

# An Introduction to Clarity: A Schematic Functional Language for Managing the Design of Complex Systems.

by

T. R. Addis<sup>1</sup> and J. J. Townsend Addis

University of Portsmouth, School of Computer Science and Mathematics, Mercantile House, Portsmouth  
PO1 2EG, UK.

## **Abstract**

*Clarity is a functional schematic programming language currently freely available to the community<sup>2</sup>. It is a programming environment that allows a user to draw a program as a set of directed graphs. The term schematic is drawn from the traditions of engineering where the diagrams that represent electronic circuits or those of physical objects are often referred to as schematic drawings. A schema is a set of pictures or graphs that represent a program or working model. A schematic is taken as a system of tokens and structuring rules that expresses a program, model or concept; it is a graphical language. This paper introduces the principles behind design and issues to be considered when dealing with complex systems. The reasons why a 'functional' representation provides a non-invasive approach to design and forms the basis of 'good' design are described. In particular, the advantages of using diagrams is shown to be because the schema constructions make the structure of complex systems explicit as well as make a functional representation more intelligible than its sentential equivalent.*

## **Introduction**

There has been some very strong criticism of those attempting to produce a general-purpose visual programming language (Green 1990, Petre & Green 1993, Blackwell 1996). In particular, Citrin (Citrin 1996) identified three distinct areas that make such languages unacceptable. These are:

- *Appropriateness of mapping*, deciding which aspects of problems map naturally into visual representations and which are best left as text.
- *Scalability*, the ability to construct and comprehend very large visual programs.
- *Transparency of the interface*, allowing changes and modifications of visual programs to be as easy as editing text.

---

<sup>1</sup> Visiting Research Fellow at the Science Studies Centre, Department of Psychology, University of Bath.

<sup>2</sup> <http://www.sis.port.ac.uk/research/clarity/index.html>

Experimental findings of Green and Petre (Green 1990, Petre & Green 1993) seemed to show that overall graphics was:

- Significantly slower than text to interpret.
- Slower than text in all conditions.
- Intrinsically difficult.

In particular, it was observed that 'Knotty Structures' (those involving many decision points) overload the short-term memory of those trying to understand such diagrams.

This paper is the second of a pair of papers that is intended to respond to these issues. The first paper (Addis & Addis, Expected publication date Feb 2002) addresses all these criticisms and shows they only apply to certain kinds of programming languages. It was shown that the real advantage of a graphical notation is that it makes a functional language easy to use; a language form that has excellent design properties. Here we describe the schematic functional language formally (see Appendix 1 & 2) and illustrate the characteristics that give it value as a tool for managing complex designs.

There seem to be many advantages in being able to draw a model directly into a computer system. One major advantage would be that the diagram, provided it is of the right kind, offers the designer or engineer a representation that seems more comprehensible than a linguistic description. It is not that the linguistic description necessarily lacks anything but a simple diagram for certain problems seem to be much easier to grasp. We identified semantic coherence<sup>3</sup> as being one of the necessary properties of a representation that ensures this comprehension (Addis & Addis 2002). In addition to semantic coherence are the requirements to keep the diagrams simple and yet still able to define completely a complex system.

One approach to simplicity and comprehensibility is through the separation of different design manifestations of a system. For example, the distinction between the engineering of a car as part of a working transport system, an ergonomic device and a work of art. Complex designs such as that required for computer 'chips' (wafer fabrication) in practice demand several stages (about 11 in this case).

Another approach to simplicity and comprehensibility in design is to consider a system as a set of interfaces. A familiar example can be found in the designs of a housing estate. The overall plan of the estate may simply be a road system with coloured squares where each square is a house and each colour represents a particular kind of house. The purpose of this layer is to show how houses are distributed over the land and how they relate to each other. At the next level, we have the house plans showing the different layouts of the rooms. This layer is aimed at the potential house owner and is used to assess the potential of the house. Finally, there are the detailed builders plans showing how the different services (gas, water, electricity and sewage) are connected. The representation at each layer may be considered a distinct and

---

<sup>3</sup> There are two kinds of semantics: the formal semantics that relate symbols to the machine processes that the language controls and the informal semantics that relate the symbols to the world to be represented. See later.

unique pictorial language consisting of a set of services and functions. Each pictorial language is designed for a particular job at that level.

The exact nature of a set of representations for new domains cannot always be predefined. The style and structure of a representation to serve a particular purpose may only emerge as the scheme develops through trials and use. Thus one of the major stages in design is a process of identifying these layers (and this requires insight), reorganising the internal structure of the system to reflect the layers and complete each layer with a full set of operations. This flexibility in design will often lead to greater simplicity and generality of a system.

Such layering is only required to reduce the complexity of the system for the designer (or maintainer). For working models, it may bring with it a cost of inefficiency. Other ways of reducing the perceived complexity may be needed that do not affect the system design. This is where a representation rich in informal description potential is necessary. Diagrams, for example, are particularly rich in extra dimensions of representations that are not all used in formal specifications. Examples are the spatial layout, the size and colour of the set of tokens. All these dimensions can be used to provide informal meaning to the formal structure. These approaches can be used together in order to establish some kind of overall grasp of a complex system; a grasp that ensures the quick development of a system with the potential of zero errors.

### **Models, Purpose and Complex Systems**

Suppose you wanted to explain the rules of the road to a visitor in your country or explain to your solicitor what happened in a car accident. Then you may well deploy a stone, matchbox or salt cellar to represent a car and other objects in the road. What is important is that you can show the spatial relationships between cars, objects and the road as well as recreate the sequence of events to be described. In the case of an accident, we will be able to assess who might have been at fault. We use the model to make inferences about the world.

On the other hand, you may want to describe to a child the different visible parts of a car. In this case, you may find a simple LEGO™ brick car sufficient for the task. Therefore, you will be able to show where the bonnet, the wheels, the seats and the boot are and with little LEGO™ 'people' you can show where they sit and who are passengers and who is the driver. In this case, we use it for training.

However, none of these models would be any good if you wanted to describe how a car is motivated and how a differential works. We might like to demonstrate the effect of a gear getting jammed or a coupling falling off. For this, we would have to buy Technical LEGO™ or use the construction tool kit Meccano™. Each model is an *abstraction* of the real thing and each model has some properties that are ignored. For example, how many cars do you know that have little pegs distributed in lines all over its surface?

Computer models serve a similar role as these toys in that they are abstractions of part of the world. The question arises is why is it difficult to create (to engineer) most computer programs to do what seems to be an easy task? Certainly, some tasks are quite easy to do but not many. Part of the reason is whether the representation used has been designed to handle the task. The representation in the easy case is said to be designed for a problem domain (tasks of a particular type). At an extreme level, a single program call will be all that is needed because someone has already 'solved' your particular problem. Unfortunately, the problem domain for this representation is confined to just that single task.

Computer languages and development environments can be likened to building kits such as LEGO™. The LEGO™ bricks, for example, have been designed to be put together in a very large number of ways and if you wish to build models of square houses, there is no problem. If you wish to model a car, you can, but you need a bit of interpretation. However, you may wish to build a model boat; fine, provided you don't expect it to float.

LEGO™ has been created for children and therefore has been designed to make the building of (say) model houses easy. The cost of this reduction in complexity is a representation that is highly constrained and for many purposes is useless. If you wish to build a model of a 'working' car then you need a different representation such as Meccano™. So we can see that each representation has a specified range of tasks to which it is aimed and, in general, the wider the range the more difficult it is to create a model in that medium.

Computer models are confined to models of a particular type. So once the medium of the computer has been decided for modelling then the range of possible tasks, although infinite, is constrained. Nevertheless, like the model car in LEGO™, a leap in imagination has to happen to determine those elements of the task you can represent and those you can't. One of the principal reasons why computer models become complex is because this leap falls short of either what can be done or what you would like to be done. The link between the problem domain and the implementation has to be bridged.

The question then arises as whether it is possible to design a programming language and an associated environment that can be:

- as close to the problem domain as one likes,
- as fluid to potential change as is appropriate,
- supportive of good design,
- linked formally with a computational engine?

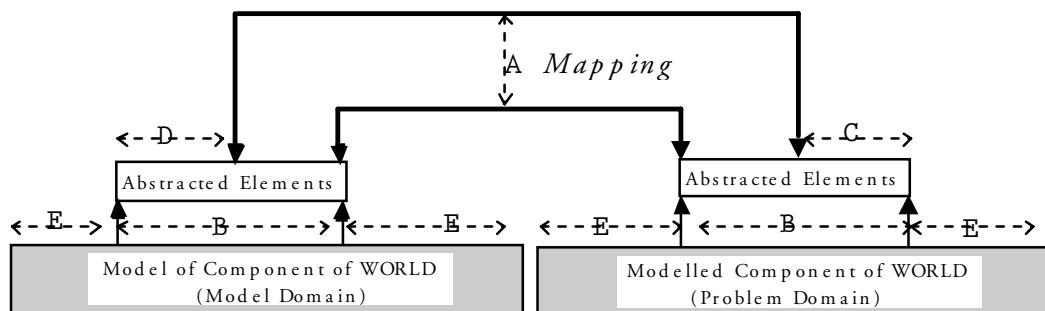
To have all this and still cope easily with the complexity of modelling would be the major objectives of any programming language and its environment.

## The Symmetry of Informal and Formal Semantics.

It is clear that central to computer modelling is the ability to abstract from the world and to link this abstraction to some construction in a programming language. Problem solving in the world depends primarily on the ability to abstract successfully (Jaques 1978) and to relate these abstractions to coherent strategies; strategies, in our case, that involve modelling. The models provide the focus for thought. It is the thought that then leads to the prediction of useful insights or actions.

We will propose here that it is the model which encompasses a particular interpretation of something in the world (what it models) and hence is viewed as a unique abstraction of itself (the elements involved in the model). In the simplest case, the relationship between the model and the modelled is a mapping between two abstractions of the world (see figure 1).

The relationship between the world and itself during the act of modelling depends upon an observer (e.g. the interpreter, the designer, and the user) who is capable of making *consistent* abstractions. The different aspects of this act are illustrated in Figure 1.



**Figure 1. The different aspects of Models and the Modelled**

The boxes in figure 1 represent sets of distinctions (features) and the letters between the solid arrows indicate a subset of those distinctions. The solid arrows indicate the boundaries of all the connections (a mapping function) that relates a set of distinctions to another set of distinctions. So:

- A. is the range of abstract elements where there is a one to one correspondence between the two abstracted distinctions of the world. It is where the common semantics, both formal and informal, exist for both world and model. The common semantics are related and are accessed by the observer for his/her purpose(s). In our toy example, it is the set of LEGO™ bricks locked together as a unit that represents (say) the engine compartment of the car.
- B. is that part of the world that can be abstracted and recognised by the observer to be of possible significance for his/her purpose(s). It is the full semantics of the problem or model domain for which the observer has access. In our example, it is the engine compartment, boot, differential and

clutch in the problem domain; it is the different bricks, pegs, colour and shapes in the model domain.

- C. is the extended abstraction of the world to be modelled that is not involved in this model domain. If the model is formal, (i.e. the semantics are formal) this is referred to as the *informal semantics* of the problem domain. Components of this part may also be modelled with another model domain. C in our car example contains the differential or gears.
- D. is the extended abstraction of the model that is ignored for modelling. (D+A) represent the full semantics (B) of the model for which the observer has access. If it is a formal model then this will be the full formal semantics. D in our example includes the pegs and colour.
- E. is the extended part of the world that is not abstracted and plays no part in the semantics of the observer; it has no meaning within the framework of the observer's purpose. In the case of the modelling domain, it is the internal flanges of the bricks or the smell of the plastic. In the case of the problem domain, it could be the colour of the garage or the temperature of the sea.

Note that the areas A, B, C and D are achieved by the observer (user, programmer, designer etc.) and not by any external mechanism. These areas help define the observer's relationship with the model, the world and the mapping of the world into the model.

There is symmetry in the above arrangement in that what is the model or the modelled can be swapped. For example, the idea of modelling the relationship between a University and the student body as the same as a shop and its customers is a current example of the use of this symmetry. In practice, the direction is from a non-standard world to a well-understood domain that offers some advantage. We understand the simpler relationship between a shop and its customers better than the more subtle relationship between a University and its students.

In particular, the advantage of mathematics as a modelling domain is that it offers a rich source of inference mechanisms. These mechanisms have been developed over centuries and have the additional advantage that engines exist that embody many of them. The computer is, of course, one of the most useful of these engines.

Modelling with a computer means that we have to learn strategies and techniques that allow us to map a problem into a program. From examples, we learn different solutions that we can adapt for our purposes. These 'programming games' are hard won skills that come through experience (see Green and Petre 1996). It is not surprising that professional programmers are reluctant to move to new modelling methods that do not draw upon their current expertise.

### **Diagrams and the functional Language**

We have shown previously (Addis & Addis, 2002) that a visual representation for creating a computational model (a program) based upon a functional language

interpreter will reduce significantly the most frequent errors of the kind that normally plague any programmer. We started from noting that many introductions to a functional language find it necessary to use a diagrammatic representation of the language in order to describe how it should be understood (e.g. Field and Harrison 1988, Reade 1989). The form of the diagrams tend to be similar in each introductory example simply because the function lends itself naturally to this kind of representation. There is a perfect semantic coherence between the two representations. We then extended a version of these diagrams as a pictorial equivalence of a functional language to form the programming schematic language Clarity.

However, does such an equivalent representation help? Our own work has shown that the value of a functional language is significantly increased if presented in a diagrammatic form. We have found that the functional language is more easily understood in its diagram form. We have also shown that *the diagram* when used over a range of standard tests *reduces the frequency of errors for a functional language by about a factor of four* (Addis, Booth and Townsend, 1992).

In a paper by Jill Larkin and Herbert Simon (Larkin & Simon 1995) where they compare sentential and diagrammatic representations they conclude:

“The advantages {of diagrams}, in our view, are computational. That is diagrams can be better representations not because they contain more information, but because the indexing of this information can support extremely useful and efficient computational processes.”

