A Programming Language for Graphic User Interface Designs

Shachar Itzhaky and Mooly Sagiv

February 24, 2012

Abstract

This document describes a language that will be compiled in the workshop.

1 Introduction

Throughout this workshop, we will be dealing with the most common graphical interaction elements: *windows*. Application windows usually sport a hierarchical structure, where a window constitutes of several child windows, each of which may contain children of their own. Leaf windows (which do not have children) provide some primitive UI functionality, and are usually referred to as *widgets*.

For both ease of implementation and of use, windows were historically defined to be *boxes*: i.e., rectangular areas on the computer screen, where the sides are always parallel to the sides of the monitor. Thus we can uniquely identify a box with four scalar values: left, right, top, and bottom.

The mechanism through which an application determines the box coordinates and effectively splits the area assigned to it between windows is called *layout*. A good layout would efficiently and intuitively adapt to different monitors and window sizes. For example, if you resize the window of the application viewing this document, the area containing the text will most likely grow or shrink, but the window's toolbar, status-bar, and possibly side-bar will retain its size. Viewers with a side-bar usually allow the users to drag the borderline between the main text area and the side-bar, setting the amount of area allocated to each to their taste.

Layout is Achilles' Heel of most novice GUI programmers. Different GUI frameworks provide different ways for setting the layout, with varying degrees of control, but they require a lot of skill and a steep learning curve.

2 Motivation

In the above example we can see the face of Apple's iconic CEO, the late Steve Jobs, along with some trivial information about him. In order to convey this information properly for generations to come, some constraints must be applied to the layout of the window:

1. The size of the box containing the picture should be equal to the size of the image + a small constant for the width of the border on each of the four sides.
2. The image should be placed at the center of the picture box.
3. The border around the image should be painted green.
4. The left side of the picture box should touch the right side of the "Articles" list.
5. The left side of the "Profile" box should touch the right side of the picture box.
6. The upper side of the "Contributions" box should touch the bottom of both the picture box and the "Profile" box.
7. Both the "Profile" and "Contributions" boxes have a grey heading which goes all the way from left to right but only X pixels down; any box contents appear only below this heading.
8. The "Profile" box: texts item are arranged one below the other with horizontal lines above and below the company name. Company logo and

national flag appear below the text, aligned to the left and to the right of the box, respectively.

9. Contributions are depicted by graphical thumbnails with a caption attached. The caption appears below the thumbnail, horizontally centered. Several entries appearing in the "Contributions" box are arranged one next to the other, aligned vertically through the bottom of their thumbnails.

Maintaining the above requirements is a tedious programming task, and we cannot avoid a fair amount of arithmetic on our way to compute the coordinates of the widgets in this window. For example:

$$\begin{array}{rrcl} \textit{Let} & W \times H & = & \textit{size of the image} \\ & B & = & \textit{border width} \\ & X & = & \textit{height of grey headings} \end{array}$$

(1) Projects.*left* = PictureBox.*left* = Team.*right*
(2) PictureBox.*right* = PictureBox.*left* + $W + 2B$
PictureBox.*bottom* = PictureBox.*top* + $H + 2B$

Therefore, we design a language with three main parts ("varieties"): the first one deals with the arithmetic, the second one with the boxes that build up the layout, and the third one steers the flow of data between the various elements in the program.

3 Language Design

3.1 Arithmetic Variety

A small language of arithmetic expressions will serve as an accessory in building the UI description language.

Using this language we will be able to express linear equalities with arbitrarily many variables.

3.1.1 Building Blocks

Atoms. Two types of atomic *terms* exist:

- A numeric real literal (may or may not contain a decimal point). E.g. `6` or `102.5`.
- A variable identifier, which can be any (properly delimited) sequence of alphanumeric characters not beginning with a digit. An underscore (_) is also permitted. Delimiters include whitespaces (space, tab, new-line), parenthesis, operators, and relations (see below). E.g. `ay`, `v1`, and `r2_d2`.

Operators. To enforce linearity of the resulting expressions, this language has only 3 arithmetic operators with the following constraints:

- If e_1, e_2 are terms, then $e_1{+}e_2$ and $e_1{-}e_2$ are also terms.
- If e is a term and r is a numeric literal, then $r{*}e$ is also a term.

Operator precedence is as usual, and parenthesis may be used to enforce any other order.

Equations. If e_1, e_2 are terms, then $e_1{=}e_2$ is an *equation*.

The semantics of an equation is equivalent to its normal mathematical interpretation. Notice that all the equations are linear.

