Scalable Connectionless Iconic Programming

Dr John R Rankin
Computer Science, La Trobe University, Australia
j.rankin@latrobe.edu.au

Abstract

An experimental solely icon-based visual programming language has been designed and implemented and is reported here. After the implementation and testing of version 7 a thorough usability testing procedure was pursued before the implementation of version 8. This VPL is specifically designed to not use any connection lines in the visual presentation and to provide a fixed but resizable visual format to avoid the time-consuming visual rearrangement tasks in other icon-based VPLs. The language was also designed to be generic for building applications based upon any chosen system library. The language was made to enforce a number of simplifying programming constraints that foster good software engineering in the design of applications that it produces. The purpose of the implementation of this experimental language is firstly to see if a connectionless iconic programming language in this style is possible, viable and scalable for general purpose problem solving, and then to find its implications for how users must think about programming and software design to use this system well.

Introduction

The use of icons (which are equally sized small square images and often very colourful) for communication is proliferating in the modern world. Icons have been used to overcome the language barrier between different cultures and nations, to simplify instructions and to make web pages more pleasant to use. Instructions for setup or repair of equipment can be found now in printed iconic pictograph form. Typically Instant Messaging services supply icons and "emoticons" to express messages and feelings in a language independent way: it saves some amount of typing and can sometimes express ideas or feelings better than words can. Mobile phones, PDAs and modern touch-based computer devices are using icons as an attractive user interface to music, games, web browsing and other applications. Icons seem to be the modern easy-to-use way to interface between man and computer.

Computer programming even for modern GUI-based computer programs is predominantly text-based. Perhaps after 60 years of this text-based approach it is time for a more direct approach to producing modern graphically-based programs (1). Visual programming (VPL) paradigms (2,3,4,45) have been available for over two decades (for example Visual Basic started in 1991 [47]) and yet many (including Visual Basic) still at least most of the time involve large amounts of exacting text entry of complex logic. The visual aspect of standard Visual Programming typically consists solely of placing and sizing visual components onto the main form of the application and similar GUI designing for dialogue boxes and daughter forms of the application. However when it comes to defining the behaviours and responses of these visual controls the programmer can choose from a selection of pre-coded methods or else he is directed to a text code editor window for entering the appropriate commands in the chosen text-based High Level Language (Basic, Pascal, C/C++, C#, Java or other HLL). Efforts to escape the need for text-based coding have lead to icon-based VPLs. Some icon-based VPLs have been described as programming with boxes and arrows where the boxes may still involve some text code entry. However some of these languages are true icon-based programming systems that limit the amount of text-based programming required. Research on iconic programming languages began over 20

DOI : 10.5121/ijpla.2013.3101

years ago (5,6,7,8,9,10,11,12,13,14,15,16,17,18) but it made little change in the way the majority of programmers program today. This is mainly blamed on scalability (19, 46): icon-based programming was fine for small sized programs but for commercial sized software with large scale architecture and much more complex logic, the iconic programs became too difficult to understand and work with: program readability was the limiting issue (20). Iconic programming was therefore relegated to the introductory teaching of computer programming where only small programs are attempted (21,22,23,24,25,26,27,10,28,29,30,31,32) and also to specialist programming areas such as distributed process control (33), databases (34,35,36,37,38), image processing (39,40), testing (27,41), distributed systems (42), signal processing (43) and robotics (28,44) where only relatively small coding changes are made. The scalability problem (19) is that large iconic programs became too difficult to design and understand with large amounts of complex code represented as icons connected by lines in large 2D diagrams: the diagrams became too confusing with so many connecting lines. They also became too large - larger than the screen itself for viewing and understanding the overall program structure. There is also a re-usability problem with iconic programs (19,20): calling the same methods with parameter passing in many different parts of the code increases the number of wires (connecting lines) in the iconic diagrams and increases the complexity of the program's visual appearance. Thus the problems of traditional iconic programming systems: scalability and re-usability are both related to readability (20) and have created doubt (5, 47) about the future of iconic programming and resulted in a drop off in papers published in this area: papers discussing new or experimental iconic programming languages have been hard to find in the last 15 years [45, 47].

However, if connections and layout flexibility cause the visual confusion that effectively limits the size of the application that these languages can manage, then we need to consider connectionless iconic programming which displays icons but no connecting lines between them or between icons and data items and yet still shows control flow. IL8 described in this paper, is an example of the possibilities for making connectionless iconic programming languages and this paper shows that it could be just as powerful as text-based languages for general purpose problem solving without limitation on program size. With the type of iconic programming described in this paper, the scalability and readability problems of the past appear to no longer occur. The question of whether this experimental connectionless iconic programming system makes programming easier to learn, easier to do and quicker to get to bug-free results than traditional text-based programming, suggestive though it may seem to be, is relegated to future research. (The question of whether or to what extent VPLs provide cognitive benefits to programmers was raise by Whitley et al in [46].)

Version 8 of this experimental language is under development after a thorough Cognitive Dimensions framework analysis on the usability of version 7 (IL7) [48]. The restrictions of the language impose conditions on how the programs are constructed and these will be discussed in this paper. The purpose of this paper is to show that it is possible to build a programming language for general purpose programming and problem solving of any size in which no textual coding is required and the program appears in a fixed and readable layout of icons without distracting connecting lines. Other issues such as user testing of the language or of comparative user testing between IL8 and other VPLs or standard HLLs are not addressed in this paper. Some other issues such as software design and development processes resulting from the nature of IL8 are however explored.

Iconic Language IDE Design

This paper describes IL8, the connectionless Iconic Language version 8 and its IDE (Interactive Development Environment) for creating "iPrograms", programs made with icons rather than with text. The IL8 IDE displays a main window divided into three vertical areas. (See Figure 1.) The leftmost area is called the Data Area and is for representing the data that the the user will create for his program. The middle area is called the Method Area and is where the user places the icons that he wants in his program and rearranges their order suitably. The rightmost area is called the Palette Area and is the menu of icons that the user can select from to place into his program in the Method Area.

IL8 allows the user to construct modules and in each module to construct data and methods. There is no data in IL8 other than the data inside a module and there are no methods (programming code) in IL8 other than inside a module. Thus there is no global data and no local data for methods and no stand-alone methods external to modules. The module encapsulates its data so that methods from other modules cannot access it: a method can only access the data in the module to which it belongs.