Their paper showed, amongst other things, that the diagrams provided, at essentially zero cost, all the inferences that have a dimensional organisation. In our case, the functional language is essentially non-dimensional; it may not be able to take advantage of such dimensional inference directly but at least it is not restricted to such a bias. Never the less, we can use the dimensional organisation of our diagrams to make informal statements about the structure of the model and because of the low inferential cost we can support complex structures with little effort.

It has been argued elsewhere (Addis & Addis, 1998) that a complicated structure can be managed through at least three possible techniques. These are:

- *The introduction of conceptual levels.* Another approach to retaining simplicity in model design is to consider the structure as a set of interfaces or abstractions.
- *The segmentation into functional domains (aspects).* An approach to retaining simplicity is through a perspective that falls naturally in line with the separation of different manifestations of a model. For example, the distinctions between the engineering of a car as a working transport system, an ergonomic device and a work of art.
- *The introduction of informal semantics.* Diagrams, for example, are particularly rich in extra dimensions of representations that are not all used in formal specifications. Examples are the spatial layout, the size and colour of the set of tokens. All these dimensions can be used to provide informal meaning to the formal structure (see above).

These techniques can be used together in order to establish some overall grasp of a complex model. The diagrammatic functional interpreter, Clarity, developed by the authors has incorporated all these principles as a basis for its design.

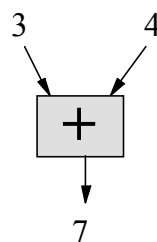
### An Introduction to Functional Programming

Those readers who are familiar with a strongly typed functional language such as ML or Miranda may skip the next section. Others may wish to extend their understanding by reading Field and Harrison (1988) or Reade (1989). However, the Faith language that underlies the work in this paper is a *functional database language*. This was originally described by Poulouvasilis, and King (1990). The version we are using is defined in Appendix 1.

Most people are familiar with a mathematical function. The simple notion of adding two numbers together such as 3 and 4 can be represented as:

$$3 + 4 = 7$$

Another way of looking at this is as a process  $\oplus$  that takes two numbers and translates these numbers into another. This can be shown as a black box with inputs and outputs (figure 2).



**Figure 2. The diagrammatic representation of a function**

where the order of the parameters of  $\oplus$  is defined clockwise from the output. We could have written this in normal functional notation as:

$$\oplus (3\ 4) \rightarrow 7$$

where  $\oplus$  forms a *function* followed by two *parameters* (3, 4) and this results in 7. The advantage of this form is that all functions follow the same structure of:

**function** (parameter 1, parameter 2, parameter 3. ....parameter n)

Since the function is always first we can move the first bracket so that the brackets encompass the complete expression. Thus in the following example of the function 'between' we can present it as:

$$(\text{between } 4\ 7\ 15) \rightarrow \text{True}$$

where this asks the question is 7 between 4 and 15 (in value) and in this case the result is True. This form has the advantage that we can now nest the functions thus:

$$(\text{between } 4 \text{ (+ } 3 \text{ 4) } 15) \rightarrow \text{True}$$

where the calculation proceeds from the inner bracketed expression outwards. Thus the function '+' is carried out before the function 'between'. We could draw this expression as shown in figure 3.

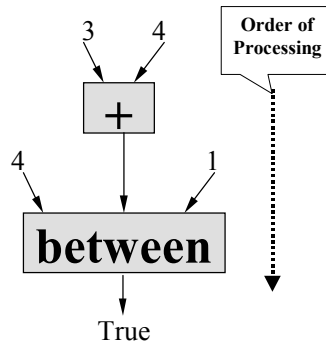


Figure 3. The combination of two functions

Note that this schematic representation changes the description of the processing of imbedded functions to top down rather than inside out. The top down description follows the same superficial form as an imperative language although formally it remains functional.

We could of course replace any of the integers with others of the same kind (we could not use 'words' or 'Boolean values' for example). A function is said to **map** from a domain consisting of all possible combinations of the input parameter values (i.e. cross product of the domains) to values of the output parameter (called the co-domain of the function). So if we were to define the function + for example we would have to specify what sort of input parameters it has and what kind of output it produces. This might be done as follows:

```
fdec
+ ::= integer X integer → integer ;
;
```

where **X** represents the cross product. However, we can make an add 3 function by leaving the last parameter blank so it may be used as follows:

$$(+ \ 3) \ (4) \ \rightarrow \ 7$$

or

$$(+ \ 3) \ (5) \ \rightarrow \ 8$$

These functions (that are kind of incomplete functions) are called **curried** functions (after the mathematician H. B Curry). We can thus write the above declaration as:

```
fdec  
  + ::= integer → integer → integer ;  
;
```

So far, there is no mechanism of carrying out the addition. If we did not have a computer to do this adding the best we could do is simply list all the possible combinations thus:

```
fdef  
  + 1 1 ::= 2 ;  
  + 1 2 ::= 3 ;  
  + 2 5 ::= 7 ;  
  + 3 4 ::= 7 ;  
;
```

and so on. Then all we do is look up the pattern of parameters in the table. Each pattern is referred to as a *component* of the function '+'. The set of components that are specifically defined with constants (as in this case) is called the *extension* of the function. If a pattern cannot be found then the function is returned thus:

$$(+ 3 5) \rightarrow (+ 3 5)$$

So if a function cannot be evaluated it is simply returned unevaluated (it is not an error). There is a way round this particular problem of defining addition by using the idea behind the curried function (+1):

$$(+ 1) \text{ number} \rightarrow \text{next number}$$

We would have to define a mechanism for generating the next number in a series of infinite numbers (see appendix). Luckily the computer system already has this function defined and it is one of the built in library functions of Clarity. In this case we have to use the integers provided by the computer and we indicate these type of integers by preceding them with the symbol # (an example of a basic constructor with a single parameter).

$$(+ \#3 \#4) \rightarrow \#7$$

The calculation is carried out by the computer and the result returned. This calculation is called a **side effect** because it is triggered by the calling of the function + (in this case) and the calculation is done 'on the side'. Side effects happen in the world and are beyond the range of defining events within the functional language.

However, if we wanted to define our own numbers we can do so by introducing our own type where each integer is a **constructor** with no parameters. These constructors define an '**enumerated**' type.

```
cdec  
    0 ::= digit ;  
    1 ::= digit ;  
    2 ::= digit ;  
    3 ::= digit ;  
    .  
    .  
    .  
;
```

We would then have to define the way in which numbers are concatenated to form new numbers in an infinite series followed by the arithmetic functions. The appendix shows how this is done using just pattern matching and recursion. However, we have a perfectly good set of mathematical mechanisms associated with the computer. What this shows is that the functional language is capable of defining from first principles any calculus using just these two principles.

### Overview of a functional representation

Figure 4 shows that the functional language consists of only the function. The function has the job of mapping from one (and possibly compound) domain to another. This mapping has a clear formal semantics described in terms of functions. In this way, the constraints to be modelled are expressed only by the mappings. However, a function may have side effects that influence the world directly (see figure 1) and sometimes these side effects are used to bypass the normal mappings. This use of side effects can be a source of opacity in program design because the communication between functions is hidden. The 'job' of a constructor is to provide an interface to the world that requires interpretation by a user. This job is part of the informal semantics as shown in figure 1. It also provides a means of 'packaging' information to be passed onto functions.

- *Side Effects* - This is any event or state in the world triggered by calling a function or constructor. Examples are the value of a computer word, the printing of a character on the screen or the launching of a rocket.
- *Basic Constructors* - A constructor is a word or symbol and is a means of relating states of the world to the modelling domain. A constructor on its own will need to be interpreted by the user and a basic constructor has a side effect within the computer hardware. The computer has a range of value types that relate to its functionality. The value types are basic to the computer mechanism and are side effects. These different value types are marked by special predefined constructors (e.g. # for integer, #r for real) that are part of the functional language. The basic

computer types are bits, characters, and words. Basic constructors have a formal and informal semantics.

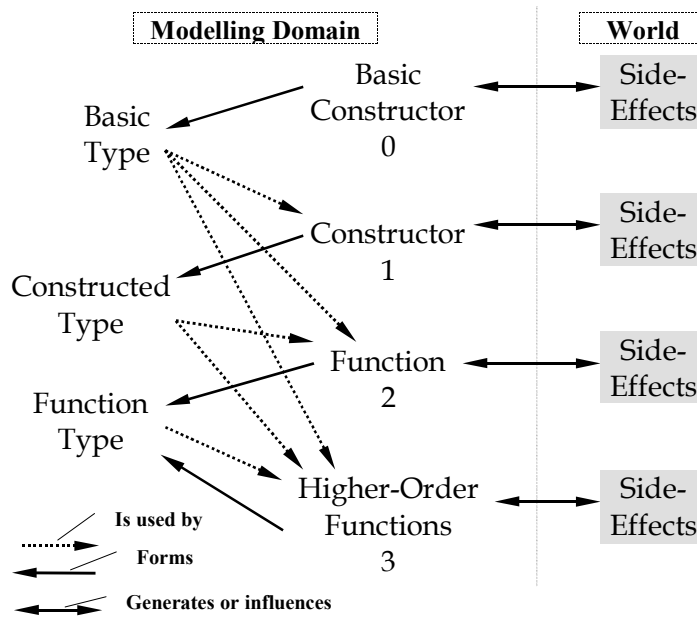


Figure 4. The principle elements of Faith; a functional language




- *Basic Types* - The computer types are mapped into the functional language types Boolean, character, integer, real and string. **Bits** are the fundamental unit of computing and are either on or off (current), high or low (voltage), 1 or 0, *True* or *False* depending on how the two extreme states are interpreted. These bits form into standard length **bytes** (usually 8 bits to the byte). Bytes are normally coded into *characters* (e.g. ASCII). In their turn, bytes are grouped into **words** (usually four bytes per word) and words are usually related to *integers*. It is on the word boundary that most of the hardware operations are designed to be most effective. A further coding usually uses two words to represent **floating point** or *real* number. Part of the pair defines where the decimal point is to be placed and the other determines the number. Finally, Because text is one of the major objects manipulated by computers, there are *strings* of characters and these can be of any length.
- *Constructors* - These provide a mechanism for defining ones own types. They can be considered to be like functions that cannot be interpreted and some people see them just as data structures. Their main task is to provide the informal links to the modelling process.
- *Functions* - the function is the mechanism for mapping a set (including the null set) of domains into a single domain. Its a transformation.
- *Higher-Order Functions* - a mapping where one of the domains is a set of functions of a specified declaration.

### Formal Diagrams and Clarity

It is the practice of most engineers, systems analysts etc. to use sketches and diagrams as aids to implementation. However, accounts about the design process assume that all diagrammatic modeling reduces to a linguistic representation of some kind.

Formal representation is essential to computation; nevertheless, formal notation may prematurely displace informal diagrammatic working during the process of developing a program or model (Shaw & Woodward, 1990). It is conceivable both to be formal and informal at the same time because it is now possible to generate program code directly from diagrams. These diagrams are the Clarity schema and can be interpreted directly into the functional language Faith, a strongly typed functional database language (Poulovassilis, 1988). We refer to such a representation as a schematic. Schema diagrams of this kind are designed to generate computer interpretable code. If the code is a functional database language then this generated code enables the coherence and consistency of the diagrammatic representations to be checked through type checking.

The graphs of the schematic Clarity are constructed from a small set of tokens (e.g. oblongs and lozenges) linked together with arrows. The schema components (graphs) in this case are constructed in windows. A formal definition of Clarity and its relationship with Faith is given in Appendix 2. The 'function window' is one of three windows that uses the tokens oblong and lozenge each in their three basic forms. Each of the forms of a token has a specific meaning as shown in the following table:

Tokens	Dotted Lines	Plain Lines	Bold Lines
<b>Oblong</b> 	Construct	Function	Recursive Function
<b>Lozenge</b> 	Type	Type Held (HOLDN) <sup>4</sup> .	Parameter, Constant or Faith expression.
<b>Arrow</b> 	Uses	Value	Not defined

**Table 1. The range of tokens for Clarity**

The simple principle behind a function declaration and definition is that every function needed for a definition can be represented by an oblong with the function's name inside; the name is put in by the programmer. There are nearly 100 library functions in Clarity Light<sup>5</sup>.

There are three different kinds of schema: *constructor*, *function* and *network*. In general, a constructor schema introduces new types and constructors, a function schema declares and defines a function and a network schema displays the dependencies between functions (i.e. a 'called by' graph). The constructor function and network schema are illustrated and detailed in the next sections. As an example, we will show how X Y co-ordinates may be represented in Clarity and how the beginnings of vector algebra might be created.

<sup>4</sup> Equivalent to the function *let*

<sup>5</sup> Clarity Light is available free from <http://www.sis.port.ac.uk/research/clarify/index.html>

## Data Types

There are seven basic data types defined for modelling and these types are distinguished from user defined data types in that their predefined constructors are integrated as part of the value (see Appendix 1). Thus the integer constructor # abuts to a number thus: #3462 or #-7144 for positive and negative numbers. Boolean values are just the enumerated types True and False; the values are constructors. Strings and characters have the common syntactic form found in most languages. List uses the constructor ':' (colon). The normal way in which a user may write a list of characters (say) is within square brackets (e.g. [ 'a' 'b' 'c' ]).

DOMAIN	TYPE	CONSTRUCTOR	SYNTAX	EXAMPLES
Integer	int	#'	#n where n is a number	#4 #125 #-36
Real	real	#r'	#rn where n is a number	#r4.3 #r46.32 #r-14.73
Boolean	bool	True, False	True, False	True, False
String	str		" "	"All good men"
Character	char		' '	'A' 'a' 'x'
List	list	:'	(: (: .. (: nil).. ) or [ ] where .. means repeat pattern for each item in list.	(: 'A' (: 'B' .. (: 'c' nil) .. )), ['A' 'B' 'C' 'c'] [#3 #45] Note that items in a list must be all the same type.
Variable	Var	?	?n where n is a number	?0, ?1, ?2 Note that the arguments of a function can be referred to by a variable. The first argument is always ?0. Sometimes a referential confusion can occur between variables used within an argument pattern and an argument. This can be avoided by using a ?n for a variable such that n is > number of arguments.