3.1.2 Programming Constructs

Subprograms. A subprogram definition comprises of a header and a body, where the body is a new-line-delimited sequence of statements enclosed in curly brackets ({ }), and the header has the form:

$$\langle name \rangle(\langle v_1 \rangle, \langle v_2 \rangle, \cdots, \langle v_n \rangle)$$

Where v_i are variable atoms. They are called the function's *inputs*. All other variables referred to in the body of f are called the function's *outputs*.

The semantics of a subprogram is a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$, such that f maps actual values for the inputs v_i $(i = 1, \ldots, n)$ to a tuple of values for f's outputs such that these values satisfy the constraints expressed by the statements in the function's body.

This is more easily demonstrated than defined, so:

```
f(b) {
 a+b=10
 b=8-c
}
```

The semantics of `f` is $f\colon \mathbb{R} \to \mathbb{R}^2$ such that f maps any value b to corresponding values $\langle a, c \rangle$ that satisfy the two equalities $a + b = 10 \wedge b = 8 - c$.

$$f(b) = \langle 10 - b, 8 - b \rangle$$

Note. When the order of variables is not specified by the program, such as in the return value of a function, the order in which they occur first in the body is used.

3.1.3 Limitations

As you may know, solving a general set of linear equations requires some knowledge of linear algebra. This knowledge, however, is *not* required for this assignment.

In order to simplify matters for the compiler, and to allow generation of efficient code, we set the some *semantic restriction* on the program. Consider again the function `f`:

```
f(b) {
 a+b=10
 b=8-c
}
```

The semantics of `f` can be obtained by the following reasoning steps:

1. First, consider the first equation $a + b = 10$. By elementary arithmetic, transform it to $a = b - 10$. Since the value of b is an input to the routine and therefore known at run-time, this equation can now be seen as an assignment.
2. Similarly, $b = 8 - c$ reduces to $c = 8 - b$, which can be seen as another assignment.

Here is a slightly more complicated example:

```
g(b) {
   a+b+c=10
   b=8-c
}
```

The code for `g` can be generated like this:

1. $b = 8 - c \Rightarrow c = 8 - b$
2. $a + b + c = 10 \Rightarrow a = 10 - b - c$

The second assignment is valid, since b is given as input and c is computed by a previous assignment, so its value is also known at run-time. Notice, however, that the order in which the statements are evaluated at run-time is *different* from the order in which they occur in the program.

What remains, however, is the following rule-of-thumb:

> We shall allow only functions such that, in the resolution of the equations for any single function, it suffices to only consider *one equation at a time*.

Of course, this is not true in general. For instance, the function $\widehat{\text{h}}$ does not fulfill the restriction:

```
h(b) {
   a+2*c=b
   2*a+c=b
}
```

because when only b is known, neither equation can be transformed into a valid assignment.
Note. This entire reasoning must be done *at compile time*. This includes issuing compiler errors if a function fails to satisfy the requirement. At runtime, the only operations that should ever be performed are arithmetic.

For Linear Algebra Lovers. If you are keen on mathematics and want to get a deeper understanding of the semantics defined above, read the following paragraphs. Otherwise, skip straight to 3.2.

The above ad-hoc restriction rule can be stated equivalently as:

> In any subprogram, the equalities expressed by the statements, viewed as a linear equation-set in the *output variables*, is represented by an *upper triangular matrix*.
> (up to row/column reordering)

This restriction saves the need for linear equation-set solving via triangulation (Gauss elimination, etc.). All the compiler has to do is find an ordering of the rows and columns of the matrix such that the resulting matrix is upper triangular; then, if the equation-set has a single solution, the values of the output variables can be computed, from right to left, by scanning the rows, bottom to top, assigning the values of the variables which have already been computed to get the value of one more variable.

E.g. the function `f` above induces the following matrix:

$$A = \left(\begin{array}{ccc|c} 1 & 1 & 0 & 10 \\ 0 & 1 & 1 & 8 \end{array}\right)$$

which, when omitting input variables and constants, boils down to:

$$\hat{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

To solve the equation-set, scan the matrix for rows where all the variables (non-zero cells) are known save one. In the case of `f`, both rows satisfy this. Then one additional unknown value can be computed from each of them.
For the second function, `g`, where the matrix is

$$\hat{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

two computation steps will be required:

1. At first only b is known, so the only solvable row is the second row, expressing $b = 8 - c$. Note that the left cell is 0, so this row is independent of a.

2. Once c is computed, the first row becomes solvable since the only unknown left is a.

3.2 Layout Variety

In this subsection we will design a declarative language for specifying layouts, and implement a compiler that generates a UI based on the layout specified. We follow the example of existing declarative standards, such as HTML5, CSS, and QML.