IL8 divides data into either numeric data (“variables”) or string data (“strings”). Numeric data includes integers, floating point and Boolean values where, as in C, a zero value means false and a non-zero value means true. Each data variable is represented as a thin but wide rectangle with a vertical division in the middle. The left half of the rectangle displays the data variable's name (the Name section) and the right half of the rectangle displays its value (the Value section). Numeric data is displayed as a stack of such rectangles (ie a 2-column table) and the string data is displayed as a similar stack of rectangles underneath the numeric data table in the Data Area. By a main menu item or pressing the 'v' key or an on-screen button the user can add another numeric variable into his program in the Data Area: it extends the numeric table by one more row at the bottom. The numeric variables form a neat vertical stack in the top half of the left hand side of the main IDE window and display default names v1, v2, v3 ... all with the default value of 0. By a main menu item or by pressing the 's' key or an on-screen button the user can add another string variable into his program in the Data Area: it extends the string table by one more row at the bottom. The string variables form a neat vertical stack in the bottom half of the left hand side of the main IDE window and optionally display their default names s1, s2, s3 ... all with the default value of the empty string. The user can select any cell in either of these two tables and type to change the data variable's name or value. (Quote signs are not used in the string Value sections.) If the variable or string table becomes too long for their IDE display areas then up and down scrolling is enable for that table.

The IDE has enough vertical space to display 12 variables and 12 strings in the Data Area, and 10 icons vertically in the Method Area or Palette Area but each of the three areas has vertical scrolling when needed. Initially the Palette Area displays only 10 selectable in-built (system) icons in a vertical column. These icons are called the Basic System Icons for program building. The function names for each icon are listed below:

Basic.NewVariable/Add/Subtract/Multiply/Divide/Copy/Input/Output/Import/Export

If the user moves the mouse over any one of the displayed icons, its name is displayed and its function is explained in words on the window's status bar at the bottom of the IDE main window. As soon as the user clicks on one of the Palette Area icons a copy of it appears in the Method

Area - at the top of the Method Area if no icon had been previously selected, otherwise at the bottom of the vertical stack of icons previously selected. Thus repeated selection of icons from the Palette Area results in a vertical column or stack of juxtaposed icons in the Method Area. The choice and sequence of these icons forms the program that the user is creating. At any time the user can click on any icon in the Method Area and then move it up or down in the stack by using the arrow keys, or else delete it from the stack by pressing the Delete key. Also when the user clicks on an icon in the Method stack an Information Box opens up next to that icon. This Information Box is a rectangle that displays the name of the icon and its list of parameters. Each parameter is labelled as either literal, input, output or string and is a rectangle the same size as a variable rectangle Value or Name section in the Data Area. Initially the input and output parameter rectangles are blank. The user must assign variables from the Data Area to each input or output parameter in the Information Box for each icon in his program. To assign a variable to an input or output parameter in the Information Box the user first clicks on a variable in the Data Area (clicking either inside its Name section or its Value section) and secondly clicks an input or output parameter box in the icon's Information Box. Once he has done this the variable's Name appears in the chosen parameter box. If a mistake is made the user can just repeat this parameter assignment process to change the parameter settings. In the case of a literal parameter, the parameter box initially contains 0 and clicking the Value part of a numeric data item and then a literal parameter box (labelled as a literal) inside the Information Box, will copy the selected value to the literal parameter. Alternatively the user may type in a value in the literal parameter box in the same way that he typed in values within the Value section of the variables in the Data Area. Clicking on a literal parameter box causes the text entry cursor to appear and the parameter box background colour to change from white to yellow indicating to the user that a literal value must be typed in. In the case of a string parameter, the parameter box is initially blank (indicating the empty string), and the user can click a string value in the Data Area and then the string parameter box to copy the selected string to the string parameter, or else he can type in a new string into the string parameter box directly. When all the parameters in the Information Box are set in this way, the Information Box background colour changes from cyan to green to indicate that it is now executable. Once the user selects another icon in the Method Area, the Information Box of the previous icon disappears and the Information Box of the selected icon is displayed. All parameters for all icons must be set before the program can be executed, however the IL8 IDE also allows a single icon or a sub-sequence of icons in the current method to be executed on their own i.e. IL8 supports “slicing”. To execute the program the user selects the Execute main menu item or clicks an on-screen button (which has already changed colour from cyan to green to indicate that all icons in the method are executable, and therefore the method itself, is now executable), and then the IL8 IDE will run through the program that the user has made in the Method Area making changes to the variables in the Data Area and displaying those changes in the Data Area.

The IDE also displays an IO window from which the program can read in values typed by the user or display output values from the program during execution. (See Figure 1.) If a graphics system module is added to IL8 then graphics output would also appear in this second window.

If more icons have been appended to the current method in the Method Area than can be displayed at the one time then the icon stack becomes scrollable. On-screen up and down arrow buttons become highlighted (a change of colour from cyan to green) as appropriate and by clicking them the user can scroll his list of icons up or down one icon at a time in order to review or edit the entire method.

Initially the Basic icons are displayed in the Palette Area as listed above, however by selecting main menu items or clicking on-screen left and right arrow buttons, the user can change the System Icons displayed in the Palette Area. The system Basic module's icons are replaced in cyclical order by the following modules of in-built System Icons:

Control.IfNotZero/IfZero/IfPositive/IfNegative/WhileNotZero/WhileZero/WhilePositive/WhileNegative/For/Repeat

Maths.SetVariable/SquareRoot/RaisedToThePowerOf/NthRoot/NaturalLogarithm/LogarithmToTheBase10/Exponential/10ToThePower/Round/IntegerPart

Trig.Cos/Sin/Tan/ACos/ASin/ATan/Reciprocal/AbsoluteValue/FractionalPart/PI

String.NewString/StringLength/SubString/Concat/Pos/ASCII/Char/Insert/ReadLn/CopyString

Format.PrintNoLF/PrintFormatted/PrintStringNoLF/PrintStringFormatted/PrintChar/PrintSpace/PrintTab/PrintLn/PrintMoney/NextColumn

Array.NewNumericArray/WriteNumericArray/ReadNumericArray/ZeroNumericArray/NewStringArray/WriteStringArray/ReadStringArray/EmptyStringArray/NumericArraySize/StringArraySize

Boolean.AND/OR/NOT/NAND/NOR/XOR/SetBit/GetBit/BitwiseAND/BitwiseOR

Comparison.IsZero/NotZero/IsGreaterThanZero/NotGreaterThanZero/IsLessThanZero/NotLessThanZero/IsGreaterThanOrEqualToZero/NotGreaterThanOrEqualToZero/IsLessThanOrEqualToZero/NotLessThanOrEqualToZero

Memory.Peek/Poke/AddressOf/NrVariables/NrStrings/NrDynamicVariables/NrDynamicStrings/MethodNr/ModuleNr/VersionNr

The user can create more than one method sharing the same data of the Data Area. This is done by selecting the menu item New Method or clicking an on-screen button which then clears the Method Area ready for the user to create a new method. The user can move backwards or forwards through the list of created methods for the given data to review and re-edit methods by means of menu choices or on-screen arrow buttons. Each method also has an assigned default name that the user can modify. He can also associate a user-chosen icon with each method. The method icon is shown below the current method’s column of icons in the Method Area. (See Figure 1.) Clicking this icon allows the user to change the picture used for that icon.