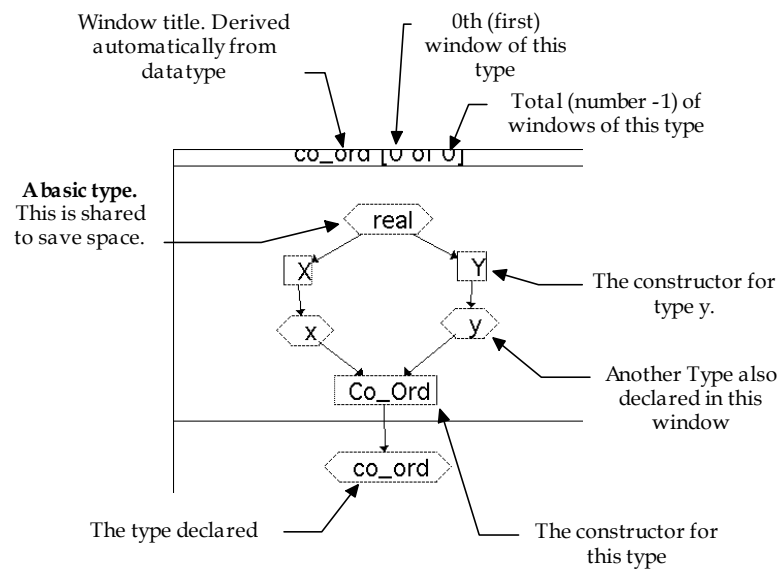
**Table 2. The main Basic data types of Faith & Clarity**

A variable on the other hand fulfils many roles. In the first instance it can be used to indicate within the body of a function definition a parameter position (e.g. the first parameter would be referenced as ?0 and the second as ?1 etc.). Alternatively, a variable can be used as a generic type (e.g. list ?3) showing where different but unspecified types should match. It is also used to indicate a general element in a pattern within the input parameters of a function (e.g. (: ?2 ?3) where ?2 is the head of a list and ?3 is the tail)<sup>6</sup>.

## Constructing Types

To declare a new type to the 'program' a constructor window must be opened. The new type is usually placed in a single output (dotted) lozenge in the bottom of the constructor window. An empty output lozenge will appear as soon as the first (empty) constructor box is placed in the body of the window. New types can then be declared in the body of the window. The boxes and lozenges are joined to create a structured 'package' that is used both for passing information from function to function and for self documentation purposes.

<sup>6</sup> Also ?0 can refer to literally to ?0.



**Figure 5. The declaration of a types `co_ord`, `x` and `y`.**

The same data type may have more than one constructor associated with it (for example the type 'date' may have both UK and USA form). To avoid an overcrowded diagram each variant of a type can be placed in a different constructor window. Each new window is referred to as a 'component' of that type.

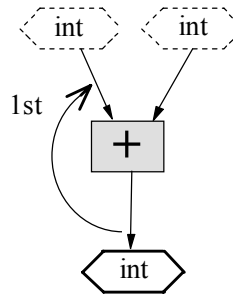
### Creating Functions

The function window is a palette on which a function can be declared and defined. The generic input and output parameters declare the function and the body (the middle part of the window) of the function is its definition. All functions have to be declared and it is possible to declare a function without commitment to its generic form.

A generic form of a function is a component of a function (if it exists). This component has all its input parameters specified as a type (e.g. as `int` or `str` etc.) and there are no actual values such as `#3` or "Fred". The output parameter can only be specified as a type in all cases.

The inputs of a function are the connecting arrows from either the output of a function or a parameter. The output of a function is a single connecting arrow directed outwards to another function, constructor or output parameter (see Appendix 2).

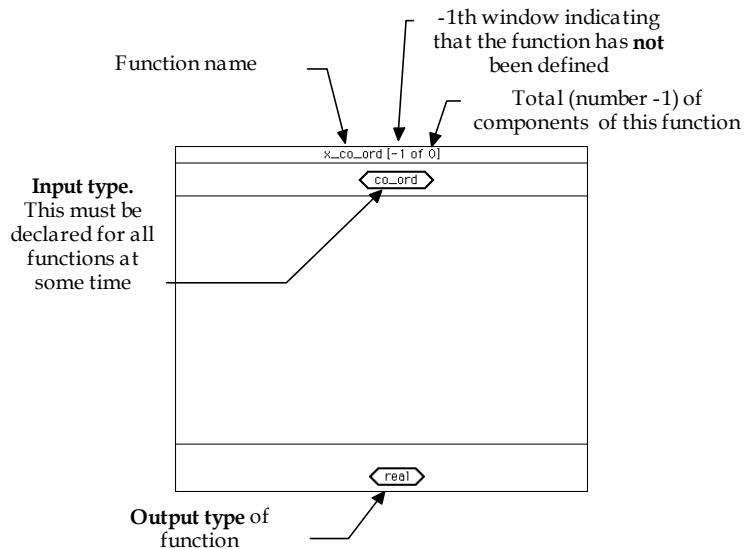
- The order of the parameters of a function or constructor is strictly ordered from the single output arrow (as reference start) in a clockwise direction.



**Figure 6. The order of the parameters is important**

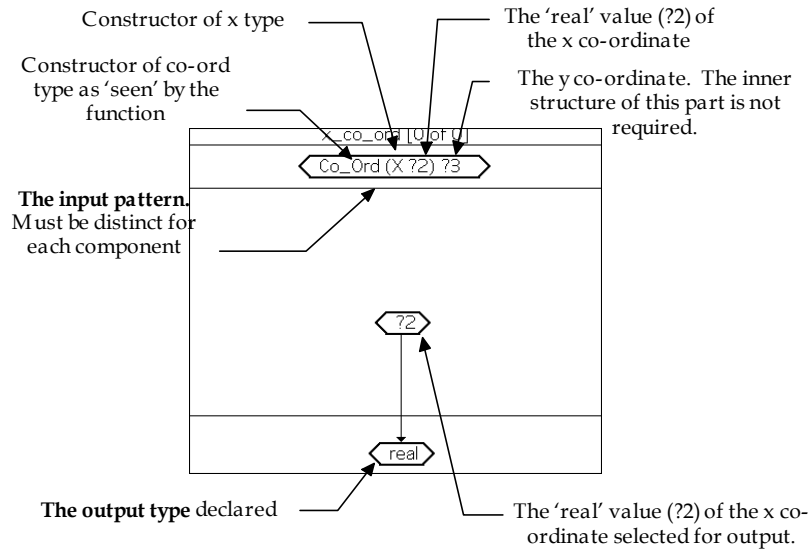
Although a function may only have a single output, it can provide many inputs via a lozenge. Once a function has been declared and defined, it may then be used actively in the declaration or definition of other functions. In practice, it is possible to use functions before they have been declared or defined but only as placeholders.

Other tokens or aids can be used to make a function or type declaration and definition clear. These do not provide any further functionality but do provide a means of assigning informal semantics to the schema (the graphs that represent a program). These include labelled join nodes and colour.



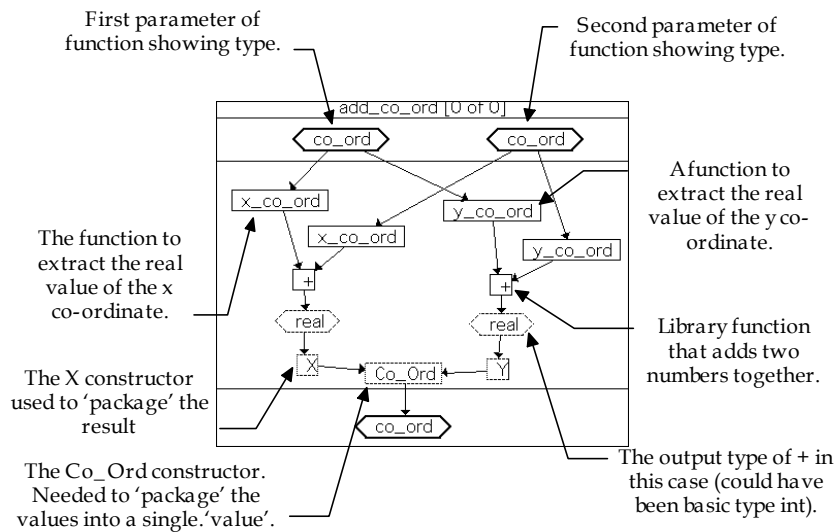
**Figure 7. An initial declaration of a function**

When a function is declared without a body (a definition) then it adopts the component number -1. The reason for making an isolated declaration of this kind is to establish the input and output types. This is usually done when the intended finally declared function will rely only upon specific 'patterns' of input or the intended function is to be used currently without a definition. It is good practice to always define a general case.



**Figure 8. A function for extracting the real value of an x co-ordinate.**

In the function `x_co_ord` (figure 8) we only use it for the specific pattern shown. Here we identify the real value of x that is 'wrapped-up' in the constructor formation and we pluck it out as the returned value of the function.



**Figure 9. The declaration & definition of a function.**

Figure 9 is an example of the simultaneous declaration (saying what the input and output types are) and a definition (what to do with the input and what to return as a value). In this case, the function adds two co-ordinates together. Note that we have to 'construct' the output so that it is packaged into a single (but complex) value.

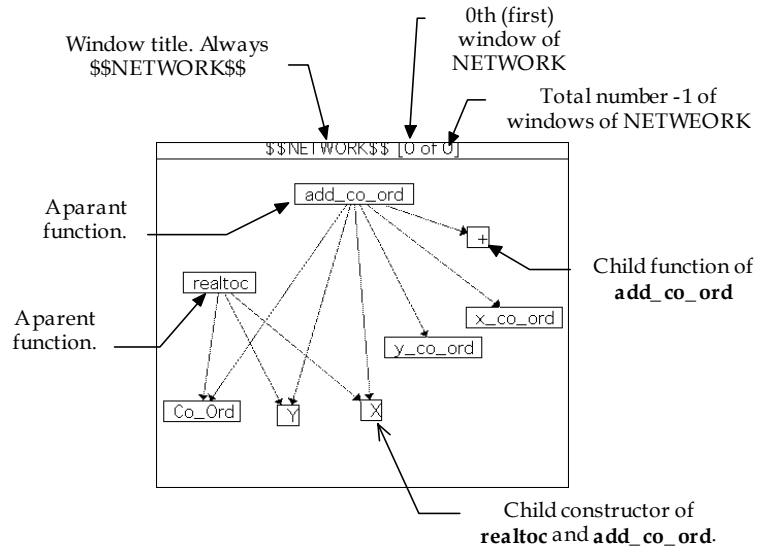
```

CONTROL [Co_Ord]
QUERY> add_co_ord ( Co_Ord ( X *r3.700000) ( Y *r5.620000))( Co_Ord ( X *r4.250000) ( Y
*r9.600000))
( Co_Ord ( X *r7.950000) ( Y *r15.220000))
QUERY>|

```

**Figure 10. The 'querying' of a function and the result.**

Once the function is defined and declared it can be tested by 'querying' it with some example input in the control window. Below (figure 11) shows the a network diagram during the development of the vector algebra 'program' (database). Here there is reference to a function 'realtoc'. This function takes two real numbers and constructs a co-ordinate (see figure 12).



**Figure 11. A Network window showing function dependencies.**

Faith, the underlying interpreted language, is a *strongly typed functional database language* (Poulovassilis, 1988, Poulovassilis and King 1990). The form of Faith is similar to Miranda or ML. The descriptive value of Faith depends upon the principle of 'best match' for a function component to be activated rather than the order it was defined (see Addis & Addis 2002). This 'best match' allows interactive development of a program because the components of functions can be defined at any time and in any order. Faith functions are normally stored on disc as a database. The functions are retrieved and run in main memory. Currently, all of the database is loaded into main memory thus avoiding retrieval problems at the cost of larger program size.

Clarity generates Faith code from the schema and conversely the schema can be generated from the Faith code (see in Clarity: File -> Build Segments ... ). Programs can be written as a mixture of both. It is the Faith code that is interpreted into computer instructions; it is Clarity that does most of the type checking and programmer support.

```

Faith [1 of 1]
tdec
co_ord ::= typeop #0 ;
x ::= typeop #0 ;
y ::= typeop #0 ;
;
cdec
Co_Ord ::= x ->y ->co_ord ;
X ::= real->x ;
Y ::= real->y ;
;
fdec
x_co_ord ::= co_ord ->real ;
y_co_ord ::= co_ord ->real ;
add_co_ord ::= co_ord ->co_ord ->co_ord ;
realtoc ::= real->real->co_ord ;
;
fdec
;
fdef
x_co_ord (Co_Ord (X ?2) ?3) ::=
  ?2 ;

y_co_ord (Co_Ord ?2 (Y ?3)) ::=
  ?3 ;

add_co_ord ?0 ?1 ::=
  Co_Ord (X (+ (x_co_ord ?0) (x_co_ord ?1))) (Y
    (+ (y_co_ord ?0) (y_co_ord ?1))) ;

realtoc ?0 ?1 ::=
  Co_Ord (X ?0) (Y ?1) ;

```

**Figure 12. Faith code generated from the Clarity windows.**

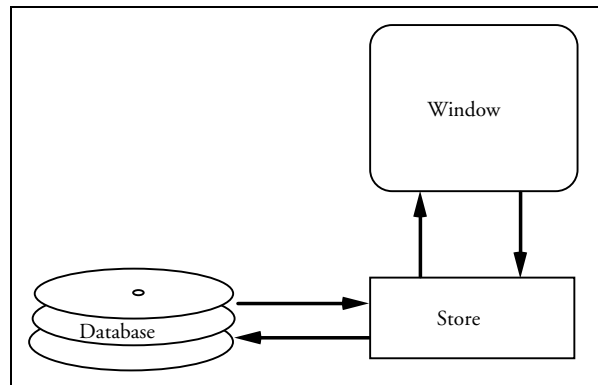
The main purpose of the design of the Clarity environment was to provide a supportive environment for a programmer that matches an interactive approach to modelling or programming. This might mean that some code is best programmed in terms of code rather than diagrams. A Faith program can be typed directly into the Faith window (see Appendix 1), committed, modified and run without resorting to Clarity. The Faith window will also display individual functions and individual functions can be declared and defined so that a program can be constructed from functional units. From these functional units, it is possible to generate a Clarity schema directly from the Faith code (not in Clarity Light). However, the results usually need some manual rearrangement.