3.2.1 Syntax & Semantics

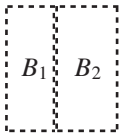
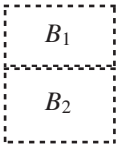
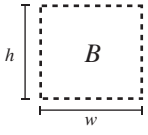
Box. Let $Box = \mathbb{R}^4$ be the *domain of boxes*. A box is given by a 4-tuple:

$$\begin{bmatrix} & top & \\ left & & right \\ & bottom & \end{bmatrix}$$

Combinator. A *box combinator* would be an operator on boxes. We will have two types:

- Unary – $c\colon Box \to Box$
- Binary – $c\colon Box^2 \to Box$

The core consists of two binary box combinators and one unary combinator:

Notation	Semantics	Depiction
$B_1 \vert B_2$	$B_1 = \begin{bmatrix} & \beta_1 & \\ \alpha_1 & & \alpha_2 \\ & \beta_2 & \end{bmatrix}$ $B_2 = \begin{bmatrix} & \beta_1 & \\ \alpha_2 & & \alpha_3 \\ & \beta_2 & \end{bmatrix}$ $B_1 \vert B_2 = \begin{bmatrix} & \beta_1 & \\ \alpha_1 & & \alpha_3 \\ & \beta_2 & \end{bmatrix}$	
$\frac{B_1}{B_2}$	$B_1 = \begin{bmatrix} & \beta_1 & \\ \alpha_1 & & \alpha_2 \\ & \beta_2 & \end{bmatrix}$ $B_2 = \begin{bmatrix} & \beta_2 & \\ \alpha_1 & & \alpha_2 \\ & \beta_3 & \end{bmatrix}$ $\frac{B_1}{B_2} = \begin{bmatrix} & \beta_1 & \\ \alpha_1 & & \alpha_2 \\ & \beta_3 & \end{bmatrix}$	
$B : w \times h$	$B = \begin{bmatrix} & \beta_1 & \\ \alpha_1 & & \alpha_1 + w \\ & \beta_1 + h & \end{bmatrix}$ $\Big(B : w \times h\Big) = B$	

Note that the dimensions operator does not change the value of the argument, it only enforces unification of any unspecified values to produce the desired width×height.

Coordinates. Some or all of *left*, *right*, *top*, and *bottom* may be unspecified. We will mark an unspecified value with '?'.

For example: $b = \langle 0, ?, 20, ? \rangle$ is a box where $left = 0$, $top = 20$, and *right* and *bottom* are unspecified.

Unspecified values are indispensable for composing windows using box combinators, since the combinators illustrated above introduce many repeating subexpressions: $B_1 \big| B_2$ is defined only if B_1's *top* and *bottom* are the same as B_2's, and $\frac{B_1}{B_2}$ is defined only if B_1's *left* and *right* are the same as B_2's. Thus if you want to avoid repeating the values you could specify a chain of windows like so:

$$b = \begin{bmatrix} & 0 & \\ 0 & & 20 \\ & 20 & \end{bmatrix} \Big| \begin{bmatrix} & ? & \\ ? & & 50 \\ & ? & \end{bmatrix} \Big| \begin{bmatrix} & ? & \\ ? & & 80 \\ & ? & \end{bmatrix} \Big| \begin{bmatrix} & ? & \\ ? & & 100 \\ & ? & \end{bmatrix}$$

resulting in the box:

$$b = \begin{bmatrix} & 0 & \\ 0 & & 100 \\ & 20 & \end{bmatrix}$$

and assigning the concrete values to the four sub-boxes:

$$b_1 = \begin{bmatrix} & 0 & \\ 0 & & 20 \\ & 20 & \end{bmatrix} \quad b_2 = \begin{bmatrix} & 0 & \\ 20 & & 50 \\ & 20 & \end{bmatrix} \quad b_3 = \begin{bmatrix} & 0 & \\ 50 & & 80 \\ & 20 & \end{bmatrix} \quad b_4 = \begin{bmatrix} & 0 & \\ 80 & & 100 \\ & 20 & \end{bmatrix}$$

Following is a short discussion of this issue from a programming-language point of view. If you are bored, tired, or short on time, feel free to skip to the next paragraph, "Top-level Window".

The process of determining unspecified values using specified ones and the operators' domains is known as *unification*; two values are *unifiable* if unknowns may be substituted in a manner that will assert the equalities required by the definition. Many forms of unification exist, but our case is very simple:

- '?' always unifies with '?', resulting in '?'.
- '?' unifies with any specified value, resulting in that value.
- Any two specified values unify iff they are equal, resulting in their common value.

