in the action tree. A loop icon has as attributes, a list of pre-loop actions, a list of post-loop actions, and a short comment. The loop icon can also have an extended comment associated with it. The short comment is shown inside body of the loop icon, and the pre-loop, and post-loop actions are shown on the top edge of the icon in either a reduced or expanded representation.

The loop icon is modified (and the base loop updated accordingly) by double-clicking on the icon when the select tool is selected. When the icon is double-clicked on, a check is performed to determine in what part of the icon the double-click happened. If the double-click happened in a reduced pre-loop or post-loop actions area, the area is expanded, and if it happened inside an expanded pre-loop or post-loop actions area, a check is performed to determine if the click occurred on the action list. If the double-click

occurred on an action list, a dialog is activated for modifying this list the same way as the action list of an action sequence icon, and the appropriate list of actions in the corresponding base loop is updated. If the double-click was in the expanded area, but not in the action list, the representation of the actions area is changed to the reduced version. If the double-click occurred on the short comment, a dialog is activated for editing the comment (and updated in the base loop), and if it occurred in the body of the loop icon, the extended comment window view for the loop is activated or created.

A loop icon has a one-to-many relationship with graphical connectors to the icons representing the actions within the execution of the loop. In addition to normal graphical connectors, this relationship can also include continue graphical connectors and terminate graphical connectors. These continue and terminate graphical connectors can only be attached to a condition sequence icon. Graphical connectors are added (as before) using the connection tool, and the continue and terminate graphical connectors are added in the same way, but holding down the command key for a continue graphical connector, and the shift key for a terminate (stop) graphical connector. A condition sequence icon can be added at the same time as a continue or terminate graphical connector by performing the connection mouse-up in empty space.

The condition sequence icon has the same appearance and editing characteristics as the action sequence icon, except it can also represent a condition (a blue text field). The continue graphical glue has the representation of a normal graphical connector (coloured green) with a green triangle-shaped icon at its connection point with the loop, pointing to the right (meaning continue through the loop). The terminate graphic glue is coloured red, and the icon at its connection point is a red triangle, facing upwards (meaning exit from the loop). The attachment positions of these graphic connectors to the loop icon determine where in the loop the associated condition will be evaluated, and they can be moved by dragging. A loop icon with associated terminate graphic glue and condition sequence is shown in figure 5.18.

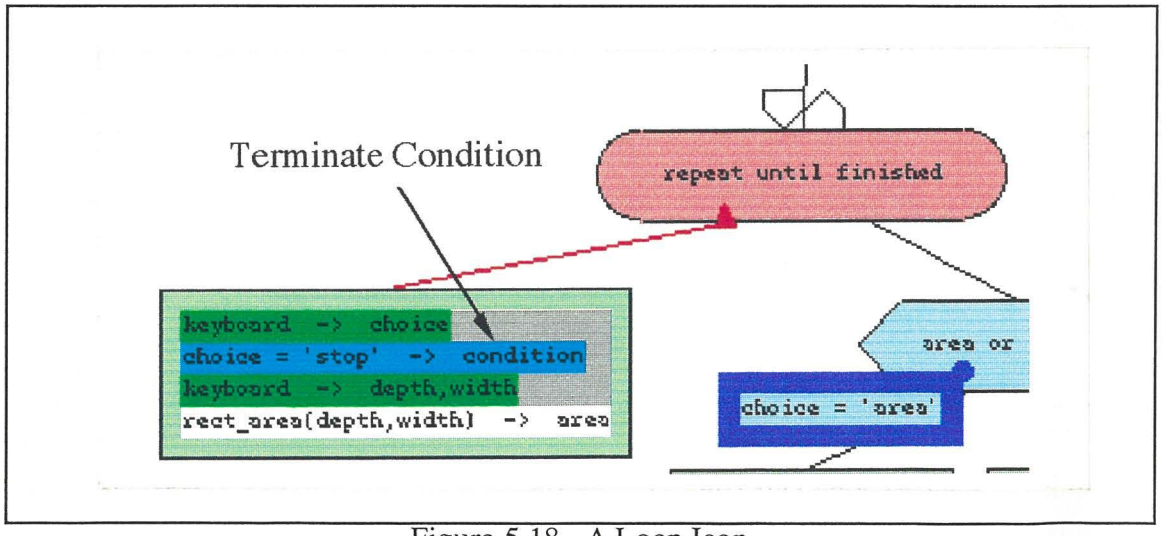


Figure 5.18. A Loop Icon

5.4. Experimentation with HyperPascal

A number of hyperprograms have been developed using the HyperPascal prototype, in order to test the generality of applicability of the language. These examples have also been used to judge how easy it is to edit, and navigate in a hyperprogram. The hyperprograms developed include (among others) a program for the calculation of prime numbers, a chequebook accounting program, and a program for the conversion of

numbers to a hamming-encoded bar code (these examples are included on the attached disk).