The Data Area and its list of methods are together called a module in IL8 and modules can be saved to file in a text file format called ILM and reloaded at any time for further development work. This format for storage of modules is in the traditional form of a normal programming HLL source file yet it never needs to be viewed by the IL8 user and it will be described later. The user can change the default module name to any suitable valid name. The module name is used for the ILM file name: module N will be saved in a single module ILM file as file N.ILM. Modules can also have icons which are used at the Operating System level for selecting iPrograms for display (as in Figure 2) or viewing under the User Palette of the Palette Area in the IL8 IDE. The module icon image files have the same file name as the module name (but with the JPG file extension)

such as N.jpg where the module name is N. The module icon is displayed in the IL8 IDE underneath the strings table in the Data Area as shown in Figure 1. By clicking this icon, the user can change its picture.

In IL8, icons without images (their default initial state) are simply displayed as squares the same size as an icon with their text name (default or user-selected) written inside the square (or truncated to fit). At any time the user can choose, create or edit the icon picture to assign to any icon. Any picture can be used for an IL8 icon and it will be re-sized to the IL8 icon size. For simplicity, the file name for the icon for a method is the method's name, so the chosen icon picture for method m say, is copied into the local working directory with the file name N.m.JPG (where N is its module name). IL8 decides whether an icon has an image or not if the file N.m.jpg exists or not in the working directory.

Module names and method names should have no spaces, braces or full stops and be unique. (Method names need to be unique only up to the combination N.m.) Moving the mouse over any icon displayed in the Palette Area or Method Area causes the name and description of that icon to appear in the status bar at the bottom of the IL8 IDE main window. Right clicking the status bar allows the user to edit the text string description for the current method. The number of System Icon modules (and therefore likewise the potential capability of user-developed iPrograms) built into IL8 could be increased indefinitely subject only to computer memory limitations. Currently IL8 has 100 system icons which has reached the “critical mass” where the quantity of features or system functions is deemed sufficient to make the system usable for practical problems.

The Look of a Connectionless Iconic Program

Apart from saving modules into a source language text file format, another menu item allows the user to compile the module to a semi-executable form called ILP. (It is semi-executable because its methods cannot be executed until binding is done i.e. until their parameter settings are finalized.) Files of type ILP can be loaded into the Palette Area of the IL8 IDE where each method of the module is displayed as a different selectable icon. These user-defined icons are shown in a (vertical) column to the right of the System Icons column in the Palette Area of the IL8 IDE main form. (See Figure 1.) So in general there are two columns of icons for the user to select from in the Palette Area. The left column is called the System Palette and the right column is called the User Palette. Under each palette there is an icon for the module that the icons of that palette belong to. Additionally there are four arrow buttons surrounding each of these two icons for scrolling the respective icon column up or down when possible and for changing left and right between the available loaded modules. The user can now either select System Icons or User-defined Icons for building his next method. Thus the icons in a new method are a sequence of selections from either the System Icons (Basic, Control, Maths, Trig, String, Format, Array, Boolean, Comparison and Memory) or else they are selected from previously defined User-defined Icons that have been loaded to the Palette Area from ILP files.

Figure 2 shows a typical Connectionless Iconic Program in figurative form. It displays several rows of icons grouped inside rectangles. (A method in the IL8 IDE displays as a vertical linear column of juxtaposed icons with the method icon below the column for convenience in program development and they are executed in the order from top to bottom but it makes no essential difference if methods are displayed vertically or horizontally as with Egyptian hieroglyphics.) Each row starts at the left end with a user-defined icon image then a gap followed by a horizontal string of icons placed next to each other. The left-most icon is called the "head" icon and is the

pictorial "name" of the method which is defined by the string of icons following it. The icons following the head icon in a row are called the tail icons. Each row is a method whose icons are executed from left to right and lower rows may only call upper rows which are inside other rectangles. The bottom row is the “main routine”. For example, an iProgram to print a table of values of the quadratic function y=ax2+bx+c for x=x1 to x=x2 (where x1 and x2 are integers and x1 < x2) requires at minimum only two rows of icons of 12 and 24 icons respectively corresponding to methods from two different modules. Figure 2 shows a solution to this problem with 9 rows of icons with each containing between 2 and 9 icons. The bottom row is the main routine (where execution starts) and it calls 6 method rows in the class above it. (The execution of this iProgram is shown in Figure 1.)

As Figure 2 illustrates, an iProgram does not have to consist of only one module N and one method of that module. At any time when using the IL8 IDE to develop the iProgram the user can use a main menu item to load in a different module N2 say from the file N2.ILP. The methods from module N2 are then displayed in the User Palette (the right hand column of the Palette Area), and they are now available as icons to be chosen for use in the current method that the user is developing in the Method Area on screen. All the ILP files that have been previously loaded into the IL8 IDE in this session are still in memory, and by use of on-screen arrow keys, one can cycle through the different modules of user-defined icons shown in the User Palette to select any desired icon to add into the current method in the same way that on-screen arrows allow the user to cycle through the 10 System Icon modules of the System Palette to locate the next icon needed for the current method. All methods of an ILP module are shown in one vertical column of user-icons (which is scrollable up and down if the module contains more than 10 methods). If the user selects the same ILP file for loading into the IL8 IDE again then two (or more) instances of the module occur in the list of user-defined modules for cycling through and choosing from. This corresponds in ordinary HLL programming to declaring multiple objects of the same class for use in the one program. Note that this design means that the current module under development can call any method in any previously created module, but not the methods or modules that those methods actually use. The methods of the current module can call fresh copies of those secondary modules but not the same ones that were actually called by those methods loaded in as ILP files. This is a Software Engineering feature of IL8 that calls in the hierarchy call tree go back one level only.

An iProgram display program called ILV can display any ILM file (single module or multi-module). Single module ILM files could be converted to a multi-module ILM file in order to display the whole iProgram in one view. However, ILV can load multiple single module ILM files to give the same resulting view. Note that ILV shows methods grouped into their module rectangles in calling order (i.e. load order) from top to bottom in the format shown in Figure 2. In displaying the whole iProgram, all the tail icons of all the rows in the topmost module rectangle can consist solely of System Icons. These represent new methods that can be called in the user’s iProgram. The tail icons in the second or later module rectangles can again consist of IL8 System Icons but can now also include head icons of higher module rectangles symbolizing calls to those methods. So as we go down the rows, any string of tail icons can consist of System Icons and any User-defined Icons (head icons) which are defined by a string of tail icons in module rectangles above it. The name of a called method such as N2.m is contained in the ILM code of the calling method and ensures that the viewer program knows the right module method icon to display. In ILV, modules are displayed as wide rectangles with the module icon on the left end of the upper edge. The methods of the module are displayed as rows of icons inside the module rectaangle. Moving the mouse over the module icon will display the name of the module in the status bar of

the ILV window. Moving the mouse over any method head icon will display the name of that method and its description in the status bar of the ILV window. When the user moves his mouse pointer over any method tail icon in this view the name of the icon and its parameters will be displayed in the window's status bar (i.e. the bottom line of the ILV window). If the user clicks a User-defined Icon then that icon and all icons like it will be highlighted and the row above where that icon is the head icon will be also highlighted. As part of effective viewing of large programs, the viewer program can shrink the icons for a larger view to fit the viewer window and it also provides scroll bars when needed. This view of the iconic program is high level, attractive and uncluttered allowing readability for any level of code complexity.

