University of Edinburgh

Department of Computer Science

Two-Dimensional Syntax for Functional Languages

by

Luca Cardelli

Internal Report CSR-115-82

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

May, 1982

Two-Dimensional Syntax for Functional Languages *

Luca Cardelli
Dept. of Computer Science
University of Edinburgh
JCMB The King's Buildings
Edinburgh EH9 3JZ, Scotland

* This paper has been submitted for publication in the 1982 European Conference on Integrated Computer Systems.

Author's present address: Bell Laboratories,
600 Mountain Avenue,
Murray Hill,
New Jersey 07974
U.S.A.

Two-Dimensional Syntax for Functional Languages

Luca Cardelli
Dept. of Computer Science
University of Edinburgh
JCMB The King's Buildings
Edinburgh EH9 3JZ, Scotland

Abstract

The ideas discussed in this paper developed from some attempts at programming with boxes and other graphical data structures. Boxes, intended as rectangles with reference points (like in Knuth's TEX text formatter) are interesting data structures for expressing two-dimensional layouts of text or pictures. They can be composed and moved around by a simple set of operations, and can be manipulated in a fairly abstract way, often independently of their exact size.

Unfortunately, the programs one writes to compose boxes are not very suggestive of the compound boxes they produce, mostly because an essentially two-dimensional activity of box composition has to be flattened out in textual format. Is it possible to write these programs on a (high-resolution) screen in some graphical fashion, so that they can give a feeling of what they are doing? This can be done easily for non parametric expressions, but if we look for generality the need for two-dimensionality rapidly spreads to all the features of the language. One wants to manipulate lists of boxes, to write functions producing boxes, etc. The problem becomes that of defining a general purely graphical notation for arbitrary data structures (not just two-dimensional ones), and involves solving difficult problems like the graphical interpretation of parameterisation.

This paper is a preliminary attempt at graphical programming. Some standard programming constructs do not seem to fit in this approach, while some non standard ones developed for functional languages fit particularly well. We describe a graphical syntax for a small but complete functional language, avoiding semantic considerations whenever possible, and we introduce a simple graphical data structure (boxes) which can be more effectively manipulated by a graphical notation.

These ideas may be the basis for a programming system using a structure editor to manipulate two-dimensional representations of programs. What we hope to achieve with this kind of notation is freedom from the keyboard slavery, an intuitive interface for naive users, and an effective way of exploiting high-resolution screens and pointing devices. Experiments are needed to determine whether this approach can help in the normal software development activity.

Along with the increasing availability of graphical devices, picture manipulation is becoming a more and more important and widespread activity. We hope that notations on the style of the one we propose can help in making it also an easier activity, especially when naive users are concerned.

Two-Dimensional Syntax for Functional Languages

Luca Cardelli
Dept. of Computer Science
University of Edinburgh
JCMB The King's Buildings
Edinburgh EH9 3JZ, Scotland

Introduction

The ideas discussed in this paper developed from some attempts at programming with boxes (Cardelli 82) and other graphical data structures (Cardelli 81). Boxes, intended as rectangles with reference points (Knuth 79), are interesting data structures for expressing two-dimensional layouts of text or pictures. They can be composed and moved around by a simple set of operations, and can be manipulated in a fairly abstract way, often independently of their exact size.

Unfortunately, the programs one writes to compose boxes are not very suggestive of the compound boxes they produce, mostly because an essentially two-dimensional activity of box composition has to be flattened out in textual format. Is it possible to write these programs on a (high-resolution) screen in some graphical fashion, so that they can give a feeling of what they are doing? This can be done easily for non parametric expressions, but if we look for generality the need for two-dimensionality rapidly spreads to all the features of the language. One wants to manipulate lists of boxes, to write functions producing boxes, etc. The problem becomes that of defining a general purely graphical notation for arbitrary data structures (not just two-dimensional ones), and involves solving difficult problems like the graphical interpretation of parameterisation.