The chequebook accounting program allows the user to input withdrawal and deposit information. The program stores these transactions, allows the current balance to be calculated, and also allows the transactions to be reviewed. The hamming code conversion program prompts the user for a number, then produces a graphical hamming encoded bar code corresponding to that number. The prime numbers program calculates all the prime numbers up to a maximum value (set by the user), using the sieve of Eratosthenes. This program is shown in figure 5.19.

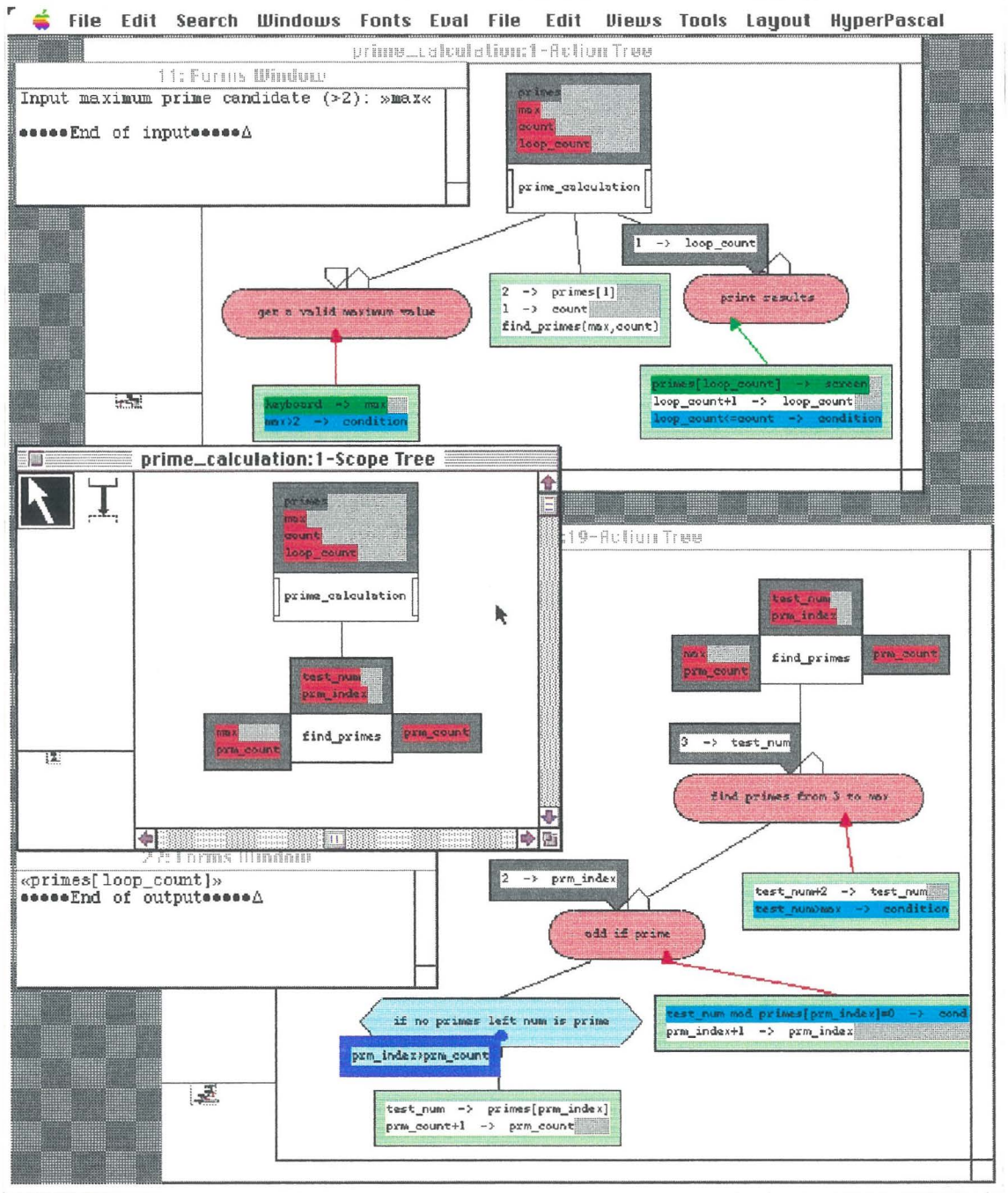


Figure 5.19. A Screen Dump of the Prime Calculation Program

The prime number calculation program has two program units, the prime_calculation program, and the find_primes subprogram. The prime_calculation program declares an array of integers to contain the primes calculated. The program works by prompting the

user to input the maximum prime value, uses the find_primes subprogram to find the prime numbers and load them into the primes array, and after all the primes have been found, prints them out.

The find_primes subprogram calculates the primes up to a maximum value using the sieve of Eratosthenes. To do this, it iterates through integer values in steps of two from three to the maximum number set by the user, adding the value to the primes array if it is a prime. To check if the number is a prime, it is divided by each prime found so far, and if none of them divide without remainder into the candidate number, the candidate number is a prime.