### Three Levels of Memory

The Clarity environment consists of **three levels of memory**. To understand what is happening depends upon keeping these three levels in mind. The program or model is normally stored in **main memory** but before this can be done, the program or model has to be created or loaded into main memory from a database of such programs or models.

The initial stage of creation is to construct a model in one or more of the different types of schema windows: Faith, Constructor, Function and/or Network. The information placed in these **windows** is temporary and not linked with the model in any way at this stage. However, certain consistency checks are done by the Clarity interpreter. The 'running' model is kept in main memory ('Store' in figure 13). To include in the model (in store) a construct that has been created in one of the windows is the act of 'commit'. This action will translate the Faith Code, or schema into the store by integrating into the current model. However, even this is not permanent and to

ensure that the model is not lost it has to be 'saved' to the database. Once this is done, the model is secure.

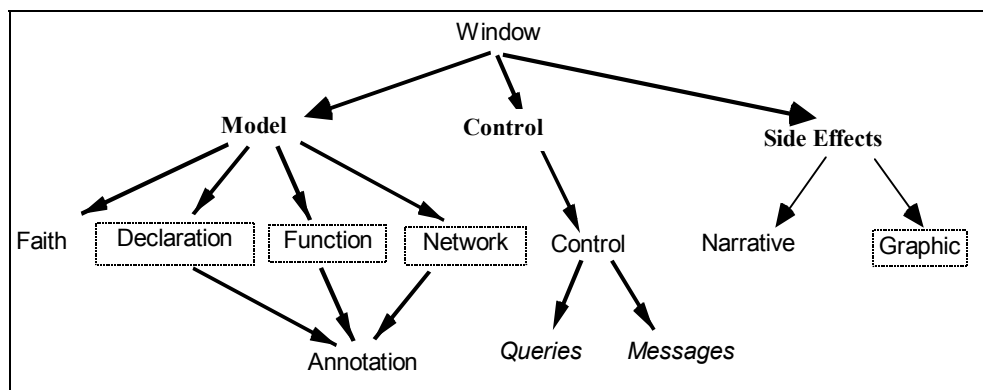


**Figure 13. The Three Types of Memory**

The Network window is unusual in that it is principally concerned with describing the state of the model as expressed on the **database** only and not in the main memory<sup>7</sup>. The Faith window describes the state of the database in the first instance and after every 'save'. It also maintains a record of the state of the functions committed through the schema windows (Function and Constructor) in between one 'save' and the next.

For the creation of a program or model there are three classes of window that can be identified:

1. *Model* ... Includes all those windows that provide mechanisms for creating a model.
2. *Control* ... The window that informs about the design environment and provides input to the model.
3. *Side Effects* ... The windows that provide output of the model.



**Figure 14. Windows and their role<sup>8</sup>**

Figure 14 shows the complete set of windows separated into their roles. The dotted boxes indicate those windows that are graphical and the others are text. The program or model is normally formed in the declaration and function windows. An

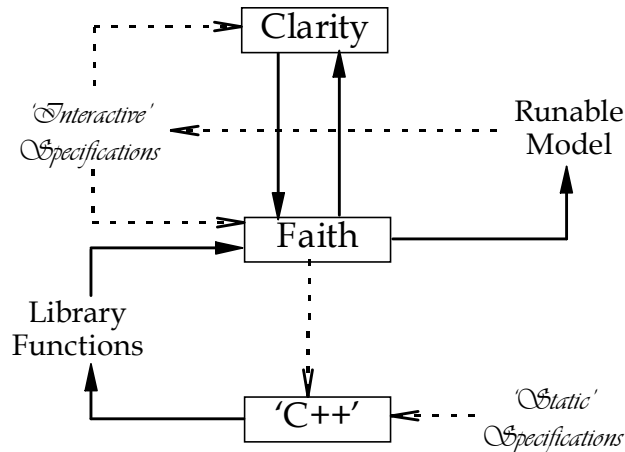
<sup>7</sup> However, it can be used in reverse to place function frames into main memory.

<sup>8</sup> Side Effects windows not available in Clarity Light.

annotation window contains text that is tied to a specific window and the tokens in that window. This provides a means of keeping detailed comments, descriptions and explanations on the model.

### User Libraries

Clarity generates Faith code, which is interpreted by a C program. Most of the library functions (Built In (BI) functions) have been written in C and are fast compared with the interpretive code. Having developed a program in Clarity it can be advantageous to make it a library function.



**Figure 15. The major Components of the Design Process**

Figure 15 shows the stages in which a working model of a complex system can be tried and tested as a Faith program. However, in most cases it will be appropriate to encode the system in an imperative language such as C or C++. This can be done by the Clarity/Faith specification being translated into C or C++ code and incorporated into the Faith library (the utility - F2C does this). In this way, the C or C++ code can be tested within the system model and the model itself slowly changes into the final system (whence all the system may be written in C or C++).

### Loops and Recursion

So far we have shown how a simple function that adds two vectors together may be defined and used. However, programming depends upon much more involved processes that require, in imperative terms, loops, iteration and jumps. The functional language relies upon recursion and higher order functions to support these kinds of operations. Jumps (gotos) where there is a change of control from one part of the program to another are definitely not allowed<sup>9</sup>. Jumps can cause problems in tracing the history of events and makes debugging difficult.

<sup>9</sup> At the time of writing this paper we are implementing the functions 'catch' and 'throw' which provide non-standard exits from functions.

One of the many mechanisms for providing a loop is recursion. Recursion can be considered as a means of re-entering a function from the beginning but with different parameters. Just as with (say) a 'for' loop there needs to be some kind of stopping condition. The simplest example of recursion is the calculation of 'factorial'. If we were to do this imperatively, we might write:

```
define factorial (N) ;
begin
  X := 1 ;
  If N = 1 then (return N) else
    for i := 1 step 1 until N do
      X := X * i ;
  return X ;
end ;
```

In the functional schematic, the approach is slightly different. The assumption is that factorial is assumed to work before it is defined so that the output of factorial can be relied upon. It is observed that:

$$\text{factorial (N) = N * factorial (N - 1)}$$

and

$$\text{factorial (0) = 1;}$$

and since this is a true relationship that holds for all N we can define it this way. The Clarity equivalent of this is shown in figure 16 where the general and specific cases are shown. This sort of runs the loop backwards (from N) until the terminating condition of factorial (0).

The query:

```
QUERY> factorial #5
#120
```

tests the definition which will work for all positive integers or until the computer's upper limit of integers is reached.

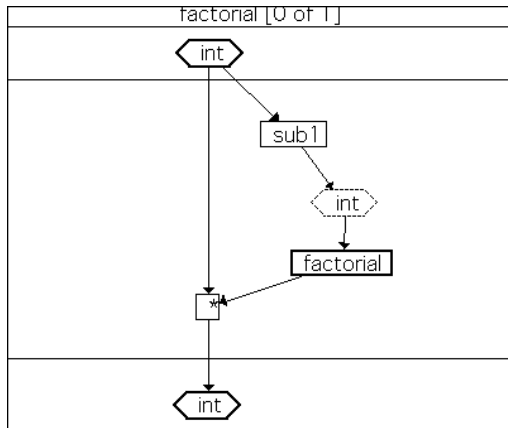


Figure 16a. The general case.

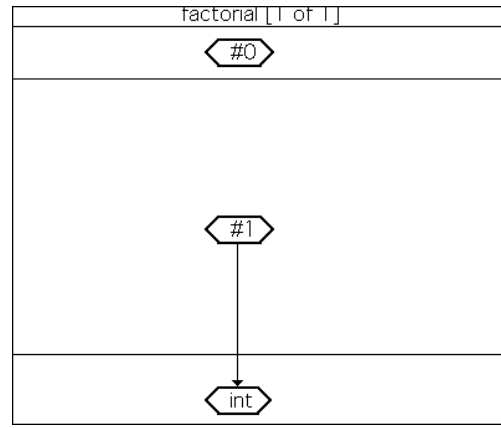


Figure 16b. The specific case.

Recursion can be used whenever there is at least one clear termination condition that can be guaranteed to be reached. This does not have to be numeric as can be seen from the next example (figure 17). Here we are to find the difference in length between two lists by pairing each item in one list with the other. When there is no more pairing possible (i.e. one of the lists is empty) the remainder of the list which is not empty is returned. In this function the library function 'tail' is used which removes the first item from the list and returns what is left.

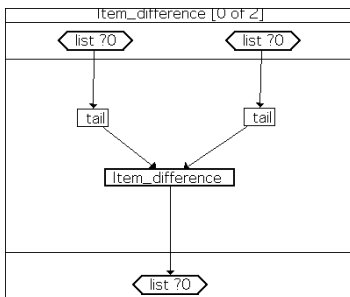


Figure 17a. General

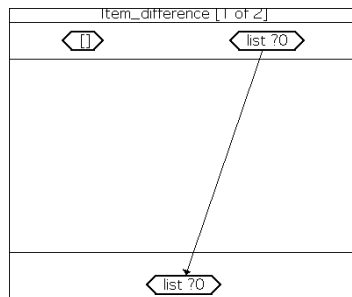


Figure 17b. Specific

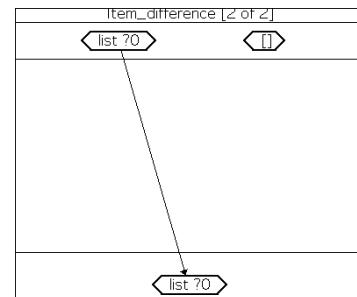


Figure 17c. Specific

The queries:

```

QUERY> Item_difference ['a' 'a' 'a'] ['b' 'b']
['a']
QUERY> Item_difference ['a'] ['b' 'b']
['b']
QUERY> Item_difference [] []
nil

```

tests the definition.

### Conditional Control

Programming also needs conditional control so that processes will treat some data items differently to others depending upon some test. One of the tests is where a particular data item is equal to some form or structure. This can be dealt with by using the in-built pattern matching as we have done in recursion above. So in this example (figure 17) we have it such that when the left hand list is empty we stop the processing and return the right hand list and vice versa. This approach to control is by far the best and should be used wherever possible (as supported by the results of Petre and Green, 1993). However, the tests for some situations do not conform to a simple pattern match.

Consider a situation where we have two objects on a plane each of a different size. Now suppose one of the objects was to move to a new co-ordinate position, can we provide a function that will show if the two objects collide? Now collision in this case occurs if the two objects are within 0.1cms of each other.

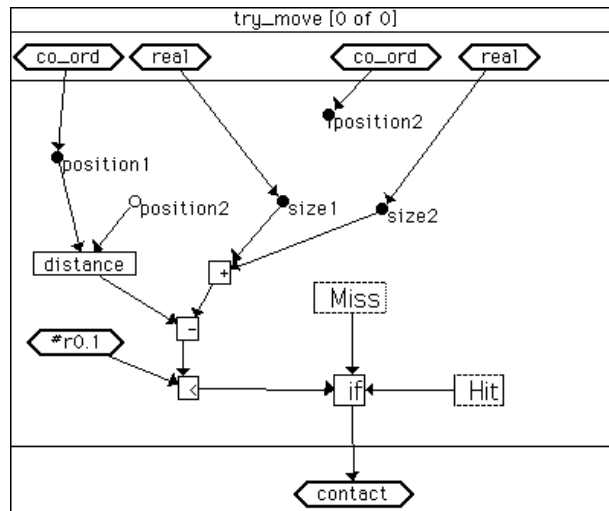


Figure 18. Detecting the collision of two objects.

Figure 18 shows a solution using the user defined co-ordinate operation 'distance'. If the distance is less than the sum of the two sizes (radii) + 0.1 then the constructor Hit is returned otherwise Miss. The constructors Hit and Miss could be replaced by any pair of functions. The function 'if' can be considered to be equivalent to a Boolean controlled two-way switch where the first parameter is the toggle and the other two parameters the alternatives. **Note** that this works in reverse (or upside down) to the normal flow diagram equivalent but is similar to that found in spreadsheets.

### Higher Order Functions

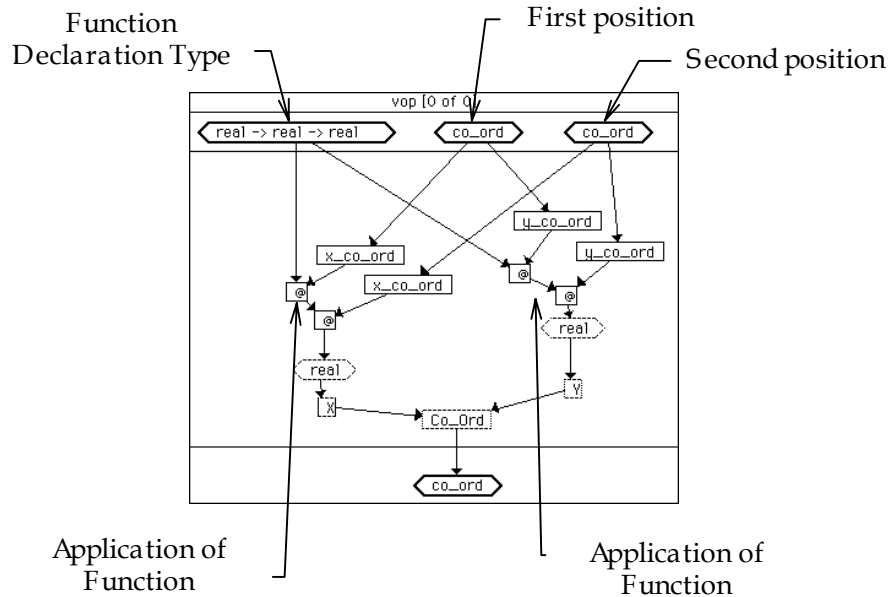
When making new functions for manipulating co-ordinates many of the functions are very similar. For example, the addition and subtraction of two co-ordinates only differ by a single function. Now it is possible to design a function that will accept a function as one of its parameter values. If this can be done then all we need do is design a single function that will take one from a range of potential functions and apply it within itself to produce a unique result. However, a guard is placed on the type of function that can be passed over as a parameter value. So an extension of the type checking ensures that the correct form of function (function type) is deployed.