Selection and Repetition

The design of a connectionless iconic program has been described above for sequence statements. Selection and repetition statements in computer languages have caused large programs to look complex when represented visually whether by flow charts, flow graphs or iconic programming. Connection lines are used to show the flow of control within a program and in standard HLLs even with Structured Programming this can soon become complex. IL8 as a Connectionless Iconic Programming language takes a considerably simpler approach to the visual layout of the program logic. The idea is to use one icon for the selection consequence or loop body. For example with an If statement such as if x > 0 then A, B, this is instanciated as 3 juxtaposed icons represented in IL8 as IfPositive(x),A,B. The first icon is the IfPositive icon with x as the input parameter. (This icon comes from the Control module of System Icons.) The second icon is the icon for action A and the third icon is the icon for action B. If it happens during execution that x ≤ 0 then the second action A is skipped and only B is performed otherwise A is performed and then action B. A similar process occurs for repetition statements. Consider using the Repeat icon as in Repeat(n), A, B. (The Repeat icon also comes from the Control module of System Icons.) Its parameter n is the number of times to repeat the next action A before continuing to action B. The execution of the Repeat icon checks whether n > 0, and if so decrements n and sets a stack-based repeating flag so that after the next action (A) is done the execution routine sees the repeating flag and puts the program counter back to the start of the Repeat icon again. Once n is no longer positive, the repeating flag is cleared and action A is skipped so that the program counter moves on to action B. (The repeat flagged gets stacked to allow for nesting.)

The Control System Icons also contain a For (i,istart,iend,istep) icon, other If condition icons and many While condition icons all acting only on the next action. All of the Control system icon actions expect a following action and an iProgram will not run until the user has added a following icon. There is no Else icon or clause in the list so that to achieve the equivalent of If x = 0 then A Else B the programmer must use IfZero(x), A, IfNotZero(x), B which thereby requires 4 icons. (It is however possible for action A to change x such that both actions A and B will be performed but this is the exceptional case and here it is assumed that the user does not want that and that x is not an output parameter of action A.) This approach also forces the programmer to write modular code when the consequence of an if statement or a loop body is bigger than a single System Icon: he has to create a separate module in IL8 with the consequence of the if or the loop body (action "Ä" above) as a method to call. In a previous version every Control icon had to be followed by a non-Control icon so to implement selection based on compound decisions one would use a combination of modularization and nesting over modules. For example for the conjunction (x > 0) and (y < 0) nesting is used as in IfPositive(x),A where A = IfNegative(y),B. For the disjunction (x > 0) or (y < 0) one would use a sequence of icons with the same consequence icon such as IfPositive(x),A,IfNegative(x) B where B = IfNegative(y),A. (Again A

will be executed only once unless it changes x from positive to negative which is regarded as the exception case here that the IL8 user would not choose.) Nested If statements and nested loop statements could not be made without modularization of the code like this so that the consequence of each decision is a single icon. The current version (IL8) however allows methods to contain more than one Control icon in sequence and the last Control icon must be followed by a non-Control icon. This means that IfPositive(x), IfNegative(y), A is a possible way for implementing the conjunction. Likewise nested For loops can be implemented as a two Control.For icons followed by the loop body icon. An easier way to deal with compound decisions however is to use the system modules provided in version 8 called Boolean and Comparison.

Each of the system functions in the Boolean and Comparison modules return 0 for false or 1 for true. Each of the Boolean system module methods take three arguments except NOT which takes two. Each of the Comparison module methods take two arguments. The user can then combine these with the IfZero, IfNotZero, WhileZero and WhileNotZero Control icons for general decision statements and loops and again he can choose whether a decision value should change between the if and else clauses or not. Further useful system modules for version 8 would be a module of more string functions, a module for file I/O and a module of Operating System functions.

Equivalent Object-Oriented Coding Language

The IL8 IDE allows an iProgram to be saved in ILM format, ILP format or ILX format. These are all text files (for portability across a variety of computers). The IL8 iPrograms are stored as source code files in single module ILM format which can be loaded, edited and saved. When the IL8 IDE saves a module in ILP format however all its methods are compiled to a non-editable semi-executable byte code format and if saved in ILX they are compiled to non-editable executable form. ILX files are in opcode form and these files can be dragged and dropped onto the ILX application which then executes the ILX file directly as a stand-alone executable, that is it does not require the IL8 IDE to execute the ILX iProgram. The Java version of ILX makes IL8 iPrograms platform and OS version independent. Of course the ILX applications that do this are created from a copy of the program execution code inside the IL8 IDE.

The ILM file type stores a module in a traditional neatly-printed HLL source code format structured as follows:

```
[Uses: <ILM-file>]
Module: <module-name>
 Variables: <number-of-numeric-variables>
   <variable-name> = <initial-value>
   {repeated for each numeric variable in the module}
 Strings: <number-of-string-variables>
   <string-name> = <double-quoted-initial-value>
   {repeated for each string in the module}
Method: <method-name>
 Description: <method-description-string-which-appears-in-the-status bar>
   <function-called> <space-separated-function-parameter-list>
   {repeated for each action icon ie function in the method}
{repeated for each method in the module}
```

This shows that the ILM file format contains only 6 keywords, each ending in a colon. An ILM file can have zero or more of the Uses: clauses and one or more of the Module: clauses. IL8 outputs single module ILM files with a Uses: clause for every ILP file loaded into IL8 in load order. The ILP files originate from other similar single module ILM files (via the IL8 menu item called “Save for Palette”). The list of Uses: clauses at the start of an ILM file show what other methods the methods of the module may call. If all the Uses: clauses are replaced by the corresponding single module ILM files and no Uses: clauses are left in the ILM file then the resulting file is the multi module ILM file of the whole iProgram which can be used in ILV for viewing the whole iProgram code in one view.