To run the program, it is first converted into Pascal source code, then compiled and run using the Think Pascal (Symantec, 1990) programming environment. The Pascal code generated from the HyperPascal program is shown in figure 5.20 (below).

```
{Unit#1}
program prime_calculation (input, output);

 var
  h_p_24: boolean;
  h_p_8: boolean;
  primes: array[1..200] of integer;
  max: integer;
  count: integer;
  loop_count: integer;

 procedure find_primes (max: integer; var prm_count: integer);
 forward;

{Unit#19}

 procedure find_primes;

  var
   h_p_50: boolean;
   h_p_46: boolean;
   test_num: integer;
   prm_index: integer;


 begin {find_primes}

{find primes from 3 to max}

  test_num := 3;
  h_p_46 := true;

  while (h_p_46) do
   begin
{add if prime}

    prm_index := 2;
    h_p_50 := true;

    while (h_p_50) do
     begin
{if no primes left num is prime}

      if (prm_index > prm_count) then
       begin
```

```
      primes[prm_index] := test_num;
      prm_count := prm_count + 1;
      end;
     h_p_50 := not (test_num mod primes[prm_index] = 0);
     if not (test_num mod primes[prm_index] = 0) then
      begin
      prm_index := prm_index + 1;
      end
    end;

   test_num := test_num + 2;
   h_p_46 := not (test_num > max);
   if not (test_num > max) then
    begin
    end
  end;

end; {find_primes}


begin {prime_calculation}

{get a valid maximum value}

h_p_8 := true;

while (h_p_8) do
 begin
  write('Input maximum prime candidate (>2): ');
  readln(max);
  writeln;
  writeln;
  h_p_8 := not (max > 2);
  if not (max > 2) then
   begin
   end
 end;

primes[1] := 2;
count := 1;
find_primes(max, count);
write('The primes are:');
writeln;
{print results}

loop_count := 1;
h_p_24 := true;

while (h_p_24) do
 begin
  write(primes[loop_count]);
  writeln;
  loop_count := loop_count + 1;
  h_p_24 := loop_count <= count;
  if loop_count <= count then
   begin
   end
 end;

end. {Unit#1}
```

Figure 5.20. The Pascal Source Code Generated

The Pascal source code generated by HyperPascal is intended as an intermediate step in the execution of a program. The source code produced is therefore not intended to be viewed for the purposes of comprehension or modification, so minimal formatting has been used.

This program allows the calculation of up to 200 prime numbers, but more can be accommodated by increasing the upper bound of the primes array. A screen dump of the output of the program is shown in figure 5.21.