Figure 19 illustrates a possible solution to providing a generalised vector operation (vop) where any function of the type (real -> real -> real) can be used to combine two co-ordinates to produce a co-ordinate result. We have then:

```

QUERY> vop - (Co_Ord (X #r3.7)(Y #r5.2)) (Co_Ord (X #r2.4)(Y #r3.71))
(Co_Ord (X #r1.300000) (Y #r1.490000))
QUERY> vop + (Co_Ord (X #r3.7)(Y #r5.2)) (Co_Ord (X #r2.4)(Y #r3.71))
(Co_Ord (X #r6.100000) (Y #r8.910000))
QUERY> vop / (Co_Ord (X #r3.7)(Y #r5.2)) (Co_Ord (X #r2.4)(Y #r3.71))
(Co_Ord (X #r1.541667) (Y #r1.401617))
QUERY> vop * (Co_Ord (X #r3.7)(Y #r5.2)) (Co_Ord (X #r2.4)(Y #r3.71))
(Co_Ord (X #r8.880000) (Y #r19.292000))

```



**Figure 19. The creation of a higher order function**

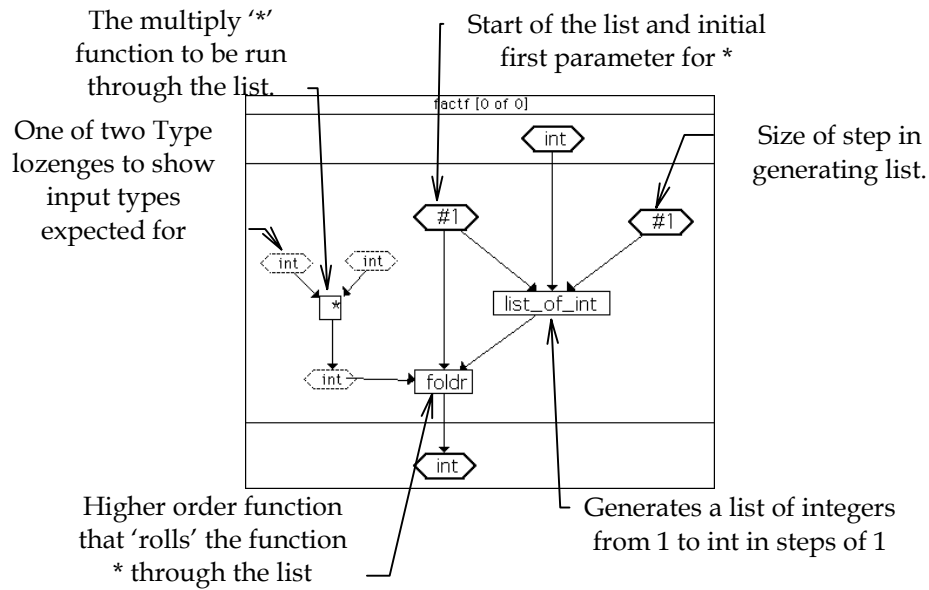
Any function that takes on two real numbers and returns a real number can be used. However, not all of them will be useful. Note that the apply function '@' can only apply one parameter at a time. In this case, because there are two parameters, the '@' function is used twice; once for each parameter of the incoming function.

### Non-Recursive approach to Loops

There are built in library functions which help to perform what is often a loop without resorting to recursion. This has the advantage in most cases of not using the stack. Consider factorial again. Another way of defining factorial is as:

$$\text{Factorial (N)} = [1 * 2 * 3 * \dots * N]$$

We have a library function (list\_of\_int) that will generate a list of integers from any number to any greater number with a specified step. What is also needed is a means of multiplying these generated numbers together. For this we have to call upon the higher order function 'foldr', which refers to the process 'fold right'. This applies the initial parameter (#1 in this case - see figure 20) to the end of the list (N) and moves down the list (N-1) applying the result of the operation to be folded (\* in this case) to each number until there are no more numbers in the list. The result is returned. Fold left 'foldl' starts at the beginning of the list. In this case, it makes no difference, in other cases, it might.



**Figure 20. A non recursive function factorial 'factf'**

```

QUERY> factf #0
#1
QUERY> factf #1
#1

```

```

QUERY> factf #5
#120
QUERY> factf #9
#362880

```

### Non-Recursive approach to Iteration

There is a higher order function called 'iterate' that captures all the essential elements of normal iteration. However, it is not often used since many of the desired results can be done more directly with the function 'map'. Map is used to apply a function to a list of items; items which may be complex. For example, taking a list of paired integers and characters by using the built in constructor 'Pair'. Then we can apply the built-in operation 'first' to each item giving us a list of integers thus:

```

QUERY> map first [(Pair #1 'a')(Pair #2 'b')(Pair #3 'c')(Pair #4 'd')]
[ #1 #2 #3 #4 ]

```

Alternatively, we could have applied the function 'second' to each item to get a list of characters.

```

QUERY> map second [(Pair #1 'a')(Pair #2 'b')(Pair #3 'c')(Pair #4 'd')]
[ 'a' 'b' 'c' 'd' ]

```

On the other hand, we may want a list of integer and character pairs but only have lists of the items separately. The first step is to 'pair' the two lists together. This can be done for two lists of the same length with the function 'zip'. Zip takes any number of lists as a list of lists and generates pairs, triples, or n-tuples depending on how many lists there are. Therefore, we can get our two items together as follows:

```

QUERY> zip [[ #1 #2 #3 #4 ] [ 'a' 'b' 'c' 'd' ]]
[[ #1 'a' ] [ #2 'b' ] [ #3 'c' ] [ #4 'd' ]]

```

However, this is not a constructed pair. To bind the set of items to a function or constructor we need the function 'combine'. This will convert a list of items into the parameters of a function to create an expression. For example:

```
QUERY> combine + [#3 #2]
#5
QUERY> combine Pair [#3 'c']
(Pair #3 'c')
```

So we can 'map' the Curried function '(combine Pair)' over the list of pairs to produce the structure we need.

```
QUERY> map (combine Pair) [ [#1 'a'] [#2 'b'] [#3 'c'] [#4 'd'] ]
[ (Pair #1 'a') (Pair #2 'b') (Pair #3 'c') (Pair #4 'd') ]
```

### ***Lambda*: Controlling the Parameter Assignment**

Now suppose we wanted to divide a number (#3 say) successively with a range of numbers then we could use the notion of the Curried function and write:

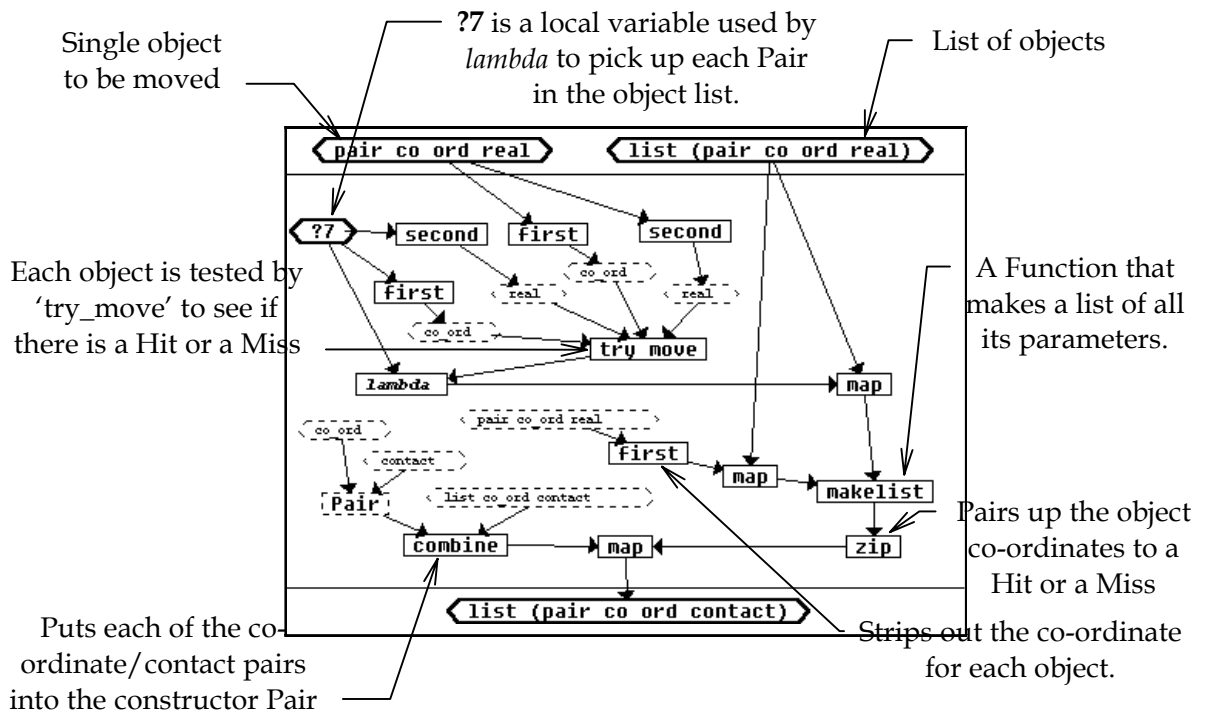
```
QUERY> map (/ #r3) [ #1 #2 #3 #4 ]
[ #r3.000000 #r1.500000 #r1.000000 #r0.750000 ]
```

On the other hand, suppose we wanted to divide each integer in a list by a number (say #3 again). We would have to somehow inform the interpreter that each number to which the function is applied must be treated as the first parameter of divide (/) and not the second as was done above. For this, we need the 'interpreter instruction' (it is not a function) '*lambda*'. Now in Clarity *lambda* always appears in italic since it is not recognised as a function or constructor. In the control window, we can write the function we need thus:

```
QUERY> map (lambda ?7 (/ ?7 #r3)) [ #1 #2 #3 #4 ]
[ #r0.333333 #r0.666667 #r1.000000 #r1.333333 ]
```

The use of all these functions can now be drawn together to produce the function 'check\_move'. This function takes a list of 'objects' each of which are represented by a position and size pair. 'Check\_move' assesses if a move to a particular position is going to cause a Hit with any of the objects and if so which one.

Figure 21 shows the final version in Clarity. The list of objects is given in the second parameter and each of these is tested with 'try\_move' to see if there is a Hit or a Miss. Since 'try\_move' was defined with the position and size of each object as separate parameters then each object has to be 'unpacked' to get at the individual values. Since this is a one off function *lambda* is used to provide the parameter identification in a temporary function. This temporary (and unnamed) function is applied to each object in the list returning, for each object, a Hit or a Miss.



**Figure 21. A function to check over a range of objects for a Hit or a Miss**

The list of Hits and Misses are combined with a list of positions obtained by applying the function 'first' to each of the pairs and the two lists form a list of lists through the function 'makelist'. The function 'makelist' is one of the few functions that can take on any number of parameters. It is unique to Clarity/Faith because it services the diagrammatic representation and allows 'lists' to be expressed.

The function 'zip' pairs each item of each list in order returning another list of lists; this time of position and contact sub-lists. The sub-lists are converted to a position and contact pair. The function 'combine' is used on each sub-list. An example of 'check\_move' is:

```
QUERY> check_move (Pair (realtoc #r3.7 #r5.2) #r1.5) [(Pair (realtoc #r2.3 #r4.6) #r2.3)(Pair
(realtoc #r7.3 #r8.1) #r0.5)]
```

```
[ ( Pair ( Co_Ord ( X #r2.300000) ( Y #r4.600000)) Hit) ( Pair ( Co_Ord ( X #r7.300000) ( Y
#r8.100000)) Miss) ]
```

An annotation window can be associated with each diagram and any token within the diagram. The purpose of the annotation is to help keep a record of the informal semantics of the function. However, much more can be done by keeping the diagram simple and by choosing names that have meaning. Both these strategies can be applied to 'check\_move' by replacing the *lambda* expression with a single function as shown in figure 22. This function tests to see if there is a 'hit\_or\_miss' and will return 'Hit' if there is contact with another 'object' and 'Miss' otherwise.