This paper is a preliminary attempt at graphical programming (Lakin 80). Some standard control constructs do not seem to fit in this approach, while some non standard ones developed for functional languages (noticeably case-analysis (Burstall 80)) fit particularly well. What we hope to achieve with this kind of notation is freedom from the keyboard slavery, an intuitive interface for naive users, and an effective way of exploiting high-resolution screens and pointing devices. Experiments are needed to determine whether this approach can help in the normal software development activity.

In the following sections, we first examine a simple but complete general purpose language having a graphical syntax, and then we introduce boxes and their operations. Examples are given of most of the features of the language, but they are not intended to be exhaustive; some effort may be required to see how the examples fit the definitions.

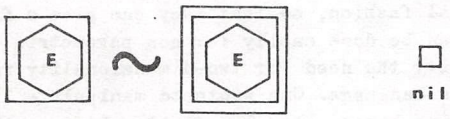
Variables and Simple Data Types

We begin with the graphical interpretation of variables as ellipses; to distinguish an ellipse from another we may insert some text or symbol into them. The ellipses surrounding identifiers may sometimes be omitted, espec-

ially when the identifiers are in function position. The type of a variable is inferred from the context of its occurrences; we assume here the polymorphic type system of Edinburgh ML (Gordon 79) (ML also inspires many of the features of the language).

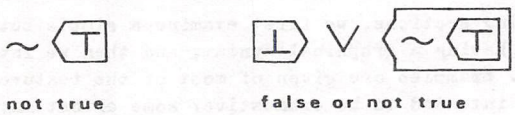
For the purpose of describing the syntax of expressions we need meta-variables denoting them. We use hexagons for this purpose, noticing that they are not part of the actual syntax.

Parentheses are drawn as rectangular frames which may contain any expression. Multiple parentheses may be collapsed. An empty parenthesis is interpreted as the object nil, the only element of the primitive type null.



Boolean constants and operators are denoted by the following symbols:

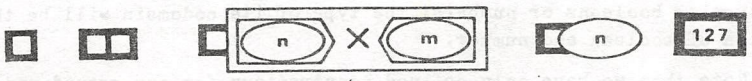
We introduce here the syntax for function application, in order to show some boolean expressions. The argument of a function is enclosed in a "dented" box pointing to the function; infix operators with two or more arguments have several argument boxes pointing to them:



Numbers are built from a constant (zero) and two unary operators (succ and pred):

zero succ pred

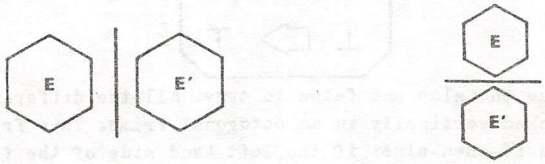
The succ operator is applied by appending it to the left of an existing number or expression, so that succ(zero) looks like one (1!) little square, succ(succ(zero)) like two (2!) squares, and so on. Similarly for pred, which is appended to the right and can form negative numbers. As an abbreviation we also allow to put arabic numerals in number frames:



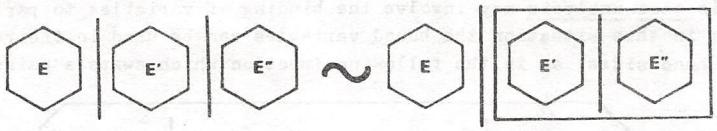
1 2 n×m + 1 n + 1 127

Somebody might worry about graphical ambiguities (like nil and the number 1, which are very similar); we assume that we are always able to disambiguate these situations, e.g. by the thickness of the lines or the absolute size of an object. We shall see later that this is not an important issue because we shall not actually try to parse pictures mechanically, and some ambiguity may be tolerated.