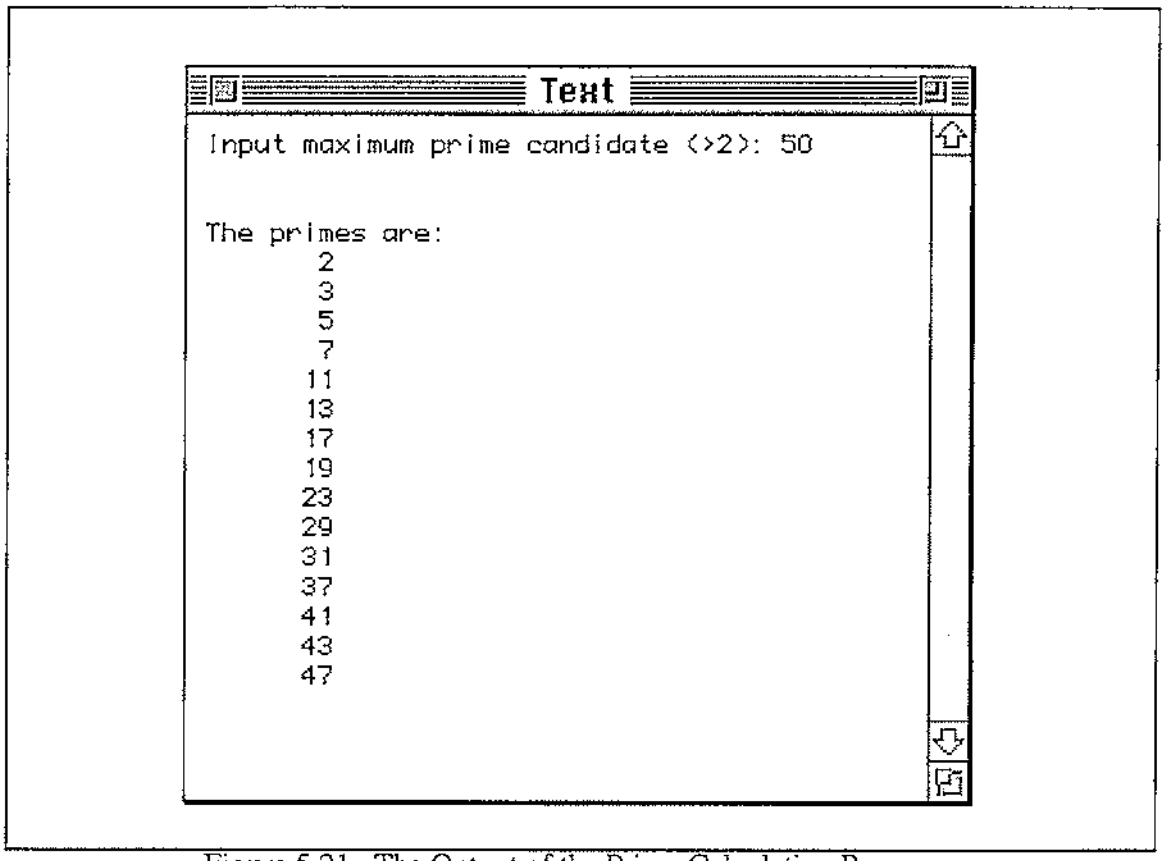


Figure 5.21. The Output of the Prime Calculation Program

Chapter 6

Conclusions

The representation of computer programs as text does not make use of the graphics and direct manipulation facilities that are common in modern computers. Research into visual programming languages has attempted to remedy this by representing programs graphically, and allowing them to be modified using direct manipulation. Visual programming languages have been found to appeal to novice users, but not experts. Most general-purpose visual programming languages do not appeal to experts because they are quite limited in scope of applicability, usually two-dimensional in nature, and when used for developing large programs, their representation can become large and hard to navigate within.

As a solution to these problems, this thesis suggests a visual programming approach which uses a number of appropriate views of a complete underlying program structure. The proposed approach utilises the hypermedia technique of representing multi-dimensional structures using nodes and links for representing the underlying structure of a program, and facilitating navigation within it. A program represented this way is termed a *hyperprogram*.

A visual programming language, HyperPascal, is proposed as a "testbed" for testing the completeness, consistency, and practicability of hyperprogramming. HyperPascal is designed for imperative programming, and is based on the semantics and data types of Pascal. Four views of the multi-dimensional underlying program are introduced, and three of them are described in detail. The representation and functionality of each of the programming components is described, and the interaction with the representations in each of the three views is also described.

The prototype implementation of a program development environment for a subset of the HyperPascal language is described. This software converts HyperPascal programs to Pascal source code for later compilation and execution. The method of converting the attributes of each HyperPascal component to equivalent Pascal source code is described, and the interaction with the different display representations is also described. A number of example programs have been written using the HyperPascal prototype, and one of these is shown together with the Pascal source code produced, and the output resulting from its execution.

6.1. The Conclusions

HyperPascal was developed to establish the feasibility of visual languages based upon hyperprogramming. The development proceeded in three phases. First, a "hyper-version"

of Pascal was designed. That is to say, a language with all the functionality of Pascal, but a representation based on multiple windows several types of view was designed. A tailored program design environment was included in this design.

Secondly, a subset of this language, embodying its most important features, was identified, and a program development environment for it was implemented. This subset concentrated on working with multiple views and on diagrammatic representation of the programs actions and, to a lesser extent, its declarations.

Thirdly, the environment was used to capture several programs and convert them to conventional Pascal, for compilation and execution.

Following this experience, what conclusions can we 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. The obvious way to verify this was to design a complete 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.

Classification of HyperPascal

HyperPascal is a visual programming language, and can therefore be classified according to Shu's (1986) classification framework. It is classified here on the same three-dimensional framework as the two general-purpose visual programming languages classified in chapter two - Pict/D (Glinert, 1984) and ProGraph (Cox, Giles, Peitrzykowski, 1989).