Once unified, the result of unification becomes the new value of both expressions.

For greater ease, we will define a unary operator for setting the dimensions (width and height) of any box.

Top-Level Window. An application typically has a main window, and the operating system's *window manager* is responsible for allocating it. A program in our DSL defining one window will have the form:

Notation	Semantics	Depiction
main_window $\leftarrow B$	$B = \begin{bmatrix} & 0 & \\ 0 & & ? \\ & ? & \end{bmatrix}$ $\overline{\left(\text{main\_window} := B\right) = B}$	

The main window's box's *left* and *top* are unified with 0. All other coordinates, including those of child windows, are relative to the window's coordinate system. This operator, like the one before, does not affect the value but only induces further unifications.

Atoms. Concrete, "undividable" widgets will serve as terminal expressions each of which comprising of a *widget type* and optional *attributes* associated with the instance.

- A widget type defines a family of widgets with similar functionality, such as a text box or a radio button.
- Attributes affect the widget's appearance, such as background and foreground colors, font, frame thickness, etc., or it may change its behavior, e.g. determine whether the widget is enabled or disabled (in "disabled" state, the widget is grayed-out and cannot accept user input).

Minimal set of supported widget types:

- Label
- Text box
- Button
- Check-box
- Radio button
- Image (loaded from an image file)
- Combo box
- Slider

In addition, the empty widget type () will be used for boxes which do not contain any widget. Such boxes are used for fillers and spacers.
Minimal set of supported attributes (not all attributes are applicable to all widgets):

- .text (a string)
- .image (a filename)
- .checked (boolean)
- .value (integer)
- .enabled (boolean)
- .fgcolor (a 24-bit hexadecimal value of the form `0xBBGGRR`)
- .bgcolor (same thing)
- .font (tuple of the form ⟨font name, size, style⟩)
- .halign (for text alignment: left/center/right)
- .valign (for text alignment: top/middle/bottom)
- .minvalue (integer)
- .maxvalue (integer)

3.2.2 Native Dimensions

Normally, if an atom occurs in the program without : $w \times h$, then :?×? is applied. We will make an exception in two special cases:

- A label defaults to : $w \times h$ where w and h are the width and height of the text inside the label.
- An image defaults to : $w \times h$ where $w \times h$ is the native size of the image assigned to it.

Note. If the native size is used and .text or .image are changed on runtime, we would like the layout to reflect the new dimensions. The easiest way to acheive this is to define artificial attributes .native_width and .native_height, set : $native\_width \times native\_height$ as the geometry, and make sure these attributes are updated whenever .text or .image is (see a more elaborate discussion of data flow in 3.3).

3.2.3 When Things Don't Add Up

In an ideal scenario, everything falls cleanly into place: all unspecified values are concretized to match known values, and the generated program maps any window size to precise placement of the widgets in the *Box* space.

Of course, this scenario does not always fulfill. This can be caused by two reasons:

Over-constraining. If a coordinate is determined by more than one constraint, but these constraints do not agree on the value. A common situation is that constraints from inside the box contradict constraints from the outside:

$$\left(\begin{array}{c} \underset{\text{[item 1]}}{label} : ? \times 20 \\ \hline \underset{\text{[item 2]}}{label} : ? \times 20 \\ \hline \underset{\text{[item 3]}}{label} : ? \times 20 \end{array}\right) : 50 \times 50$$

In the example above, the sum of heights of the boxes inside the parenthesis is 60, while the entire box is specified to a height of 50.

Under-constraining. If a coordinate cannot be determined uniquely, due to unspecified values which were not unified with any known value.

$$\left(\begin{array}{c} \underset{\text{[item 1]}}{label} : ?\times? \\ \hline \underset{\text{[item 2]}}{label} : ?\times? \end{array}\right) : 50 \times 50$$

Here, neither of the labels' heights can be unified with the constant height 50, and indeed many different solutions exist which satisfy the constraint, which is that the heights sum up to 50.

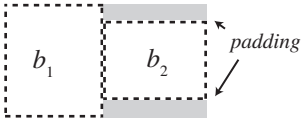
Rather than report these two cases as compile-time errors, we shall treat them in a special way, which will associate valid semantics to the program.

- When a box is overconstrained such that the width or height (or both) of its contents overflows its specified size, introduce a *scrollbar* which provides access to the overflowing region.
- When a box is underconstrained, introduce a new input variable which represents the unspecified width or height, such that the box now becomes fully specified. The run-time value for this variable should come from the user; for this purpose, create a *splitter* with a (horizontal or vertical) draggable bar.