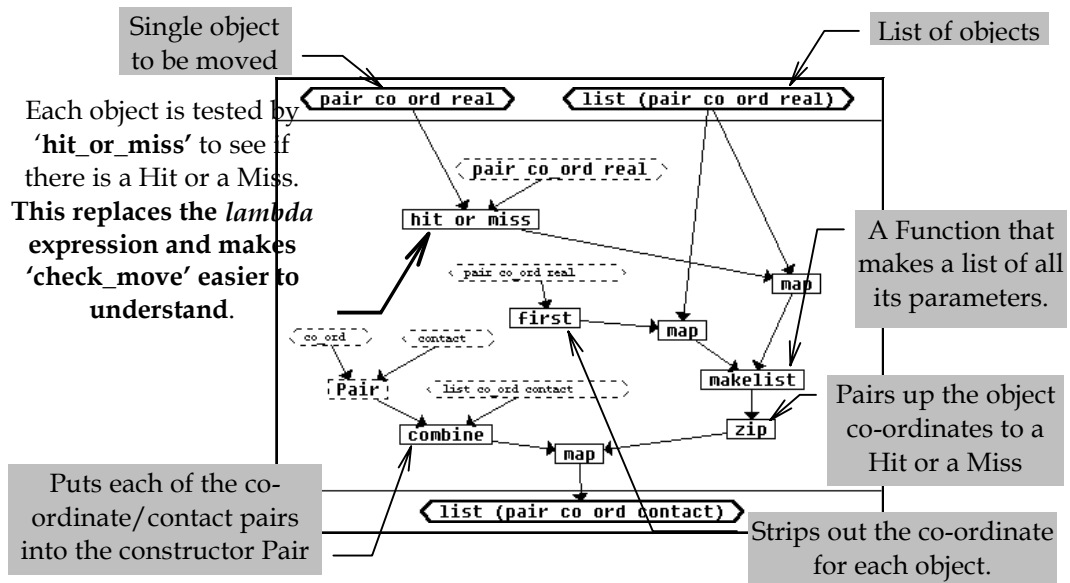


Figure 22. Replace lambda expression by a single function

Many library functions help provide mechanisms for creating and generating Faith code. A full appreciation of these functions can only be obtained through looking at the manual.

### Benign and Malignant Side-Effects

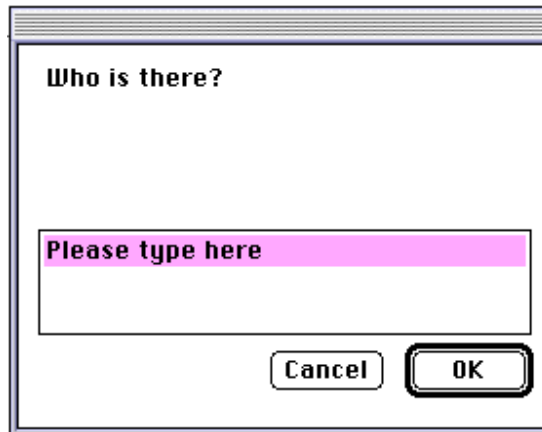
A functional program in its pure sense exists in the abstract world of mathematics. In practice, it has effects on the world and in particular it affects the states of a computation engine. Benign side effects of a function do not interfere with the normal flow of a functional program; the call of a function merely triggers events in the world. These events, in principle, could be anything and they may not relate to the meaning of a function in a program. For example, an evaluation of the function 'add1' might (but does not in our case) have the side effects of setting a kettle to boil or causing an alarm to be sounded. However, most side effects do have some sensible events attached to them. Examples are:

```
QUERY> print "Hello world"
"Hello World"True
```

will print the string "Hello world" in the control window. The function itself is defined as:

```
str -> bool
```

and in nearly every case the response will be 'True' unless there is some physical problem why a the string cannot be output to the window. The 'print' function is considered to be associated with mostly benign side effects; mostly because there is a very small chance that it could return 'False'.



**Figure 23. The gr\_dialog side effect**

A slightly less benign function is 'gr\_dialog' is shown in figure 23. This is defined as:

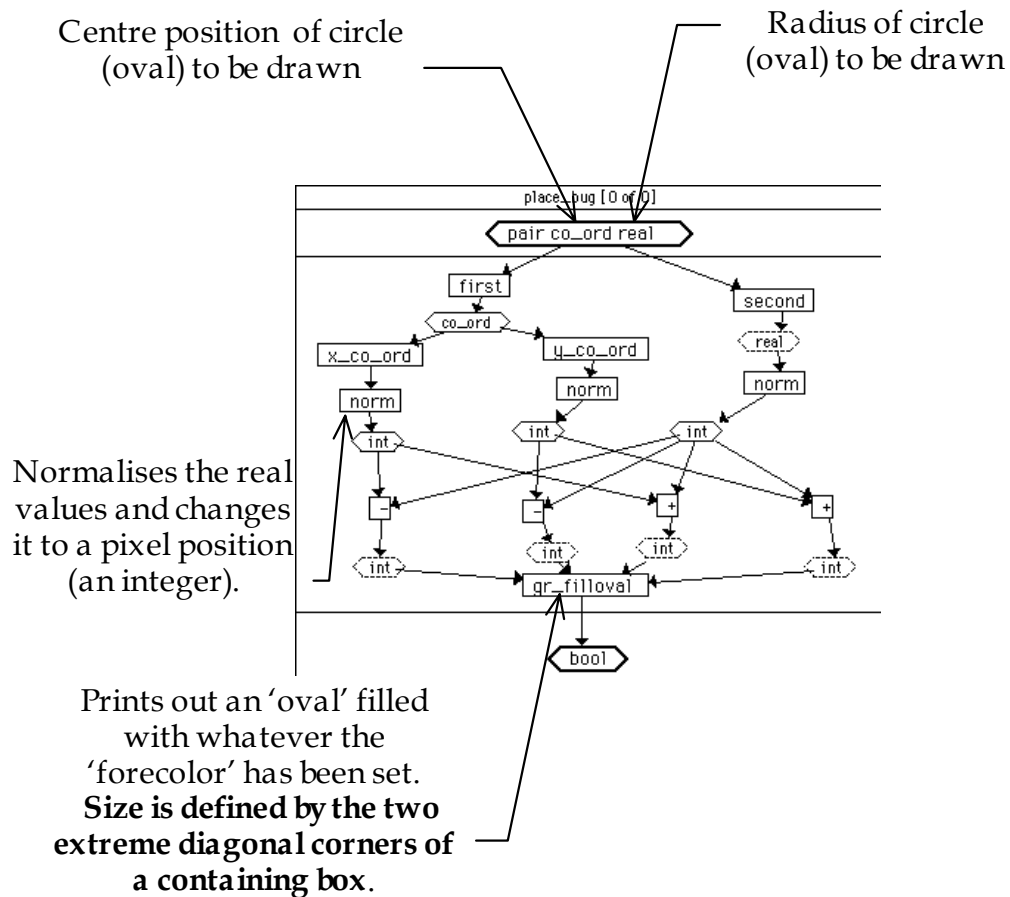
```
str -> str
```

and is called as follows:

```
QUERY> gr_dialog "Who is there?"  
"Joe Young"
```

The above 'query' will cause a dialog box to be displayed and the returned string will be what were the contents of the dialog inner box. This string can then be used to perform some task, which may influence the flow of a program.

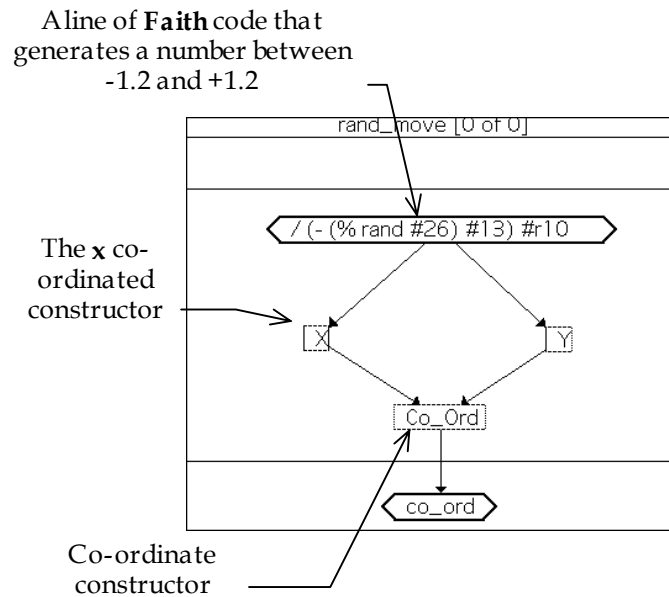
- *The main issue of 'malignant side effects' is that they cause the mapping of a function to be indeterminate.*



**Figure 24. A function that draws a coloured circle in the graphics window.**

Figure 24 is an example of a user-defined function that draws a coloured circle in the graphics window. This circle is filled with whatever the 'forecolor' function last specified. The function 'gr\_filloval' takes whatever shape an enclosing box defines and this box is expressed in terms of the diagonal line drawn between its two extreme corners (top left and bottom right). However, the function defined expresses the circle in terms of its centre position and radius.

Figure 25 illustrates the ultimate example of an indeterminate function. This one calls the library function 'rand' that generates a random integer (needs to be seeded by the function 'srand'). This integer can be contained by the modulus function '%'. The function '%' does an integer division and returns the remainder. In this case, it integer divides the random number by #26 and therefore the remainder is always going to be less than #26. #13 is subtracted so that we can obtain both positive and negative numbers. This function is called twice; once for the x and once for the y co-ordinate position.



**Figure 25. A function that generates a random co-ordinate change**

Figure 25 is also an example where a line of Faith code is used within a Clarity diagram. In this case, the random number generator is so simple it is clearer to use a bit of program code. However, the danger is that anything in a parameter lozenge and used in the body of a function definition is *not type checked*.

### Functions as Data

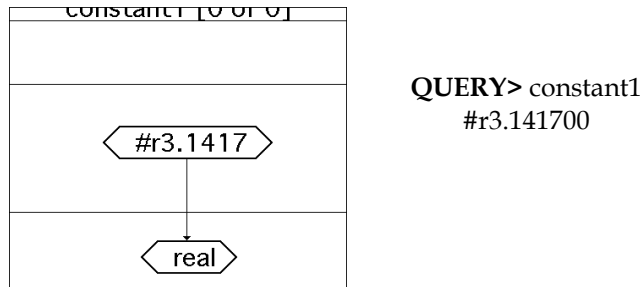
The purpose of a functional language is to avoid the problems of the imperative languages. The pure functional language gains its advantage through the idea that a function will always respond exactly the same way no matter when or how its called. This characteristic of a stable response is called *referential transparency*. The problem with imperative languages is that they rely upon referencing stored data and this data may be changed anywhere and at any time in a program. In many cases, this will result in a procedure (equivalent to a function) responding differently at different times to the same input. Imperative programs will perform differently depending upon their history. This characteristic is fundamental to imperative programming since such a program depends totally upon side effects.

For this reason, much work has gone into debugging tools for imperative languages that allows the programmer to trace the history of a process in order to 'see' at what point the 'error' occurs. Such tracing forms a major part of the activities of a programmer. An interpreted and pure functional language should need no such debugging tool since a function will always behave in exactly the same way at all times. Errors are thus easier to find.

However, there is a price to pay for referential transparency; **there is no place to 'store' data**. All data that represents the 'state' of the world has to be passed on from function to function through the parameters. In the simple case this does not matter but where the 'state' of the world is complex then there is one major side effect that will catch the functional programmer out; that is a **stack overflow**. Further, the functions

become very 'busy' as each function has to 'handle' the 'state' of the world in order to access or update the data. The problems involved in handling such states can cause the function definitions to be clumsy and unwieldy.

If we were to define a function with no parameters such that when it is called it will return a value we can then use this function as a global constant that is accessible anywhere in the programme.



**Figure 26 A function of constant value**

It is possible to change this constant at any time by using the function 'setq' thus:

```

QUERY> constant1
#r3.141700
QUERY> setq constant1 #r5.622
True
QUERY> constant1
#r5.622000
  
```

Functions can have parameters and thus each component of a function can respond with a value. In this way a function becomes an array with the option of having indexes of any kind (integer, string, real, char, list ?, or any user defined type). This gives considerable power to the expressiveness of the language. Further, it allows the existence of sparse arrays without any overheads. To make such an assignment the function 'setq' (and its alternatives such as 'assertq') can take on any function description such that:

```

setq (<function> {<index>}) <body>
  
```

is its general form. So not only values but also expressions and constructed values can be assigned to a function component within the program. For example, suppose we wanted to keep information about a set of objects such as their current position and size. Figure 27 shows a component of the function 'bug' with a constructed set of information.

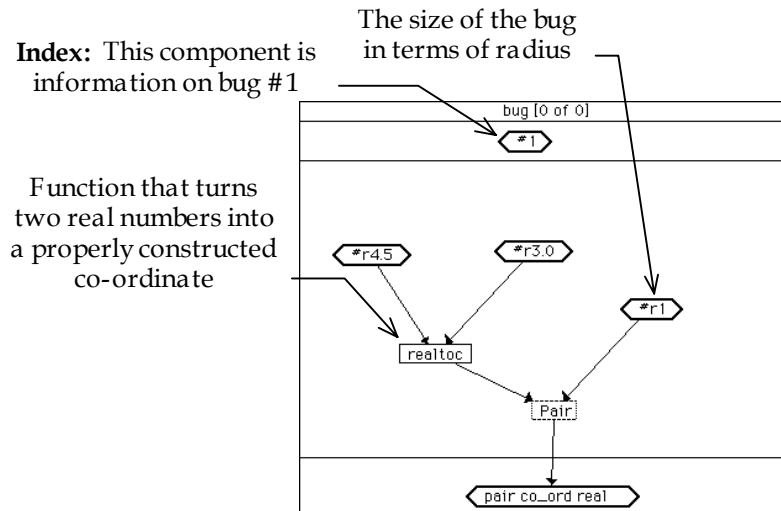


Figure 27. A bug and its details.

A function can now be constructed which will assign a new value to a 'bug' where this value is some complex construct. Figure 28. is such a function but in this case we have used 'assertq' instead of 'setq'. The function 'assertq' has a variable number of parameters so that the component can be identified with out the need for extra brackets.