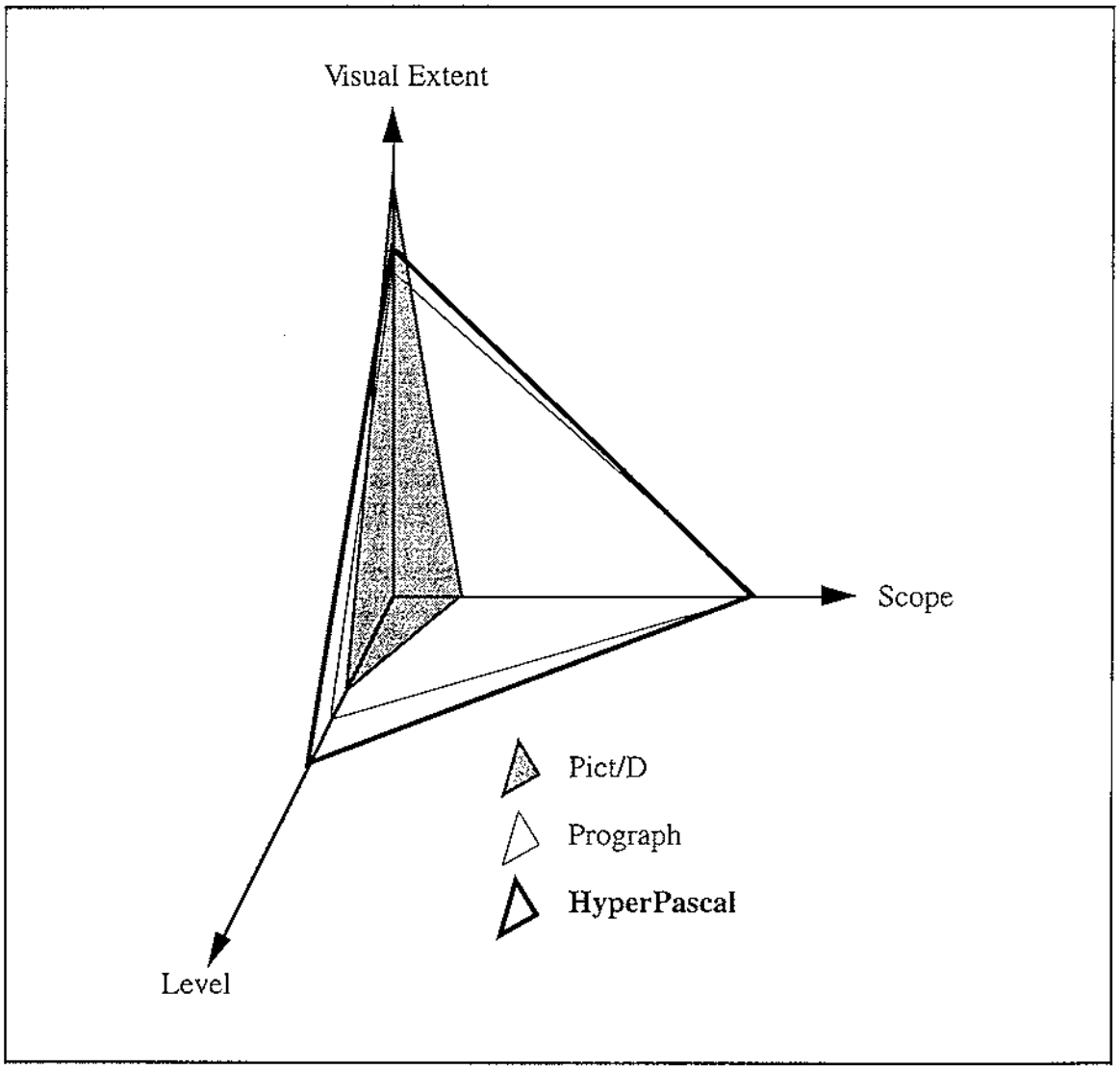


Figure 6.1. The Classification of HyperPascal Using Shu's (1986) Framework

The level of the HyperPascal is higher than that of ProGraph and Pict/D because, although primarily procedural in nature, HyperPascal utilises the forms window view to allow the non-procedural specification of input and output information.

HyperPascal has a similar scope to ProGraph, because it can be used to implement general purpose programs.

The visual extent of HyperPascal is between that of ProGraph and Pict/D. HyperPascal has a higher visual extent than ProGraph because it uses different graphical representations of complex types, and uses colour to either encode, or redundantly re-code information.

6.3. Suggested Future Work

This thesis presents the research conducted to date on the HyperPascal language, its prototype implementation, and the hyperprogramming approach in general. The progression of further research in these areas has many possible alternative routes, a number of which are described below.

The Complete Implementation of HyperPascal

The current implementation of HyperPascal is a prototype implementation of a subset of the HyperPascal language. A number of features of the language have been excluded from the implementation of this prototype in order to speed the implementation. These features include the ability to declare types and constants, the graphic representation of expressions, automatic type matching, and the representation of the calling area of the program unit. Also excluded are many interaction features, such as the capability to edit declarations and expressions "in-place", and the ability to declare variables, types, constants, and subprograms "on the fly". While the complete implementation of the HyperPascal language is outside the scope of this project, it would be worthwhile in order to further refine and test the proposed notation and interaction techniques.

Development of HyperPascal for Teaching

Teaching practices currently in place at Massey University encourage the design of programs using structure diagrams, prior to the implementation of the program. A common problem found with this approach is the difficulty of keeping the structure diagrams and actual program consistent with each other. The HyperPascal prototype could be modified to produce "tidier" Pascal source as output, therefore enabling the students to generate the program directly from the structure diagrams. Any errors then found in the program could be corrected by editing the structure diagram directly, thus keeping the two representations consistent.

A Run-time Environment

The HyperPascal prototype does not allow the execution of the hyperprogram directly, and the Pascal source code has to be compiled and executed using a separate Pascal compiler. Using this technique, it is difficult to locate run-time errors in the hyperprogram. The hyperprogram should be able to be developed, compiled and executed in a single, integrated environment. To enable the location of errors, and the dynamic visualisation of an algorithm, the hyperprogram (or part of a hyperprogram) should also be able to be executed in a step-by-step manner, highlighting the current position in the hyperprogram at each step. Allowing the user to view the contents of selected data structures during this step-by-step execution would enable the internal state

of the hyperprogram to be found at a particular stage, and would also enable the viewing of the data structures dynamically using animation.