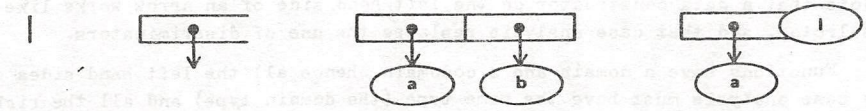
We can now consider some simple compound types. A pair of objects can be formed by a vertical or horizontal bar:



Pairing is a right-bottom-associative operation, so that the following expressions are considered equivalent:

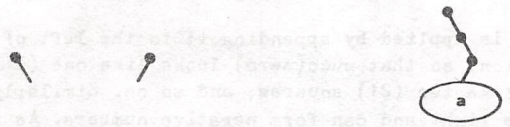


Homogeneous linear lists are obtained by a constant (empty) and a binary operator (cons) having as arguments an object and a list of objects of the same type:



empty cons cons(a,cons(b,empty)) cons(a,l)

Finally we introduce objects of disjoint union types by the unary operators of left injection (inleft) and right injection (inright) in the left and right part of a disjoint union:



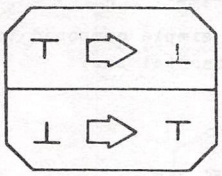
inleft inright inleft(inleft(inright(a)))

These operations are used, for example, when we want to write a function returning booleans or numbers: the type of its codomain will be the disjoint union of boolean and number.

Note that we have only defined constructors for our ground and compound types, ignoring selectors and discriminators. This is justified by the introduction of case-analysis (a simple form of pattern matching) in the next section.

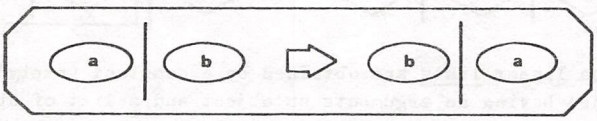
Functions and Declarations

Functions are defined by cases on a set of "typical" inputs. For example the boolean not operation can be expressed as:



i.e. not maps true to false and false to true. All the different cases of a function are stacked vertically in an octogonal frame. This frame is to be interpreted as an if-then-else: if the left hand side of the first case matches the input, then the right hand side is evaluated and given as result, else the subsequent cases are considered in turn. If no case matches the input. then we have a run-time failure.

This case analysis may involve the binding of variables to parts of the input; in this situation the bound variables can be used in the respective right hand sides, as in the following function which swaps a pair:

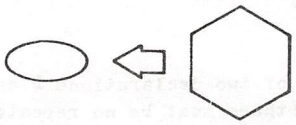


Note that a data constructor on the left hand side of an arrow works like a selector, and that case analysis replaces the use of discriminators.

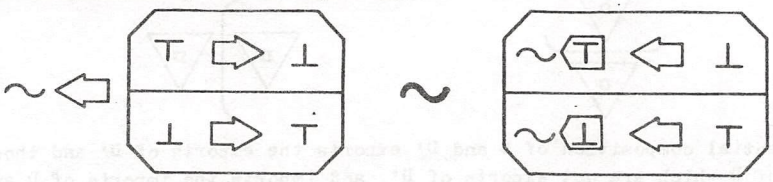
Functions have a domain and a codomain, hence all the left hand sides of a case analysis must have the same type (the domain type) and all the right hand sides must have the same type (the codomain type). These rules can be checked automatically without explicit type declarations (Milner 78), and

they are not restrictive because of the presence of disjoint union types.

To assign names to functions, and in general to any data object, we introduce definitions in the form of left-pointing arrows:

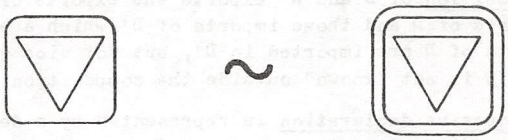


here, the variable on the left hand side of the arrow is bound to the (value of the) right hand side. In general, a pattern may appear on the left hand side (just like in case-analysis) and it should match the result of the right hand side. In the case of function definition we may use an abbreviation exemplified by the not operation:



Multiple definitions are organized into declarations. A declaration is either a single definition, or the composition of simpler declarations. We use four operators for assembling declarations, which were first described in (Milne 76); parallel declarations are independent of each other; sequential declarations are the usual cascaded declarations, each of them possibly using the previous ones; private declarations account for own variables; recursive declarations allow us to define recursive functions. From a semantic point of view, we are working with a statically scoped language using call-by-value for parameter passing.