Note. Scrollable areas and splitters are standard widgets. refer to the GUI toolkit documentation for information on how to create and manage them.

A special case of overconstraining stems from the use of binary box combinators ($B_1|B_2$ and $\frac{B_1}{B_2}$). These operators enforce unification of either the height or the width of composed elements. If they are both specified, it may be

impossible to unify. Instead of issuing a compile-time error, we can circumvent it by *padding* (adding space around) the smaller element to match the size of the larger one.

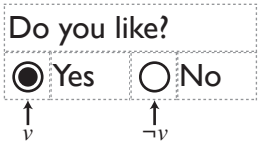


3.3 Data Flow Variety

3.3.1 Variables

Custom attributes. In addition to the built-in attributes described in the previous sub-section, the programmer can define her own. Custom attributes do not affect the widget but serve to maintain consistent state across several widgets. This is best illustrated by an example.

$$L = \left(\begin{array}{c} \underset{[\text{Do you like?}]}{label} \\ \hline \underset{[checked{=}v]}{radio} \;\Big|\; \underset{[\text{Yes}]}{label} \;\Big|\; \underset{[checked{=}\neg v]}{radio} \;\Big|\; \underset{[\text{No}]}{label} \\ [v{=}?(1)] \end{array} \right)$$



The figure above shows a dialog box with two radio buttons. The radio button labeled "Yes" is associated with the value "v", and the one labeled "No" is associated with the value "$\neg v$". It means that in any given moment, *exactly one* of these radio buttons is checked.

The attribute definition expression $[v = ?(1)]$ provides an initial value for v, to be used when the dialog is first shown. The expression involves both a question mark ("unspecified") and a value; the need for this will be explained later.

When one attribute is used in the definition of another, as in this case, the compiler has to generate, in addition to the regular GUI initialization code, some *event handlers*.

- When "Yes" is selected ($Yes.checked = 1$) – set $v = 1$ to maintain the definition of v. Consequently, set $No.checked = 0$ to match $\neg v$.
- When "No" is selected ($No.checked = 1$) – set $v = 0$, and proceed by setting $Yes.checked = 0$.

Width/height expressions. For the purpose of calculated geometries, we shall allow expressions to appear also for w and h in the : $w \times h$ operator. In cases where the expression is not a simple numeral, it will be enclosed in parenthesis. To allow referencing the height and width of other elements in the program in an expression, the special attributes *width* and *height* of an element will reflect the widget's current size.

3.3.2 Conditions

A common situation in almost any programming scenario is one in which the program has to perform one out of two or more computations according to some condition on its state.

Cases. Whenever a value is required, we shall allow the use of a *cases expression* to split the computation into two or more computations:

$$x = \begin{cases} expr_1 & cond_1 \\ expr_2 & cond_2 \\ \vdots & \\ expr_n & \text{otherwise} \end{cases}$$

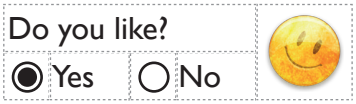
Each of $cond_i$ can be one of:

- An integer value (constant or attribute)
- An equation, as defined in 3.1.

The word "otherwise" will serve as a reserved symbol to indicate the value to use in case all of $cond_1, cond_2, \ldots$ are false.

Example:

$$\left(L \left| \begin{array}{c} image \\ [\{ \text{like.png} \quad v \\ \{ \text{dislike.png} \quad \text{otherwise}] \\ [v{=}?(1)] \end{array} \right. : 32 \times 32 \right)$$

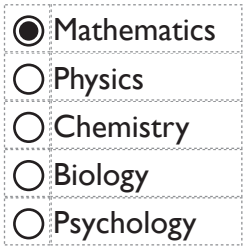


(L is the expression from the previous example.)

3.3.3 Iteration

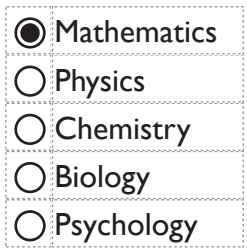
Repetition is a nuisance. Usually it is not sensible to specify multiple items, when they have identical or very similar properties.