A General Alternative to the Forms Window View

The format of the input and output of a HyperPascal program is currently specified using a text mode version of the forms window. The input and output information dealt with in modern applications is often not in the form of text, therefore, a more general approach to the specification of input and output formatting is needed. The specification of the formatting information associated with the input and output data may require the development of a visual description language for this view.

A Further Investigation of the State Tree View

This research has investigated the state tree view of the hyperprogram very little. The state tree view could be developed further to investigate its use both as a state indicator in the programming process (to show possible states at the current point) and at run-time (to show the possible, and actual states of variables at a particular point in the program). It would also be interesting to investigate the possibility of using the state tree view to specify algorithms and control structures, and to integrate it with the run-time environment.

The Structure Traversal Icon

Many programs require the traversal and manipulation of complex, pointer-based structures such as lists, and trees. The current design of HyperPascal traverses and manipulates these structures in a similar manner to traditional languages, but the feasibility of generalising structure traversals into a single programming component is currently being researched.

Alternative Views of a HyperProgram

HyperPascal assumes a structured programming, procedural view of a hyperprogram, and its views were chosen accordingly. There seems no reason why a hyperprogram could not be constructed using different programming paradigms (such as functional and object-oriented) and appropriate views for constructing hyperprograms using these paradigms will be different from those chosen for HyperPascal. It would be interesting to investigate the possible views, and combinations of views that could be used to construct a hyperprogram.

The process of mapping an abstract idea to a computer program has traditionally required much specialist knowledge, and has involved the complexity of mapping a multi-dimensional mental representation of a program to a linear medium. HyperPascal, and the hyperprogramming approach in general, use a multi-dimensional program representation to make the mapping of an idea to a computer program easier and more intuitive.

References

AMBLER, A.L., BURNETT, M.M. (1989): Influence of Visual Technology on the Evolution of Language Environments, *IEEE Computer*, October 1989, pp. 9-22.

APPERLEY, M. (1989): Taking the Hype out of Hypermedia, *New Zealand Journal of Computing*, **1**(2), pp. 19-25, 1989.

ASSMAN, K., VENEMA, R., HOHNE, K.H. (1986): The ISQL Language: A Software Tool for the Development of Pictorial Information Systems in Medicine, In *Visual Languages* (CHANG, S.-K., ICHIKAWA, T., and LIGOMENIDES, P.A., eds.), pp. 261-284, 1986, Plenum, New York.

BROOKS, F.P., Jr. (1987): No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, **20**(4), pp. 10-19, April 1987.

BROWN, G.P., CARLING, R.T., HEROT, C.F., KRAMLICH, D.A., SOUZA, P. (1985): Program Visualization: Graphical Support for Software Development, *IEEE Computer*, **18**(8), pp. 27-35.

BROWN, M.H. (1988): Exploring Alogrithms Using BALSA-II, *IEEE Computer*, **21**(5), pp. 14-36.

BROWN, M.H. (1991): Zeus: A System for Alogrithm Animation and Multi-View Editing, *1991 Symposium on Visual Languages*, 1991, pp. 4-9.

BRUSILOVSKY, P. (1993): Program Visualization As A Debugging Tool For Novices, *adjunct proceedings InterChi '93*, Addison-Wesley, April 1993.

BUDD, T.A. (1991): *An Introduction ot Object-Oriented Programming*, Addison-Wesley, 1991.

CHANG, S.-K. (1987): Visual Languages: A Tutorial and Survey, *IEEE Software*, **4**(1), pp. 29-39.

CONSENS, M., MENDELZON, A., RYMAN, A. (1992): Visualizing and Querying Software Structures, *Proc. 14th Intl. Conf. on Soft. Eng.*, ACM, pp. 138-156.

CUNNIFF, N., TAYLOR, R.P., BLACK, J.B. (1986): Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal, *Proc. CHI '86*, pp. 175-182, April 1986.

DAVIS, A.L., KELLER, R.M. (1982): Data Flow Program Graphs, *IEEE Computer*, **15**(2), pp. 26-41, Feburary 1982.

DUDLEY, T., MAHLING, D. (1991): Report on E-mail Panel: Is Visual Programming a New Programming Paradigm?, *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 82-88, IEEE, 1991.

EDEL, M. (1988): The Tinkertoy Graphical Programming Environment, *IEEE Trans. on Software Engineering*, **14**(8), pp. 1110-1115, August 1988.

EISENSTADT, M., BRAYSHAW, M. (1990): A Fine-Grained Account of Prolog Execution for Teaching and Debugging, *Instructional Science*, **19**(4), pp. 407-436.

FAIRLEY, R.E. (1985): *Software Engineering Concepts*, McGraw-Hill, 1985.

FINZER, W., GOULD, L. (1984): Programming by Rehearsal, *Byte*, **9**(6), pp. 187-210, 1984.