From the HLL equivalent of IL8 iconic programming, the multi module ILM file, we can now see the nature and capability of this style of iconic programming and compare it with standard HLLs. The language has objects but no explicit inheritance or sub-classing and no dynamic allocation of objects. The objects can have private data only: the data can only be accessed by that object's own methods. However the methods can export and import data via the system Basic module icons Import and Export so the language can have the equivalent of getters and setters for external indirect access to a class's private data. There is likewise normal parameter passing between methods in different modules by means of the Basic.Import and Basic.Export system methods. Methods in the language have no local data but can create dynamic data for the class with the system module Basic's NewVariable method for dynamically creating a variable and the system String module’s NewString method for dynamically creating a string. Variable and string arrays are also dynamically created. These dynamic variables cannot be deleted by the user and are only cleared away from memory when the program execution completes. This lack of a delete function for dynamic data is regarded as a safe-guard against faulty programming (and assumes that memory is cheap and plentiful). Multiple objects of the same class are possible in an iProgram by loading the same module many times. In the ILM file, ILP objects loaded into the iProgram automatically get named N[1], N[2], etc where N is the name of the module that was loaded in multiple times into the IDE. Access to method m in these different objects is indicated in the ILM file as N[1].m, N[2].m etc where N.m and N[1].m are identical. Thus N[1].m and N[2].m are calls to different but identical methods that only differ in using different private class data. Methods in the language can only call system icons or methods from previously created and loaded modules. Thus methods of a class (i.e.module) cannot call other methods of the same class (unless they are in a different object) and in particular they cannot call themselves. Thus no recursion is provided and users must therefore implement code in the equivalent iteration form. All arrays of numeric variables or of strings are dynamically created and handled by the system methods in the system module called Array. Like the numeric and string dynamically allocated variables, the arrays cannot be deleted, and are automatically deleted only on program termination. These arrays also cannot be ostensibly reduced or increased in size once created. If a new size for the array is required then a new array can be dynamically allocated for the new size (subject to the limits of available memory). (Knowing the internal memory data structure for arrays one can however use system functions such as NewVariable, Peek and Poke to change array sizes.) Similar to some other modern languages (eg Java and C#) there are no pointer variables to deal with explicitly. However one can access the address of a variable or array to read or write directly directly to memory using the Memory system module. In a sense IL8 also has object arrays since the IL8 IDE user can load a class N from N.ILP multiple times and the call to method m for the ith object is of the form N[i].m(p1 p2 ... pn) in the ILM code where (p1 p2 .. pn) is the parameter list for method m in module N and i is the object index. Internally, all the private data items are replicated as class contexts to the size of the array. However currently no mechanism is provided in the IL8 IDE to allow the user to change the index i during execution: all array object call

indices are set at design time when the user chose which icon of the User Defined icons for which loaded object of class N to call and so cannot be changed during execution. Also IL8 has no System Icon for dynamically creating object arrays. Like arrays of numeric values and arrays of strings, the object array size is fixed in size and persists for the execution of the whole iProgram. There can be only one such array for each class and they do not need to be contiguous in the class context sequence in memory.

The implementation of IL8 has proven that the concepts described for IL8 are achievable: an iconic visual programming system can be made that displays applications compactly in a control flow view without connecting lines in a readable manner that can be scaled to a full view of all the code of the application or zoomed down to view just one method in the application. Effectively in IL8 the text symbols of an HLL have been replaced with more general iconic pictures and just as one can have a source code text file of any size so also the iconic view of the program as rows of icons grouped into module rectangles can be of any scrollable size and yet still readable in the same way that a text file is readable to any reader who is familiar with the text symbols. The visual of the iProgram does not increase in complexity or become confusing for larger and larger iPrograms any more than the complexity increases for larger and larger text-based program source files. There is no limit to the number of modules, methods, variables or strings in an IL8 iProgram apart from memory limitations in the same way that HLL source code files can be of any size. It could be argued by software engineers that a recursion-less, pointer-less, one-pass function defining language with no deletion of dynamic data and which enforces modularization of code assists software development by reducing a lot of potential for typing errors and program logic errors that beginners, dabblers or even the more experienced programmers are otherwise prone to make and yet is still a high level programming language capable of solving the same problems that other HLLs can solve. The bottom-up implementation process enforced by IL8 where a method can only call other methods in previously constructed modules is a restriction in the IL8 system that Software Engineering principles suggest would curb novice programming errors. As suggestive as this may seem, whether or not IL8 is really conducive to good software engineered applications, is easy to use, easy to learn and quick to develop bug-free code with, is an issue for future research in user testing and user comparative testing. Nevertheless we can make observations about scalability of the language here.

Considering the nature of ILM and the software development methodology enforced by the nature of IL8 and its IDE, we can say that IL8 programming is equivalent to C++ programming in the following style. Firstly there is only one C++ source code file (which we will call app.cpp here). In app.cpp there are no #include directives. This file consists of the declarations of public classes with one object declared on each class declaration followed by the main routine main(). So module N will be represented as class CN {...} N; so that it is instanciated as object N in C++. If module N is used twice then the second occurrence would be represented in C++ by class CN2 {...} N2; etc. The order of the object declarations in app.cpp sets which methods can call which other methods, except that for C++ we must add the restriction that methods cannot call any method in the same class. There are no stand-alone functions in app.cpp except main() itself. The main routine contains no code except a call to a method without parameters in the lowest class declaration in the app.cpp file. There are no global variables in app.cpp other than the globally declared objects. There are no public or protected variables in each class: all class variables are private. There are no local variables in any method (and no local variables in main()). All class methods are public. Furthermore no class can inherit data or code from other classes - the programmer must write out each class declaration in full himself. The only statements allowed in class methods are C++ control structures and calls. Calls can be to functions provided in a System

Library - which can be as big as you like and have as many functions as you like - or calls to methods in other classes. This last restriction eliminates the possibility of recursion so that problems have to be equivalently solved using iteration. C++ control structures may be used but not with curly brackets. This means that if statements control only one statement or else call a method from a different class. The usual C++ parameter passing can be used. Numeric and string data can be dynamically created in the class private data areas but delete may not be used. The only arrays possible are dynamically allocated arrays. Subject to these restrictions on C++ programming, can we still solve general problems of any size? The C++ programmer will answer yes it is still above critical mass, though it may take some thinking to produce a software design matching the rules of this disciplined approach to C++ programming. Any needed #include directives can be placed in the System Library and the programmer can call system functions that themselves make use of the included libraries such as math and STL. Confining the application to a single C++ file is okay and doesn't limit scalability in theory. The C++ programmer can get by without stand-alone functions or global data or local data and making things public and private means being careful about how they are accessed in the program but doesn't affect scalability. Limiting control structures to a single statement only means a redesign by adding another class. Iteration is often a more memory efficient way of solving problems than recursion. Without inheritance the coder may have to copy and paste some code which only means more coding effort and doesn't affect scalability. Disallowing the C++ delete keyword means being careful to avoid memory leaks and being careful that memory allocations are efficiently used. It also presupposes that memory size is not a problem. It is possible to write a translator program that translates any multi-module ILM files or indeed many separate single-module ILM files to a C++ program source file called app.cpp in accordance with these rules which could then be compiled and executed. From this analysis, none of these programming style restrictions on C++ affect scalability and it is concluded therefore that IL8 is scalable to any size programming problem (subject only to the presumption of available memory).