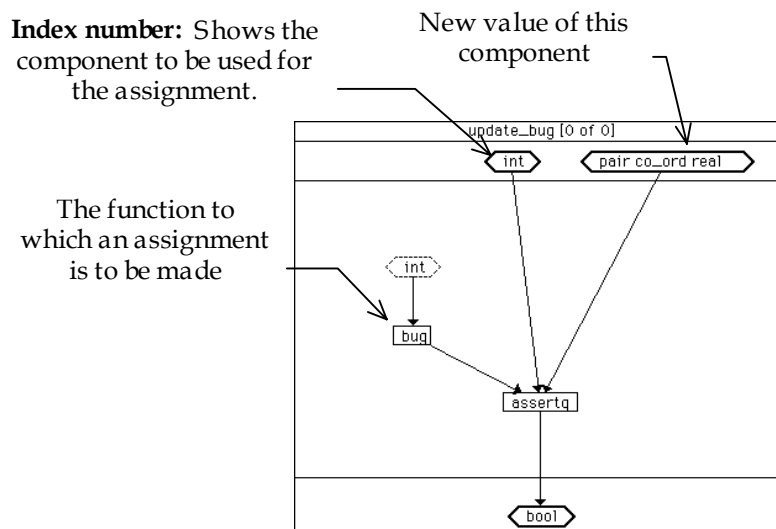


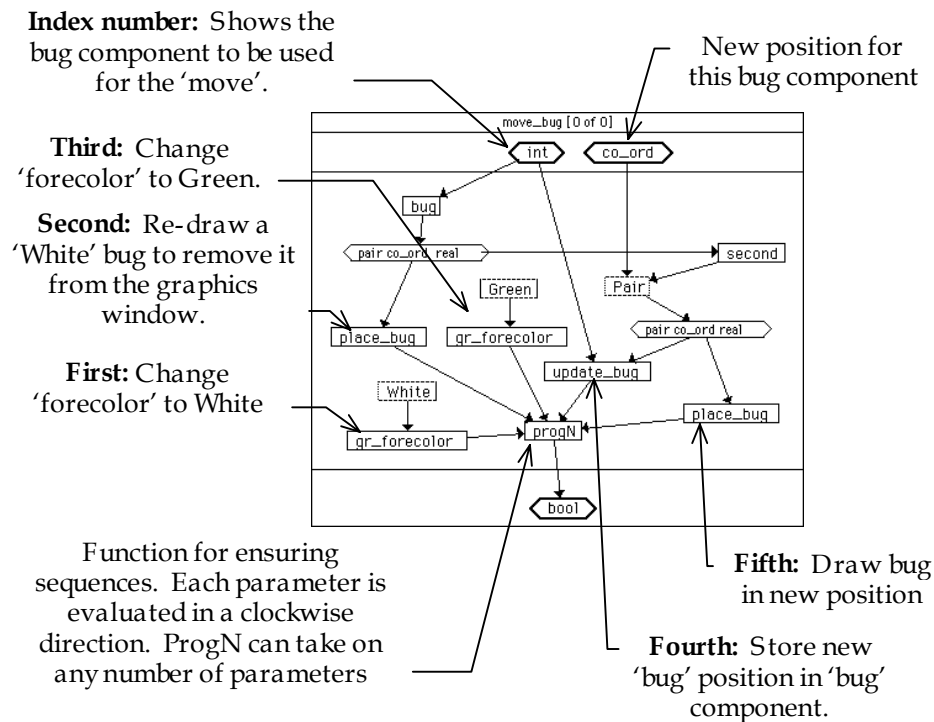
Figure 28. Function for changing a 'bug' position

## Sequences

Unless there is a requirement to keep an explicit track of time or the ordering of events, a functional language will apply functions in an unknowable order provided the function is complete (has all the parameter values). Some control can be achieved through the nesting of the 'if' function which can be arranged to act in a similar fashion to a set of up-ended dominoes; the completion of one set of functions triggering off the next. However, all this is very clumsy and it is better to provide a set of functions

whose sole job is to order events. One of the most useful is the function 'progN' and its variants.

The function 'progN' can take on any number of parameters and will evaluate them in strict sequence starting from the first parameter and finishing on the last. In this case, the final evaluation of 'progN' is to return the evaluation of the last parameter. All the other evaluations are ignored. This means that the other evaluations do not exist as far as the functional program is concerned. The only results will be through side effects.



**Figure 29. The ultimate in side effect programming.**

Figure 29 is an illustration of side-effect programming and shows its usefulness. Most of the operations are benign and this limits the risk involved in evoking side-effect functions. In this example, it is vital that the correct order of events happen. The moving of a 'bug' must involve the sequence of eliminating its appearance in the old position and redrawing it in the new position and at the same time updating its new position in the function 'bug' (potentially malignant).

## Conclusions

Our first paper (Addis & Addis, 2002) concentrated on the causes of the failure of diagrams to be adopted for programming. We showed that much of the criticism of graphical notation is due to the imperative nature of programming rather than the graphical form. Many of the limitations that were observed are avoided provided a functional programming language is used. The major source of the problem is traced to 'knotty structures' where there are deeply embedded 'if's. We showed that functional programs use ten times fewer 'if's than imperative programs for a wide range of industrial problems.

Extending the functional language to include some imperative methods further reduces the complexity of program structures without reintroducing knotty structures. This is because most knotty structures can be resolved through the mechanism of exact pattern match. We have found (as those that developed the functional language LISP before us have found) that in practice the pure functional language can become inelegant. This is particularly a problem when dealing with systems where large and complex structures are passed from function to function. We introduced 'side effect' programming where function definitions can be created and extended during the running of a program. Thus, we resolved this difficulty of the passing on of complex structures by using the function as a very flexible and accessible storage array; the function acts as a global variable.

The loss of control some imperative methods brings about is minor since the bias is always to work functionally. What we have illustrated in this paper is that there is an advantage to be gained in balancing the functional with the imperative approaches. However, this advantage can only be retained provided the functional language element dominates. Combine this with a visual representation that encourages simplicity in expression then complex systems can be easily put together. Given the support of viewing different and user selected projections of the program (i.e. a network window) then the management, the maintenance and the evolution of a complex system is easily achieved.

Modelling may be considered as the process of creating computational metaphors of a set of observed constraints. Constraints are completely definable through the mechanism of mapping (figure 1) and the formal description of mapping is the function. In addition, a functional description retains the important distinction between the informal mapping required to interpret the relationships between the model and the modelled. This is primarily done through constructor functions and the naming of functions in general. What the representation of functions does *not* do is provide the insight to create concise, simple and elegant descriptions; this must come from the person doing the modelling (see later).

As a modelling tool, the functional language has the advantage of being neutral. The language is a means of creating a formal abstraction of the world that is independent of any theories of the world. It does not, for example, presume that there are objects in the world nor does it assume that the world is divided into data and processes. The only assumption is that the world can be described in terms of mappings.

The simplicity of the language is unique. The only computational mechanisms required for any description are the *exact pattern match* and *recursion*. These two processes alone can define any formal process. The simplicity of this is illustrated in our sister paper (Addis & Addis Appendix, 2002). There we describe how integer arithmetic can be defined from first principles in a few simple steps and without recourse to any library functions. This involves defining numbers and the arithmetic operations on them in terms of mappings. Similarly, it is possible to extend this to real

arithmetic, Boolean algebra, predicate calculus, multivalued logic and proof mechanisms. *We can thus infer that it is possible to build any formal system based on just these two computational mechanisms.*

The mapping between the Clarity representation and the functional language is complete (see Appendix 2). The 'pay off' for using a graphical representation is that the *structure* of the description becomes clearer even though the *logic* of the system is not made any more explicit than in its sentential form. Such a 'pay off' provides some control over managing the design of complex systems. However, the apparent imperative style of Clarity schema allows them to be interpreted as though they were data/process flow diagrams. In fact, they are explicit relationship diagrams. Never the less, experienced programmers can find such a representation unhelpful because they already have developed very sophisticated skills to solve the mapping of world domains into code. On the other hand, those who are used to dealing with diagrams, such as engineers, find the Clarity representation obvious and natural because they normally have a highly developed set of skills for interpreting such schematic drawings.

Finally, complex problems remain complex. We can make the management of solving such problems easy by using the computer to support our process of creativity. The major issue is always in getting the ontology right (Devedzic, 1999, Gamma et al 1995) and for new problems this requires active exploration. Trial and error investigations are the triggers to insight and are usually done by designers through the extensive use of pencil and paper. Designers use pencil and paper because it is inherently fast, flexible and error free. This approach creates a separate stage since once a design has been sketched there are problems of translating that sketch into a trial system. This has always been considered an advantage because there are problems in the process of coding that make a complex problem even more difficult. Systems analysis methods, based upon this multiple stage approach, are often invoked as a guide to getting to grips in the first place with a complex problem. Such methods still require multiple attempts before the 'right' answer is achieved. Such multiple attempts are often very expensive because they involve more than one specialist.

If the process of programming was not so destructive and was a positive force towards solving the problem and designing the system then such a multiple stage process would become redundant. The Clarity environment does seem to give us a tool where the pencil and paper now becomes active and powered by a computer. Solutions can be quickly tried and tested dynamically as each component is tested and modified. Mechanisms for gross changes, such as restructuring the range of objects, are available and every move in programming is checked for consistency at the time of production. Such an environment cuts out the different stages of systems design by merging the initial exploratory stage into the creation of a final product. The final product is then checked for consistency. When it is complete, it can still be seen in the form of its original graphical design and is thus always open to be adapted in response to the changes of the world it embodies.

## Acknowledgements

Much of the Apple Macintosh work on CLARITY has been supported through the projects *Knowledge Design from Natural Science Experiments*. MRC (No SPG 9107137) and *Graphic Programming Methods for Modelling Cognitive & Social Process of Science*, ESRC (No R000235286), investigators: Gooding D. and Addis T. R. However, Addis J. J. T. has developed the environment for the PC since 1998 without support and in response by others. The Warsash Maritime Centre and Portech Ltd have contributed to raising the environment into an industrial tool. Clarity Support Limited (<http://www.clarity-support.com>) has been set up specifically to provide a service for this environment.

## References

- Addis T. R., Booth D. J. W. and Townsend J. J.** (1992) 'Clarity 1.0: A Reactive Visual Programming Environment' Unpublished report Alvey Project IKBS 140, University of Reading.
- Addis T. R. and Townsend Addis J. J.** (1996) '*Clarity: Built-In and System Functions with FAITH Examples*', Version 3.6.5, May 1996. [for http://www.sis.port.ac.uk/research/clarity/index.html](http://www.sis.port.ac.uk/research/clarity/index.html).
- Addis T. R. and Townsend Addis J. J.** (1996a) '*The Clarity Manual*', Version 3.6.5, May . [for http://www.sis.port.ac.uk/research/clarity/index.html](http://www.sis.port.ac.uk/research/clarity/index.html).
- Addis T. R. and Townsend Addis J. J.** (1998) '*A Functional Schematic Interpreter: an environment for Model Design* ', International Journal of Systems Research and Information Science, Vol. 7, ISSN 0882-3014, Pages 263-291.
- Addis T. R. and Townsend Addis J. J.** (Expected 2002) '*Avoiding Knotty Structures in Design: Schematic Functional Programming*', Journal of Visual Languages and Computing, Vol 12.
- Blackwell A. F.** (1996) '*Metacognitive Theories of Visual Programming: What do we think we are doing?*' Proc. IEEE Symposium on Visual Languages. pp240-246.
- Devedzic V.** (1999) '*Ontologies: Borrowing from Software Patterns*' ACM Intelligence, Vol 10, Number 3 Fall. Pp 15-24.
- Field A. J. & Harrison P. G.** (1988) '*Functional Programming*' Addison-Wesley.
- Gamma E., Helm R., Johnson R. and Vlissides J.** (1995) '*Design Patterns: Elements of Reusable Object-Oriented Software*' pub Addison-Wesley, ISBN 0-201-63361-2.
- Green T. R. G.** (1990) '*Programming Languages as Information Structures*' Psychology of Programming, Academic Press, ISBN 0-12-350772-3.
- Green T. R. G & Petre M.** (1996) '*Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework.*' Journal of Visual Languages and Computing 7, pp 131-174.
- Larkin J. H. & Simon H. A.** (1995) '*Why a Diagram is (Sometimes) Worth Ten Thousand Words*' in Diagrammatic Reasoning: Cognitive and Computational Perspectives. eds. Chandrasekaran B, Glasgow J. and Narayanan H. N. AAAI Press/ The MIT Press. pp. 69- 109
- Jaques E.** (1978) '*Levels of Abstraction in Logic and Human Action*' pub Heinemann, ISBN 0 435 82280 2.
- Petre M. & Green T. R. G.** (1993) '*Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill*'. Journal of Visual Languages and Computing. Vol 4 pp 55-70.

- Poulovassilis, A.** (1988) *FDL: An Integration of the Functional Data Model and the Functional Computational Model*, BNCOD6, CUP, pp 215-236.
- Poulovassilis, A and King P.** (1990) *'Extending the Functional Data Model to Computational Completeness* EDBT-90 (ref. Dept of Computer Science, Birkbeck College, University of London).
- Reade C.** (1989) *'Elements of Functional Programming'* Addison Wesley, ISBN 0-201-12915-9.
- Shaw L. G. & Woodward J. B.** (1990) *"Modeling Expert Knowledge"*, Knowledge Acquisition ,vol. 2, pp. 179-206 .