GAVER, W.W. (1989): The SonicFinder: An Interface That Uses Auditory Icons, *Human-Computer Interaction*, **4**(1), pp. 67-94, 1989.

GLINERT, E.P. (1989): Towards software metrics for visual programming, *International J. Man-Machine Studies*, **30**, pp. 425-455, 1989.

GLINERT, E.P. (1990): Out of Flatland: Towards 3-D Visual Programming, In *Visual Programming Environments: Applications and Issues*, (Glinert, E.P., ed.), IEEE, 1990.

GLINERT, E.P., BLATTNER, M.M., FRERKING, C.J. (1991): Visual Tools and Languages: Directions for the '90s, *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 89-95, IEEE, 1991.

GLINERT, E.P., TANIMOTO, S.L. (1984): PICT: An Interactive, Graphical Programming Environment, *IEEE Computer*, **17**(11), pp. 7-25, 1984.

GOODWIN, L., SANTANI, M. (1986): Learning computer programming through dynamic representation of computer functioning: Evaluation of a new learning package for Pascal, *International Journal of Man-Machine Studies*, **25**, pp. 327- 341, 1986.

GRAF, M. (1990): Visual Programming and Visual Languages: Lessons Learned in the Trenches, In *Visual Programming Environments: Applications and Issues* , (Glinert, E.P., ed.), IEEE, 1990.

GRUNDY, J.C. (1993): *Multiple textual and graphical views for Interactive Software Development Environments*, PhD Thesis, Department of Computer Science,University of Auckland, 1993.

GRUNDY, J.C., HOSKING, J.G. (1993): Integrated Object-oriented software development in SPE, in *Proc. 13 th New Zealand Computer Society Conference*, pp. 465-478, 1993.

HILS, D.D. (1992): Visual Languages and Computing Survey: Data Flow Visual Programming Languages, *Journal of Visual Languages and Computing*, **3**(1), pp. 69-101, March 1992.

HIRIKAWA, M., MONDEN, N., YOSHIMOTO, I., TANAKA, M., ICHIKAWA, T. (1986): HI-VISUAL: A Language Supporting Visual Interaction in Programming, In *Visual Languages* (CHANG, S.-K., ICHIKAWA, T., and LIGOMENIDES, P.A., eds.), pp. 233-259, 1986, Plenum, New York.

HIRIKAWA, M., NISHIMURA, Y., KADO, M., ICHIKAWA, T. (1991): Interpretation of Icon Overlapping in Iconic Programming, *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 254-259, IEEE, 1991.

HIRIKAWA, M., TANAKA, M., ICHIKAWA, T. (1990): An Iconic Programming System, HI-VISUAL, *IEEE Trans. on Software Engineering*, **16**(10), pp. 1178-1184, October 1990.

HIRIKAWA, M., YOSHIMI, M., TANAKA, M., ICHIKAWA, T. (1989): A Generic Model for Constructing Visual Programming Systems, *Proc. 1989 IEEE Workshop on Visual Languages*, pp. 124-129, IEEE, 1989.

ICHIKAWA, T., HIRIKAWA, M. (1990): Iconic Programming: Where to Go?, *IEEE Software*, 7(11), pp. 63-68, November 1990.

KODOSKY, J., MACCRISKEN, J., RYMAR, G. (1991): Visual Programming Using Structured Data Flow, *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 34-39, IEEE, 1991.

KOPACHE, M.E., GLINERT, E.P. (1988): C^2: A Mixed Textual/Graphical Programming Environment for C, *Proc. Workshop on Visual Languages*, pp. 231-238, IEEE, 1988.

KORFHAGE, R.R., KORFHAGE, M.A. (1986): Criteria for Iconic Languages, In *Visual Languages* (CHANG, S.-K., ICHIKAWA, T., and LIGOMENIDES, P.A., eds.), pp. 11-34, 1986, Plenum, New York.

KRAMER, J., MAGEE, J., NG, K. (1989): Graphical Configuration Programming, *IEEE Computer*, **22**(10), October 1989.

LAU-KEE, D., BILLYARD, A., RAICHNEY, R., KOZATO, Y., OTTO, P., SMITH, M., WILKINSON, I. (1991): VPL: An Active, Declarative Visual Programming System, *Proc. 1991 IEEE Workshop on Visual Languages*, pp. 40-46, IEEE, 1991.

LPA (1992): *LPA Prolog Version 4.5 Reference Manual*, Logic Programming Associates Ltd, 1992.

LYONS, P., SIMMONS, C., APPERLY, M. (1993): HyperPascal: A Visual Language to Model Idea Space, in *Proc. 13 th New Zealand Computer Society Conference*, pp. 492-508, 1993.

MAYER, R.E. (1988): From Novice to Expert, In *Handbook of Human-Computer Interaction* (HELANDER, M., ed.), pp. 569-580, North-Holland, 1988.

McDAID, J. (1991): Breaking Frames: Hyper-Mass Media, In *Hypertext/Hypermedia Handbook* (BERK, E., DEVLIN, J., eds.), pp. 445-458, 1991, MacGraw-Hill Publishing Company Inc, New York.