$\underset{[checked=(v=0)]}{radio} : 20 \times 20$	$\underset{[\text{Mathematics}]}{label}$
$\underset{[checked=(v=1)]}{radio} : 20 \times 20$	$\underset{[\text{Physics}]}{label}$
$\underset{[checked=(v=2)]}{radio} : 20 \times 20$	$\underset{[\text{Chemistry}]}{label}$
$\underset{[checked=(v=3)]}{radio} : 20 \times 20$	$\underset{[\text{Biology}]}{label}$
$\underset{[checked=(v=4)]}{radio} : 20 \times 20$	$\underset{[\text{Psychology}]}{label}$



We strive to improve the above code by separating the common properties from the individual ones. The difference between two items of the multiple-selection list is the text of the label and the integer constant in the condition $(v = i)$.

$$\left(\frac{\begin{array}{c} \underset{[checked=(v=i)]}{radio} : 20 \times 20 \;\Big|\; \underset{[A_i]}{label} \end{array}}{\begin{array}{c} \vdots \\ \forall i = 0, 1, \ldots, 4 \end{array}} \right) \quad \begin{bmatrix} A_0\text{=Mathematics} \\ A_1\text{=Physics} \\ A_2\text{=Chemistry} \\ A_3\text{=Biology} \\ A_4\text{=Psychology} \\ v\text{=?(0)} \end{bmatrix}$$



Notice the subscript notation A_i. This allows the definition of *arrays* (or something of that sort) and accommodate lists of values. At run-time, they are handled just like a bunch of distinct attributes.

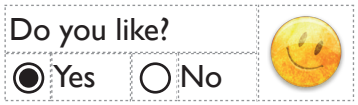
3.3.4 Subprograms

You may have noted the informal use of the symbol L to refer to a previously defined expression and use it as a sub-expression of a larger one. This is a desired trait in large programs; we shall therefore extend the semantics of the $\leftarrow$ operator to assign expressions to arbitrary identifiers (not just "main-window").

$$L \leftarrow \left(\frac{\underset{[\text{Do you like?}]}{label}}{\underset{[checked=v]}{radio} \;\Big|\; \underset{[\text{Yes}]}{label} \;\Big|\; \underset{[checked=\neg v]}{radio} \;\Big|\; \underset{[\text{No}]}{label}} \right)$$

$$I \leftarrow \left(\underset{\left[\begin{cases} \text{like.png} & v \\ \text{dislike.png} & \text{otherwise} \end{cases}\right]}{image} : 32 \times 32 \right)$$

$$\text{main-window} \leftarrow \underset{[v=?(1)]}{L \;\Big|\; I}$$



Especially notice the use of v inside the expressions for L and I. The definition of v occurs only in main-window. This should be accepted by the compiler, and is used as a way to allow *parametric* subprograms.

3.4 What Are These Strange Characters?

The above sections depict programs as various algebraic formulas. Of course, in practice, it is most convenient to write programs in plain text files. Therefore,

each operator has to have an alternative that can be typed on a computer keyboard.

As the horizontal combinator, $A|B$, use the ASCII character `|` ("pipe" or "stroke").

Instead of $\frac{A}{B}$, use any sequence of 3 dashes or more:

```
A --- B
```

or equivalenty (whitespaces are ignored anyway) –

```
  A
 ---
  B
```

For convenience, the `|` operator should precede over `---`. For parenthesis, use the normal `( )` pair.

Instead of : $w \times v$, use the form `:20x30` for plain number literals and "?", and `:(w)x(h)` when expressions are involved, to avoid the letter "x" being ambiguously interpreted as a variable identifier.

Instead of $\underset{[attributes]}{atom}$, use `(atom)[attributes]`. For example:

```
(label:?x20)[text="typical"]
```

As usual, string attribute values are surrounded by double quotes. Attributes are comma-separated. If an expression is used as the value of an attribute, it should also be parenthesized.

Operators inside expressions:

- `+`, `-`, `*`, `/` for addition, substraction, multiplication, (real) division. No surprise here.
- `=` for *equality* (note that there is no assignment within expressions).
- `!`, `&&`, `||` for logical not ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$).
- For the cases operator:

```
{(cond1) => (expr1), (cond2) => (expr2), ...
 ... , otherwise (exprn)}
```

Instead of $\leftarrow$, use `<-`.

For iteration, surround the layout operator with asterisks: `*|*` and `*---*`. The operator is then followed by a range specification, e.g. `i=1..6`.

4 Integration

4.1 Input/Output

In order for the program to have any interaction with the outside world, it has to receive input and produce some output.

Input. A valid *input* to a program is a full or partial mapping $FVar \to Val$ such that $FVar$ is the set of *free variables* of "main_window" (for this purpose, any attribute definition of the form $v =?$ or $v =?(\cdots)$ defines a free variable) and Val is the set of numeric and textual values.

For example, for the program L above, the following are valid inputs:

- $v \to 0$
- $v \to 1$

Output. The program always terminates as soon as "main_window" is closed. The output is a full mapping $Var \to Val$ such that Var are *all* the attributes of "main_window", and it maps every attribute to its actual value at that point of the execution.