IL8 can be extended to make Windows-type applications in the same way as HLLs like C++ an C# are used for this purpose. What is required is a GUI Designer, an application to design the application's windows and screens and their detailed widget contents, sizes, positions and other properties. The GUI Designer can add new windows to an application design, place and size the gadgets on each window and set the gadget properties totally by mouse interaction with no text input except for the on-screen labels. To make the GUI Design into an application, appropriate event routines need to be created and code inserted into them. The desired event routines can be selected and created as empty routines in the GUI Designer and after this iconic code created in the IL8 IDE inserted into those event routine skeletons. Alternatively, ILM code could be translated to C++ code as discussed before, for insertion into the event routines. In this way windows-type programs for any-sized general purpose application could be built by clicks and other window-type application activities alone rather than by the current indirect method of using a text editor.

Conclusions

It has been shown that general purpose programming is achievable through purely icon-based interactions without the the need for any text code entry. At the same time the IL8 iconic program has a one-to-one relationship with its own native text coding language called ILM (the IL8 HLL). This means that the programming can be developed either through using a text editor to make ILM source text files or equally through an iconic interface where the programmer is simply selecting icons from a palette, putting them together to make new user-defined icons and

thereby creating the computer program in a bottom-up process. The difference is stark: in the traditional approach one uses a text editor with unlimited possibilities for coding errors (or else a syntax directed editor with reduced scope for errors) versus using the IL8 IDE where one is simply using the mouse to select icons, assign parameters and load and save files (with no syntax rules to remember or to abide by). With the IL8 IDE the keyboard is now only needed for naming modules, methods, variables and setting values and also for input during execution. The IL8 approach enforces a software engineering programming discipline on coding through an intuitive and friendly interface that includes no public variables in classes, no explicit pointers, no calling of functions not yet defined and thus also no recursion. Modular design and programming is enforced. This discipline enforced by the nature of programming in IL8, could well be a form of safe programming for users and its effectiveness is an area for future research. Since the IL8 native text coding language otherwise contains the same principal features of modern High Level Languages and can be as long a source file as desired, and equivalent code could be written in any typical HLL such as C++, C# or Java by sticking to the stated coding style restraints which do not restrict the range of problems that can be coded, nor affect scalability we conclude that general text code programming can be replaced by a truly visual Iconic Programming system that is scalable and similar in power to existing high level languages. The appearance of the iPrograms is also visually pleasing and there are no connection lines to cause visual clutter or confusion. Restricted versions of IL8 could be made limited to any particular programming domain (eg database or graphics programming) by suitably choosing the basic in-built icons of system functions appropriately. This could suit many small memory hand held mobile units or dedicated computer systems. Alternatively it can be general purpose by including System Icons for general programming along with specialist programming function icons for larger computers and systems. The underlying system library can be as small or as large as desired and contain as much complex coding algorithms as needed to reduce the load of programming effort from the IL8 programmer. It also is available for anyone to program with possibly minimal training though future research is needed on how much training users of this system would require. Nevertheless even if it is proved to be easy to learn and to develop programs with IL8, one still needs to be familiar with what System Icons are available and one has to plan and design the program to a different style, including what modules will be needed, what methods will be needed and what temporary and other variables will be needed and where they should best be placed, before implementation in the IL8 IDE.

References

1. Tanimoto, S.L. Representation and learnability in visual languages for Web-based interpersonal communication, Proceedings of the IEEE Symposium on Visual Languages, pp 2-10, 1997
2. Shneiderman B, Designing The User Interface: Strategies For Effective Human-Computer Interaction, Addison-Wesley, 1986 p198.
3. Shu, N. C.; Visual programming: Perspectives and approaches, IBM Systems Journal Vol 38, Issue 2.3, pp 199-221, 1999
4. Shu NC, Visual programming: Perspectives and approaches, IBM Systems Journal, vol 38, issue 2.3, 1999, pp 199-221.
5. Ichikawa T., Hirakawa M. Iconic Programming Where To Go? IEEE Software Vol 7 Issue 6 pp 63-68, Nov 1990.
6. Miyao, J.; Wakabayashi, S.; Yoshida, N.; Ohtahara, Y.; Visualized and modeless programming environment for form manipulation language, IEEE Workshop on Visual Languages, 99-104, 1989
7. Lieberman, H.; Dominoes and storyboards beyond 'icons on strings', Proceedings of the IEEE Workshop on Visual Languages pp 65-71, 1992
8. McIntyre, D.W.; Glinert, E.P.; Visual tools for generating iconic programming environments, IEEE Workshop on Visual Languages, pp 162 - 168, 1992.

9. Miller, E.; Kado, M.; Hirakawa, M.; Ichikawa, T.; HI-VISUAL as a user-customizable visual programming environment, Proceedings of the 11th IEEE International Symposium on Visual Languages, pp107-113, 1995.
10. Bagert, D.J.; Calloni, B.A.; Teaching programming concepts using an icon-based software design tool, IREEE Transactions on Education, Vol 42, Issue 4, 1999.
11. Bell, M.A.; Jackson, D.; CALVIN-courseware authoring using visual notation, Proceedings of the IEEE Symposium on Visual Languages, pp 225-230, 1993.
12. Hirakawa M, Tanaka M, Ichikawa T, An Iconic Programming System, HI-VISUAL, IEEE Transactions On Software Engineering. Vol. 16. No. IO . October 1990, pp 1178-1184.
13. Di Gesu V., Tegolo D. The iconic interface for the PIctorial C Language, Proceedings of the 1992 IEEE Workshop On Visual Languages, pp 119-124, September 1992.
14. Hirakawa M., Yoshimi M., Tanaka M., Ichikawa T. A Generic Model For Constructing Visual Programming Systems, IEEE pp 124-129, 1989.
15. Chang S K, Costagliola G, Orifice S, Polese G, Baker B R A Methodology for Iconic Language Design with Application to Augmentative Communication, IEEE pp 110-116, 1992
16. Orefice S, Polese G, Tucci M, Tortora G, Costagliola G, Chang S K, A 2D Interactive Parser for Iconic Languages, IEEE pp 207-213, 1992.
17. Yamaguchi S, Tanaka M, Morita S, Iconic System with Extension Mechanism, pp 1-7
Levialdi S, Cognition, Models & Metaphors, IEEE pp 69-79, 1990.
18. Birchman J J, Tanimoto S L, An Implementation of the VITA Visual Language on the NeXT Computer, IEEE, pp 177-183, 1992
19. Koike Y, Maeda Y, Koseki Y, Enhancing Iconic Program Reusability With Object Sharing, IEEE Symposium on Visual Languages 1996, pp 288-295.
20. Koike Y., Maeda Y., Koseki Y. Improving Readability Of Iconic Programs With Multiple View Object Representation, Proceedings of the 11th IEEE Symposium on Visual Languages pp 37-44, 1995
21. Rader, C.; Cherry, G.; Brand, C.; Repenning, A.; Lewis, C.; Designing mixed textual and iconic programming languages for novice users, Proceedings of IEEE Symposium on Visual Languages, pp 187-194, 1998.
22. Bagert, D.J.; Calloni, B.A.; Using an iconic design tool to teach the object-oriented paradigm, Teaching and Learning in an Era of Change Proceedings of the 27th Annual Frontiers in Education Conference, Vol 2, 1997.
23. Calloni, B.A.; Bagert, D.J.; Iconic Programming for teaching the first year programming sequence, Frontiers in Education Conference Proceedings, Volume: 1 , pp 2a5.10-2a5.13 1995.
24. Greyling, J.H.; Cilliers, C.B.; Calitz, A.P.; B#: The Development and Assessment of an Iconic Programming Tool for Novice Programmers, 7th International Conference on Information Technology Based Higher Education and Training pp 367-375, 2006.
25. Cockburn, A.; Bryant, A.; Do it this way: equal opportunity programming for kids, Proceedings of the 6th Australian Conference on Computer-Human Interaction, pp 246-251, 1996.
26. Cockburn, A.; Bryant, A.; Cleogo: collaborative and multi-metaphor programming for kids, Proceedings of the 3rd Asia Pacific Conference on Computer Human Interaction, pp 189-194, 1998.
27. O'Toole, K.; Salopek, P.; Next generation graphical development environment for test, IEEE Systems Readiness Technology Conference, pp 145-148, 1998.
28. Houxiang Zhang; Weining Zheng; Shengyong Chen; Jianwei Zhang; Wei Wang; Guanghua Zong; Flexible Education Robotic System for a Practical Course, IEEE International Conference on Integration Technology (ICIT), pp 691-696, 2007.
29. Carlisle MC, Raptor: a visual programming environment for teaching object-oriented programming, Journal of Computer Sciences in Colleges, vol 24, issue 4, April 2009. pp 275-281.
30. Calloni BA, Bagert DJ, Iconic Programming Proves Effective for TTeaching the First Programming Sequence, SIGCSE'97 Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education, March 1997, pp262-266.
31. CarlisleMC, Wilson TA, Humphries JW, Hadfield SM, RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving, SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education, February 2005, pp 176-180.