MORETTI, G.S., LYONS, P.J. (1986): An Overview of GED, A Language-Independent Syntax-Directed Editor, *Australian Computer Journal*, **18**(2), pp. 61-66, May 1986.

MYERS, B.A. (1983): INCENSE: A System for Displaying Data Structures, *ACM Computer Graphics*, **17**(3), pp. 115-125.

MYERS, B.A. (1986): Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, *Proc. CHI '86: Human Factors in Computing Systems*, pp. 59-66, ACM, 1986.

MYERS, B.A., CHANDHOK, R., SEREEN, A. (1988): Automatic Data Visualisation for Novice Pascal Programmers, *Proc. 1988 IEEE Workshop on Visual Languages*, IEEE, October 1988, pp. 192-198.

NGAN, P.M. (1991): OpShop: an iconic programming system for image processing, *Proc. Australasian Apple University Consortium Conf.*, ANU.

RAEDER, G. (1985): A Survey of Current Graphical Programming Techniques, *IEEE Computer*, **18**(8), pp. 11-25, 1985.

REISS, S.P. (1985): PECAN: Program Development Systems That Support Multiple Views, *IEEE Trans. on Software Engineering*, **11**(3), pp. 276-285.

REYNOLDS, C.F. (1987): The Use of Colour in Language Syntax Analysis, *Software-Practice and Experience*, **17**(8), pp. 513-519, 1987.

ROHR, G. (1986): Using Visual Concepts, In *Visual Languages* (CHANG, S.-K., ICHIKAWA, T., and LIGOMENIDES, P.A., eds.), pp. 233-259, 1986, Plenum, New York.

ROUSSOPOULOS, N., LEIFKER, D. (1984): An Introduction to PSQL: A Pictorial Structured Query Language, *Proceedings of the 1984 IEEE Computer Society Workshop on Visual Languages*, pp. 77-87, December 1984.

RUBIN, R.V., GOLIN, E.J., REISS, S.P. (1985): ThinkPad: A Graphical System for Programming by Demonstration, *IEEE Software*, **2**(2), pp. 73-79, March 1985.

SCANLAN, D.A. (1989): Structured Flowcharts Outperform Pseudocode: An Experimental Comparison, *IEEE Software*, **6**(5), pp. 28-36, September 1989.

SHNEIDERMAN, B. (1988): We can design better user interfaces: A review of human-computer interaction styles, *Ergonomics*, **31**(5), pp. 699-710, 1988.

SHNEIDERMAN, B., KEARSLEY, G (1989): *Hypertext Hands On!*, Addison-Wesley, 1989.

SHU, N.C. (1986): Visual Programming Languages - A Perspective and a Dimensional Analysis, In *Visual Languages* (CHANG, S.-K., ICHIKAWA, T., and LIGOMENIDES, P.A., eds.), pp. 11-34, 1986, Plenum, New York.

SHU, N.C. (1988): *Visual Programming*, Van Nostrand Reinhold, New York, 1988.

SYMANTEC (1988): *Macintosh Pascal Reference Guide*, Symantec Corporation, 1988.

SYMANTEC (1990): *THINK Pascal User Manual*, Symantec Corporation, 1990.

SYMANTEC (1991a): *THINK C User Manual*, Symantec Corporation, 1991.

SYMANTEC (1991b): *THINK C Object-Oriented Programming Manual*, Symantec Corporation, 1991.

TERRY, P.D. (1986): *Programming Language Translation*, Addison-Wesley, 1986.

TRAVIS, D. (1991): *Effective Color Displays*, Academic Press Ltd, London, 1991.

WIEDENBECK,S. (1985): Novice/expert differences in programming skills, *International Journal of Man-Machine Studies*, **23**, pp. 383-390, 1985.

WILLIAMS, C.S., RASURE, J.R. (1990): A Visual Language for Image Processing, In *Proc. 1990 IEEE Workshop on Visual Languages*, IEEE, 1990.

WOOD, W.T., WOOD, S.K. (1990): Icons in Everyday Life, In *Visual Programming Environments: Applications and Issues* , (Glinert, E.P., ed.), IEEE, 1990.

BAILEY, D.G., HODGSON, R.M. (1988): VIPS - A Digital Image Processing Algorithm Development Environment, *Image and Vision Computing*, 6(3), August 1988.

Appendix A

Starting HyperPascal

Copy the software from the enclosed disk to the hard disk, and use ResEdit to copy the resource file `HP.rsrc` to the MacPROLOG application. Set the application size of MacProlog to at least 4 Megabytes, and the evaluation space to approx. 800k.

To start HyperPascal, follow these steps:

- Double-click on `Snart 1.1` to start MacPROLOG with the Snart environment loaded.
- Open `MViews.pl` from inside MacPROLOG
- Open `HyperPascal` from inside MacPROLOG
  The classes will compile automatically.
- To start HyperPascal, enter the Prolog Query `init_hp`