A canonical Unix-like invocation of a program will look very much like that:

```
% L v=0
<...waits until window closes...>
v=1
```

Or for the multiple-choice list of fields:

```
% PickField v=1
<...waits until window closes...>
A[0]=Mathematics
A[1]=Physics
A[2]=Chemistry
A[3]=Biology
A[4]=Psychology
v=3
```

4.2 Interface to the Host Language

The language we designed is not stand-alone. It is not even Turing-complete; for example, there is no way to compute the product of two floating-point numbers. To extend the capacity of the programs we write, we wish to exploit the power of a full-scale general-purpose language and combine it with the simplicity and succinctness of our mini-DSL.

4.2.1 Procedural API

As the previous subsection suggests, a relatively easy composition technique would be allowing DSL programs to run as sub-programs in the host language. As a sub-program, it will have the perceived prototype of a function with one argument and a return value; the argument is of a mapping type representing the *input*, the return value is another mapping representing the *output*.

($\cdots$ *under construction* $\cdots$)

```
>>> l = L()
>>> l({'v': 0})
<...waits until window closes...>
{'v': 1}
```

Note. When using the procedural API, it is no longer necessary to name the entry point "main_window"; This will allow the programmer to define more than one window per program.

4.2.2 Listeners API

Listeners allow the programmer to intervene program execution and steer its flow according to whatever logic she has in mind. This is done by responding to *events* in the UI. Events can be intercepted at certain program points, often referred to as *extension points* in software engineering jargon. We choose our extension points to be attributes; and in order not to have too many of them, we restrict them only to attributes of the application's main window.

To plug into an extension point, the program provides an *event handler* (also called in this context *extension*) and associates it with an event. The event handler is a function that is to be invoked for every change to the associated attribute.

An example in Python (using the program L defined in the previous section):

```
>>> l = L()
>>> def report(old, new):
...     print "v: %s -> %s" % (old, new)
...
>>> l.when_changed('v', report)
>>> l()
v: 0 -> 1
v: 1 -> 0
v: 0 -> 1
{'v': 1}
```

Equivalent code in Scala:

```
object Program extends Application
{
    val l = new L()
    l.when_changed('v',
       (println("v: " + _ + " -> " + _)))
    l()
}
```

This code will cause the window to be displayed, and whenever the value of v changes (as a result of user interaction with the radio buttons), the transition is printed to the standard output console in terms of the previous and updated values of v.

The return value of `l()` is a dictionary containing the value of v as the main window is closed.

Here is another, more elaborated example involving a color picker. The dialog box involves a radio button group with labels "Red", "Green", "Blue". When the user selects a radio button, a preview of the selected color is presented in a square box to the left of the selection group.

$$ColorPicker \leftarrow \left(\frac{radio}{\substack{[checked=(v=i)] \\ \forall i = 0..2}} \Big| \underset{[C_i]}{label} * \right) \Big| \underset{[bgcolor=clr]}{()} : 40\times? \quad \left[\begin{smallmatrix} v=?(0), clr=?(0) \\ C_0=\text{Red}, C_1=\text{Green}, C_2=\text{Blue} \end{smallmatrix} \right]$$

Control code in Python:

```
>>> c = ColorPicker()
>>> def preview(old, new):
...     colors_rgb = [0x0000ff,  # red
...                   0x00ff00,  # green
...                   0xff0000]  # blue
...     c.set('clr', colors_rgb[new])
...
>>> c.when_changed('v', preview)
>>> c()
```

Control code in Scala:

```
object ColorPickerApp extends Application
{
  val c = new ColorPicker()
  val colors_rgb = List(0x0000ff,  /*red*/
                        0x00ff00,  /*green*/
                        0xff0000)  /*blue*/
  c.when_changed('v',
       c.set('clr', colors_rgb(_)))
  c()
}
```

The method `set(attr, value)` sets the value of an attribute in the application's main window. This serves as another extension point that is used to transfer control back to the UI part of the application; as a result of calling `set()`, any other attributes which depend on `attr` are also re-computed.

4.2.3 Native Function Invocation

Frequently the expressiveness of the linear arithmetic-based expressions of the DSL will fall short of covering the kind of computations required by the application developer. While in essence this can be circumvented via the Listeners API of the previous subsection, it is sometimes more convenient just to be able to extend the expression power of attributes.

We can do that by providing an access to operators written in the native language (in this case, our host scripting language). They will be seen by the DSL as "black box" compute mechanisms which take some arguments and generate a new value. Whenever any one of the arguments associated with the native function call changes, the black box is blindly invoked to re-compute the value of the target expression.