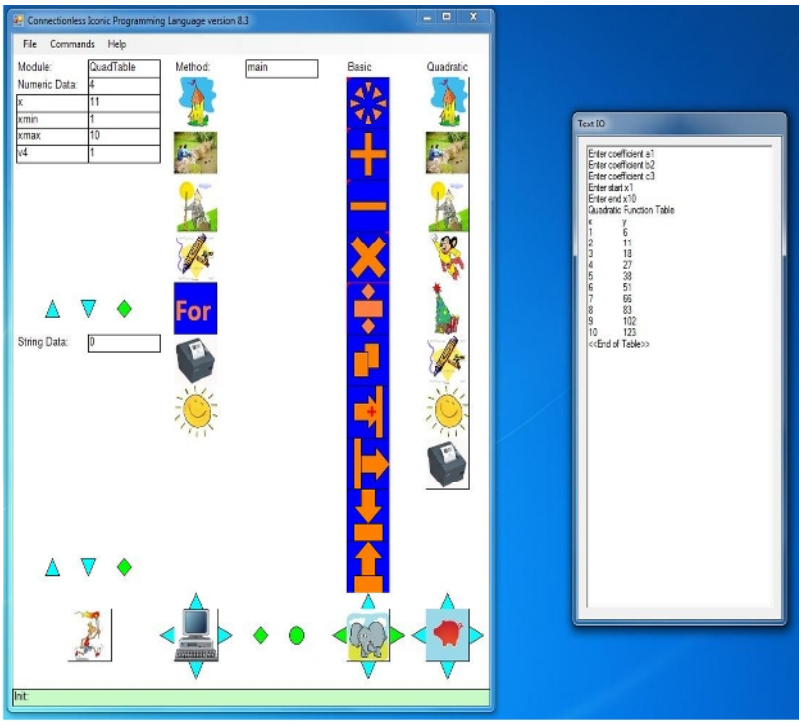
32. Carlisle MC, Wilson TA, Humphries JW, Hadfield SM, RAPTOR: Introducing Programming to Non-majors with Flowcharts, Journal of Computing Sciences in Colleges, vol 19, issue 4, April 2004, pp 52-60.
33. Coote, S.; Gallagher, J.; Mariani, J.; Rodden, T.; Scott, A.; Shepherd, D.; Graphical and iconic programming languages for distributed process control: an object oriented approach, IEEE Workshop on Visual Languages 1988, pp 183-190
34. Chang, SK Visual Reasoning for Information Retrieval from Very Large Databases, 1989 Workshop on Visual Languages, pp1-6, 1989.
35. Dudley, T A Visual Interface to a Conceptual Data Modelling Tool, 1989 Workshop on Visual Languages, pp30-37, 1989.
36. Echeverria LE, Pino JA An Intuitive Approach for the Expression of Boolean Queries, 1989 Workshop on Visual Languages, pp118-123, 1989.
37. Czejdo B, Embly D, Reddy V, Rusinkiewicz M A Visual Query Language for an ER Data Model, 1989 Workshop on Visual Languages, pp165-170, 1989.
38. Cinque L, Ferloni F, Levialdi S, Sargeni A X-VIQU: An Expert System for Visual Representation of Database Queries, 1989 Workshop on Visual Languages, pp183-188, 1989.