# Appendix 1

## A BNF Description of the Functional Data Language Faith.

### **The Faith Code Window**

#### ***Fdb Section***

<fdb section> ::= <fdb commands> <terminator>  
 <fdb commands> ::= <fdb command> <fdb commands> | <empty>  
 <fdb command> ::= <declaration section> |  
                   <definition section> |  
                   <query> |  
                   <dbport> |  
                   <memory events>  
  
 <terminator> ::= ;  
 <empty> ::=

#### ***Function and Constructor Declarations***

<declaration section> ::= fdec <declarations> <terminator> |  
                           cdec <declarations> <terminator> |  
                           tdec <tdeclarations> <terminator>  
  
 <declarations> ::= <declaration> <declarations> | <empty>  
 <tdeclarations> ::= <tdeclaration> <tdeclarations> | <empty>  
 <declaration> ::= <function name> <is> <types> <terminator>  
 <tdeclaration> ::= <function name> <is> <typeop> <terminator>  
 <function name> ::= <symbol>  
 <is> ::= ::==  
 <types> ::= <type> <to> <types> | <type>  
 <typeop> ::= typeop <int val> | nonlex | enum  
 <to> ::= ->  
 <type> ::= <function type> | <list type> | <pair type> | <basic type> | <variable>  
 <function type> ::= (<types>)  
 <list type> ::= (list <type>)  
 <pair type> ::= (pair <type> <type>)  
 <basic type> ::= int | real | char | str | bool  
 <variable> ::= ?<natural number>

#### ***Function Expressions***

<expression> ::= <fexpression> | <lambda expression> | <holdn expression> |  
 <constant>  
 <fexpression> ::= <function name> <arguments> | <combinator> <arguments>  
 <lambda expression> ::= lambda <variable> <expression>  
 <holdn expression> ::= HOLDN <int val> <expression>  
 <arguments> ::= <argument> <arguments> | <empty>  
 <argument> ::= <constant> | <list arg> | (<expression>) | <function name> | <variable>  
 <constant> ::= <int val> | <real val> | <char val> | <str val> | <bool val>  
 <list arg> ::= [<arguments>] | (makelist <arguments>) |  
                   (: <argument> <list arg>) | nil

<int val> ::= #<integer>  
 <real val> ::= #r<real>  
 <char val> ::= '<char>'  
 <str val> ::= "<chars>"  
 <bool val> ::= True | False  
 <chars> ::= <char> <chars> | <empty>

### **Function Definitions**

<definition section> ::= fdef <definitions> <terminator>  
 <definitions> ::= <definition><definitions> | <empty>  
 <definition> ::= <fexpression> <is> <expression> <terminator>

### **The Control Window**

<query> ::= <expression> <cterminator> | <constant> <cterminator>  
 <dbport> ::= load <file name> <cterminator> |  
 open <file name> <cterminator> |  
 create <file name> <cterminator> |  
 exec <file name> <cterminator> |  
 close <cterminator> |  
 commit <cterminator>  
 <cterminator> ::= <terminator> | <newline>  
 <file name> ::= <str val>  
 <memory events> ::= trace <bool val> <terminator> |  
 dump <str val> <terminator> | dump <terminator> |  
 gc <terminator>

### **Basics**

<integer> and <real> are read by C's scanf, they are terminated by any of the characters:  
 : ; newline space ( ) [ ]

They are not terminated by '-'. The following are valid:

<integer> 0 -1 -100001 100021  
 <real> 3.14 2.0 -24.5 36.7e-15

<character> any printable character

<symbol> any sequence of printable characters except combinators and the following reserved symbols:

::= ; -> ( ) [ ] cdec, tdec, fdec, fdef, load trace open, creat, exec, close, commit, lambda int real  
 char str bool list pair and any other built in library functions

<combinator> ::= B | C | I | K | S

## Appendix 2 An extension of Faith to the Schematic Language Clarity.

### A Schematic Extension of BNF

The underlying structure of the Clarity schematic language is an acyclic bipartite directed graph. The graph relates to the Faith code and this will be given in terms of the BNF structure descriptors (Appendix 1). The two types of node are a collection of functions (F) and a collection of types (T). The nodes are connected via an input mapping I and an output mapping O. The input mapping I maps a type node  $t_j$  to a collection of function nodes  $I(t_j)$ . The output mapping O maps a type node  $t_j$  to a collection of function nodes  $O(t_j)$ . A schema consists of a four-tuple thus:

$$\langle \text{Schema} \rangle = (F, T, I, O)$$

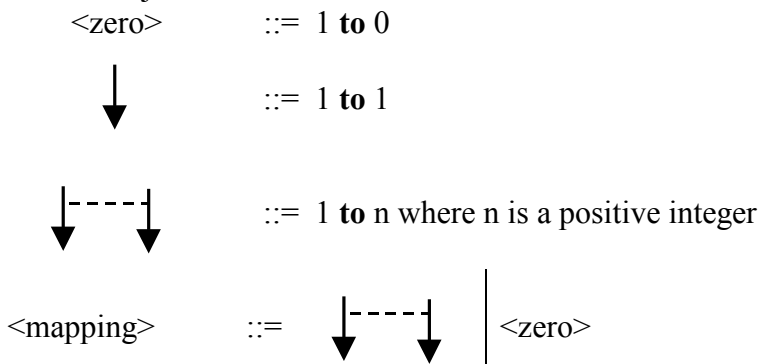
The degrees of freedom of these mappings are governed by a syntax that will be expressed in terms of a graphical extension of BNF. This extension introduces a second dimension into the normal BNF meta-language that shows the context dependent cardinality constraints of the mappings. A graph that represents a Clarity schema will be described such that the input mapping for a node is always placed above the description of that node and the output mapping will always be placed below. The resultant triple will be treated as single BNF symbol.

$\langle \text{input mapping cardinality to node} \rangle$

$\langle \text{node} \rangle$

$\langle \text{output mapping cardinality from node} \rangle$

We will illustrate a mapping  $\langle \text{mapping} \rangle$  by a collection of arrows. The input mapping will converge onto a node and the output mapping will diverge away from the node. A single arrow represents a 2-tuple either input  $\{ f_i, t_j \}$  or output  $\{ t_j, f_k \}$  which maps a single node  $i$  or  $j$  onto another  $j$  or  $k$ .

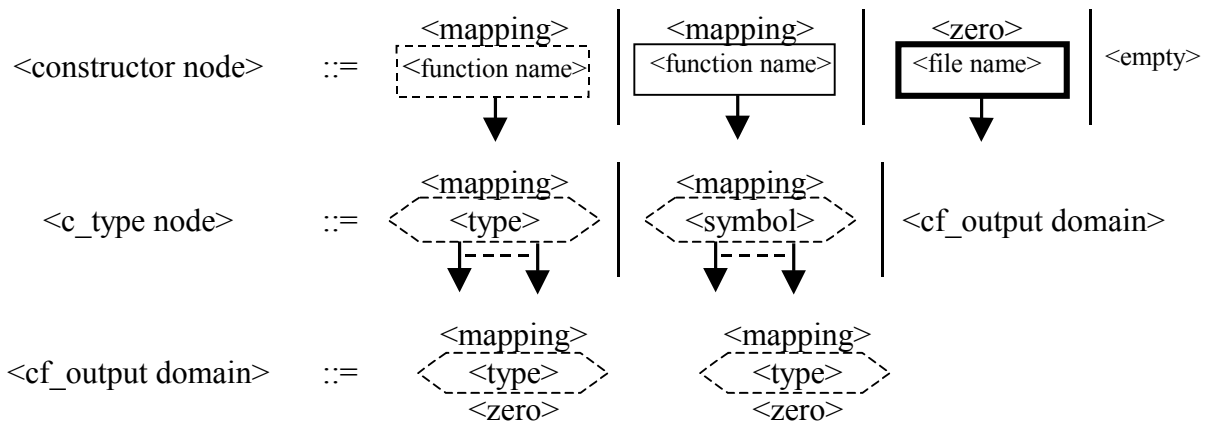


The  $\langle \text{mapping} \rangle$  is constrained by  $\langle \text{types} \rangle$  (see Appendix 1).

### The Constructor Window

The window consists of two domains. A bottom strip is the output domain and the remaining top part is the declaration domain.

$$\langle \text{declaration schema} \rangle ::= (F \ni \langle \text{constructor node} \rangle, T \ni \langle \text{c\_type node} \rangle, I, O)$$



### The Function Window

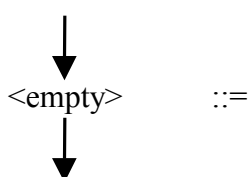
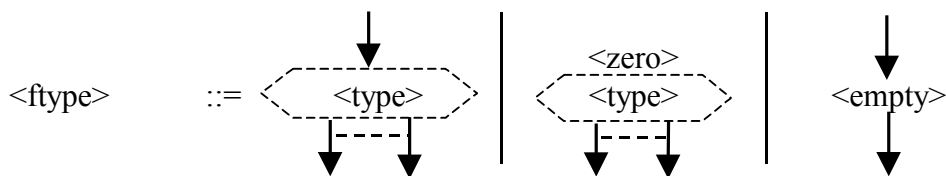
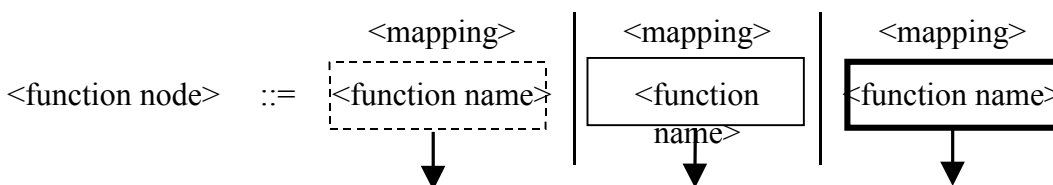
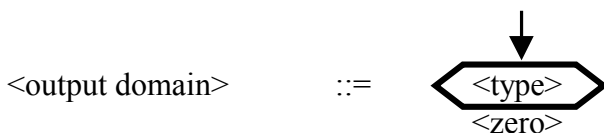
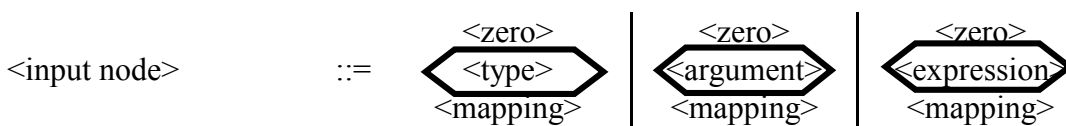
The function window consists of three domains. The top strip is the input domain and the bottom strip is the output domain. The middle domain is the body of the function.

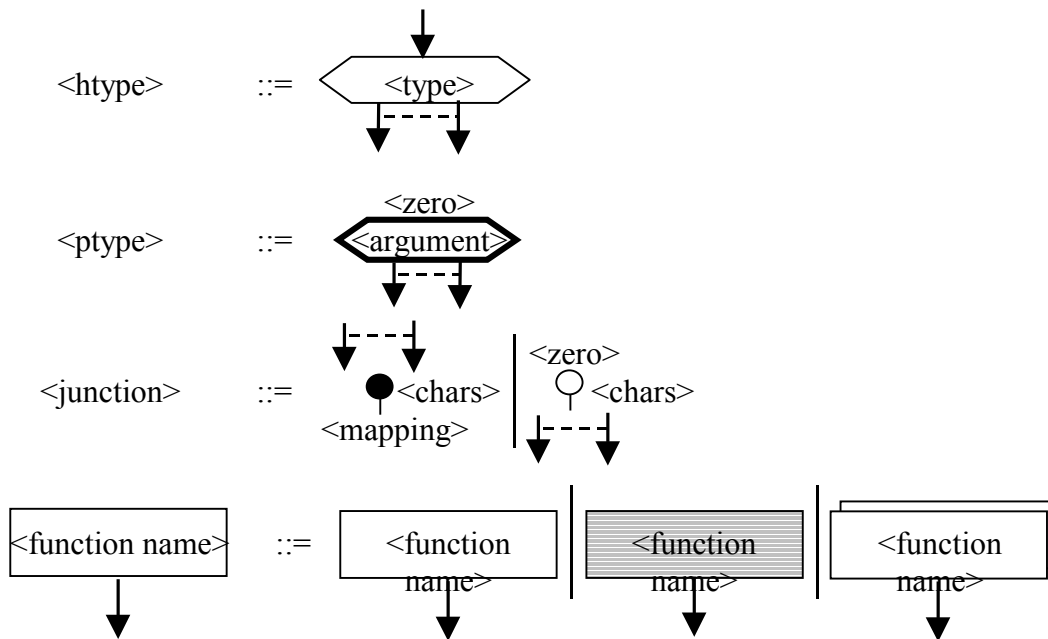
$\langle \text{function schema} \rangle ::= (F \ni \langle \text{f\_function node} \rangle, T \ni \langle \text{f\_type node} \rangle, I, O)$

$\langle \text{f\_type node} \rangle ::= \langle \text{input node} \rangle \mid \langle \text{output domain} \rangle \mid \langle \text{ftype} \rangle \mid \langle \text{htype} \rangle \mid \langle \text{ptype} \rangle \mid \langle \text{junction} \rangle$

$\langle \text{f\_function node} \rangle ::= \langle \text{function node} \rangle \mid \langle \text{junction} \rangle \mid \langle \text{empty} \rangle$

$\langle \text{input domain} \rangle ::= \langle \text{input node} \rangle \langle \text{input domain} \rangle \mid \langle \text{empty} \rangle$



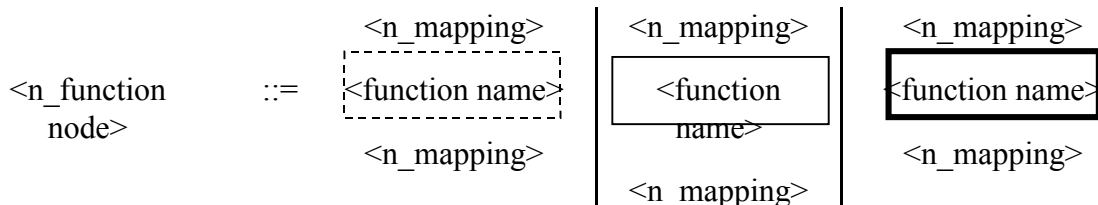


**The Network Window**

The network window has only one domain. A Clarity program can be designed as a collection of functions without regard to the individual functional schemas. The schema is simply a directed graph that may contain cycles. The mappings J and K are now from function to function (no type nodes involved). The input function J maps a function node  $f_i$  to a collection of function nodes  $J(f_i)$ . The output mapping maps a function node  $f_i$  to a collection of function nodes  $K(f_i)$ .

<network schema> ::= (F  $\ni$  <net\_function node>, J, K)

<net\_function> ::= <n\_function> | <n\_junction> | <empty>



<zero> ::= 1 to 0

::= 1 to 1

::= 1 to n where n is a positive integer

<n\_mapping> ::= | <zero>

<n\_junction> ::=