For example, in this expression PQ:

$$PQ \leftarrow \left(\underset{[\text{text}=multiply(w0,h0)]}{\underset{[w0=width,h0=height]}{label}} \right)$$

The attribute 'text' of the label is defined as calling the function 'multiply' with the arguments being the custom attributes $w0$ and $h0$. Since they are bound to the main window's *width* and *height*, this results in the following scenario:

1. The user resizes the window by dragging its frame.
2. The attributes *width* and *height* of the main window are set to the new dimensions.
3. As a result of the equations $w0 = width$ and $h0 = height$, and the code generated from them by the Arithmetic Variety, the new values are assigned to $w0$ and $h0$.
4. Since input arguments to 'multiply' have been updated, the core UI code is requested to invoke 'multiply' with the new values, generating a new value for the label's 'text'.
5. The label now displays the result of the call to $multiply(w0, h0)$.

To fill in the body of 'multiply', the programmer must provide a function with the corresponding number of arguments and some meaningful return value. We provide the method `bind` for that purpose.

Use case in Python:

```
>>> p = PQ()
>>> p.bind('multiply', lambda x,y: x*y)
>>> p()
```

Use case in Scala:

```
object PQApp extends Application
{
    val p = new PQ()
    p.bind('multiply', _*_)
    p()
}
```

Using this mechanism, we may rewrite `ColorPickerApp` from the previous subsection as:

$$ColorPicker \leftarrow \left(\frac{radio_{[checked=(v=i)]} \mid label_{[C_i]}}{\forall i = 0..2} * \right) \Big|_{[bgcolor=preview_rgb(v)]}^{()} : 40\times ?$$

$$\left[{}^{v=?(0)}_{C_0=\text{Red},C_1=\text{Green},C_2=\text{Blue}} \right]$$

Python code:

```
>>> c = ColorPicker()
>>> colors_rgb = [0x0000ff,  # red
...               0x00ff00,  # green
...               0xff0000]  # blue
>>> c.bind('preview_rgb',
...        lambda i: colors_rgb[i])
>>> c()
```

Scala code:

```
object ColorPickerApp extends Application
{
  val c = new ColorPicker()
  val colors_rgb = List(0x0000ff,  /*red*/
                        0x00ff00,  /*green*/
                        0xff0000)  /*blue*/
  c.bind('preview_rgb', (colors_rgb(_)))
  c()
}
```

Now for some examples.

Notepad

$$\text{main-window} \leftarrow \left(\begin{array}{c} \underset{[image=new.png]}{button} : 32 \times 32 \;\Big|\; \underset{[image=open.png]}{button} : 32{\times}? \\ \underset{[image=save.png]}{button} : 32{\times}? \;\Big|\; () : 10{\times}? \\ \underset{[image=copy.png]}{button} : 32{\times}? \;\Big|\; \underset{[image=paste.png]}{button} : 32{\times}? \;\Big|\; () \\ \hline \underset{\left[\begin{array}{c}text=\text{Enter your text here} \\ font=\langle \text{Arial,16pt,bold} \rangle\end{array}\right]}{textbox} \\ \hline \underset{[text=\text{Status: ready}]}{label} \;\Big|\; \underset{[text=\text{help}]}{button} : 40 \times 20 \end{array} \right) : 300 \times 200$$

Steve Jobs

$$\text{main-window} \leftarrow \left(\text{My-Team} \;\Big|\; \begin{array}{c|c} \text{Picture} & \text{Profile} \\ \hline \multicolumn{2}{c}{\text{Projects}} \end{array} \right)$$

$$\text{My-Team} \leftarrow \left(\begin{array}{c} \underset{[\text{My Team}]}{label} :? \times X \\ \hline () \end{array} \right)$$

$$\text{Picture} \leftarrow \underset{[bgcolor=green]}{\left(\begin{array}{c} () :? \times B \\ \hline () : B{\times}? \,\big|\, image \,\big|\, () : B{\times}? \\ \hline () :? \times B \end{array} \right)}$$

$$\text{Profile} \leftarrow \left(\begin{array}{c} \underset{[text=\text{Profile}]}{label} :? \times X \\ \hline \underset{[\text{Steve Jobs}]}{label} :? \times 20 \\ \hline \underset{[\text{CEO}]}{label} :? \times 20 \\ \hline \underset{[\text{Apple Corporation}]}{label} :? \times 20 \\ \hline \underset{[\text{company.png}]}{image} \;\big|\; () \;\big|\; \underset{[\text{flag.png}]}{image} \end{array} \right)$$

$$\text{Projects} \leftarrow (\cdots \textit{under construction} \cdots)$$