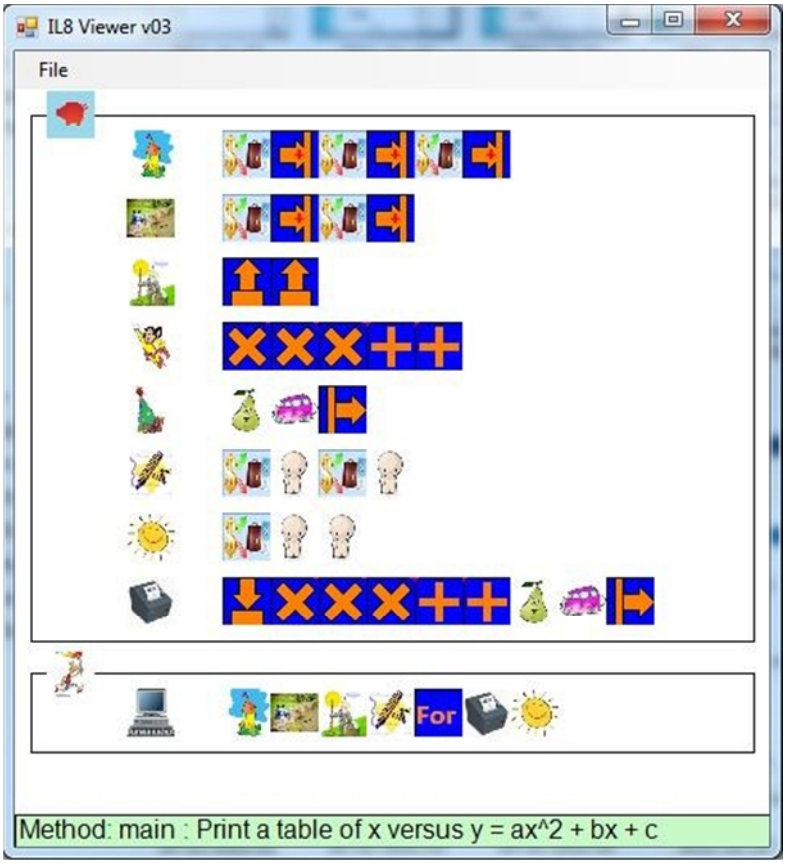
39. Hunt, N.; IDF: A graphical data flow programming language for image processing and computer vision, Conference Proceedings of the IEEE Conference on Systems, Man and Cybernetics, pp 351-360, 1990.
40. Tegolo, D.; Lenzitti, B.; Isgro, F.; Di Gesu, V.; Dynamic interface for machine vision systems, Proceedings of the 12th IAPR International Conference on Signal Processing, pp 323-326, Vol 3 1994.
41. Demazoin, P.; Barbier, J.Y.; Iconic test programming a tool for test program interoperability, Proceedings of the IEEE AUTOTESTCON Conference, pp 187-192, 2000.
42. Di Gesu, V.; Isgro, F.; Lenzitti, B.; Tegolo, D.; Visual dynamic environment for distributed systems, Proceedings of Computer Architectures for Machine Perception (CAMP), pp 359-366, 1995.
43. Quinchanegua, A.; Rodriguez, D.; A Kronecker DFT multibeamforming implementation approach, Proceedings of the 3rd International Symposium on Image and Signal Processing and Analysis, pp 1066-1071, Vol 2, 2003.
44. Microsoft Visual Programming Language (VPL) tutorials (2011) http://msdn.microsoft.com/en-us/library/bb483087.aspx
45. Boshernistan M.; Downes, M.; Visual Programming Languages: A Survey, UC Berkeley EECS Technical Report CSD-04-1368, 2004.
46. Whitely K.; Blackwell, A.; Visual Programming: The Outlook of Academia and Industry, Proceedings of the 7th Workshop on Empirical Studies of Programmers, pp 180-208, Oct 1997.
47. Ryder B, Soffa M, Burnett M, "The Impact of Software Engineering Research on Modern Programming Languages", ACM Transactions on Software Engineering, Vol 14, No 4, pp 431 477 October 2005.
48. Rankin J, "CD Analysis Of A Connectionless Iconic Programming Language", 2011 tba.

Figure Captions

Figure 1. The appearance of the IL8 IDE main window for creating and editing iPrograms and the text I/O window for displaying program execution.

Figure 2. The appearance of a typical iProgram (using ILV v03). It consists of several rows (or columns) of icons, each row having a head icon and a string of juxtaposed tail icons. The tail icons represent a method that can be called and the head icon is the icon used to call it. (In general the boxes are user-selected colourful icons each with their own functional meanings.)


Connectionless Iconic Programming Language version 8.3
File
Commands
Help
Module:
QuadTable
Numeric Data:
4
k
11
xmin
1
xmax
10
v4
1
Method:
main
Basic
Quadratic
For
String Data:
0
Init:
Text IO
Enter coefficient a1
Enter coefficient b2
Enter coefficient c3
Enter start x1
Enter end x10
Quadratic Function Table
x y
1 6
2 11
3 18
4 27
5 38
6 51
7 66
8 83
9 102
10 123
<<End of Table>>


IL8 Viewer v03
File
For
Method: main : Print a table of x versus y = ax^2 + bx + c

Appendix
The QuadTable.ilm file listing is as follows:
Uses: Quadratic

```
Module: QuadTable
Data: 4
  x = 0
  xmin = 0
  xmax = 0
  v4 = 1
Strings: 0

Method: main
Description: 43
Print a table of x versus y = ax^2 + bx + c
Icon: 30
C:\JR Work\Images\download.bmp
  Quadratic.Init
  Quadratic.InputRange
  Quadratic.GetRange    xmin    xmax
  Quadratic.TableHeading
  Control.For    x        xmin    xmax    v4
  Quadratic.PrintXY      x
  Quadratic.EndOfTable
```

The Quadratic.ilm file listing is as follows:

```
Module: Quadratic
Data: 9
  a = 0
  b = 0
  c = 0
  x = 0
  x1 = 0
  x2 = 0
  temp1 = 0
  temp2 = 0
  y = 0
Strings: 8
19
Enter coefficient a
19
Enter coefficient b
19
Enter coefficient c
13
Enter start x
11
Enter end x
```

```
24
Quadratic Function Table
3
x          y
16
<<End of Table>>

Method: Init
Description: 34
Inputs the coefficients a, b and c
Icon: 0
  Format.PrintStringNoLF        1
  Basic.Input     a
  Format.PrintStringNoLF        2
  Basic.Input     b
  Format.PrintStringNoLF        3
  Basic.Input     c

Method: InputRange
Description: 41
Input the table start and end values of x
Icon: 0
  Format.PrintStringNoLF        4
  Basic.Input     x1
  Format.PrintStringNoLF        5
  Basic.Input     x2

Method: GetRange
Description: 42
Return the table start and end values of x
Icon: 0
  Basic.Export    x1
  Basic.Export    x2

Method: ComputeY
Description: 31
Compute y = ax^2 + bx +c from x
Icon: 0
  Basic.Multiply          a       x       temp1
  Basic.Multiply          temp1   x       temp1
  Basic.Multiply          b       x       temp2
  Basic.Add       temp1   temp2   temp2
  Basic.Add       c       temp2   y

Method: PrintLine
Description: 42
Print one line of the table x and y values
Icon: 0
  Format.PrintNoLF      x
```

```
  Format.PrintTab
  Basic.Output   y

Method: TableHeading
Description: 23
Print the table heading
Icon: 0
  Format.PrintStringNoLF        6
  Format.PrintLn
  Format.PrintStringNoLF        7
  Format.PrintLn

Method: EndOfTable
Description: 30
Print the end of table message
Icon: 0
  Format.PrintStringNoLF        8
  Format.PrintLn
  Format.PrintLn

Method: PrintXY
Description: 57
Compute y from x and print a line of the table as x and y
Icon: 0
  Basic.Import   x
  Basic.Multiply        a       x       temp1
  Basic.Multiply        temp1   x       temp1
  Basic.Multiply        b       x       temp2
  Basic.Add     temp1   temp2   temp2
  Basic.Add     c       temp2   y
  Format.PrintNoLF      x
  Format.PrintTab
  Basic.Output   y
```