We use triangles as meta-variables for declarations, and round-edged rectangles to bracket declarations:



To explain the effect of declarations, we say that each declaration imports some variables, used in the right hand sides of its definitions, and exports other variables (those being defined), appearing in the left hand sides of its definitions. A simple definition imports all the free variables used in its right hand side (i.e. those variables not bound by an inner case-analysis or declaration), and exports all the variables occurring in its left hand side (there may be several of them because of pattern matching, but they must all be distinct).

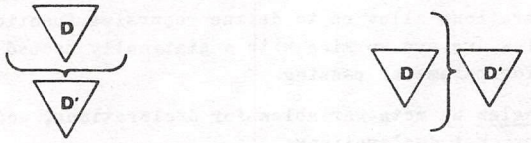
The simplest form of compound declaration is obtained by parallel composition, represented by the simple juxtaposition of declarations:

The parallel composition of two declarations D and D' exports the union of the exports of D and D' (there must be no repeated definition), and imports the union of the imports of D and D'. The exports of D are not imported in D' and vice versa.

The sequential composition of declarations is represented by the following right-bottom-associative operator:

The sequential composition of D and D' exports the exports of D' and those exports of D which are not exports of D', and imports the imports of D and those imports of D' which are not exports of D. Moreover the exports of D are imported in D', but not vice versa. Hence D may be used in D' and outside the composition (if it is not "hidden" by D').

The private composition of declarations is represented by another right-bottom-associative operator:

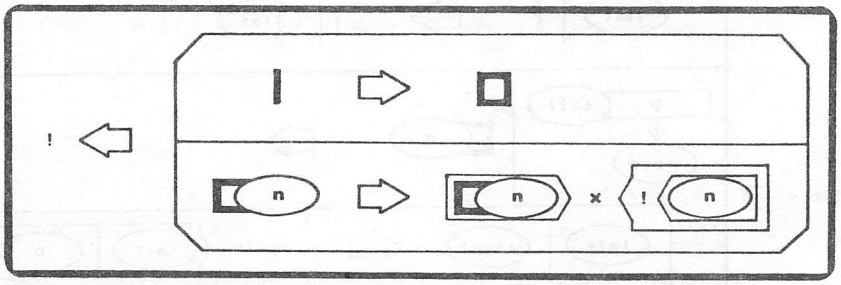


The private composition of D and D' exports the exports of D' only, and imports the imports of D and those imports of D' which are not exports of D. Again, the exports of D are imported in D', but not vice versa. Hence D is "own" by D' and it is not "known" outside the composition.

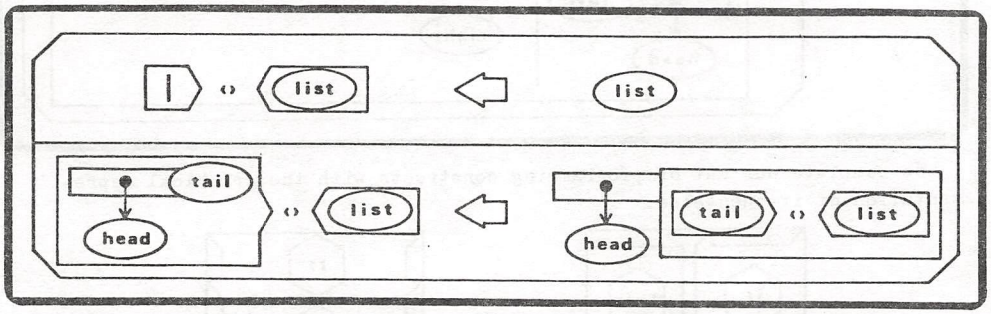
Finally, a recursive declaration is represented by a declaration enclosed in special brackets:

The recursive closure of D exports the exports of D, and imports those imports of D which are not among its own exports. Moreover the exports of D are imported back into D. Hence D "knows" its own definitions. Recursive declarations may only contain function definitions, and must not contain private declarations.

We are now able to draw some interesting examples, starting (of course!) with the factorial function:

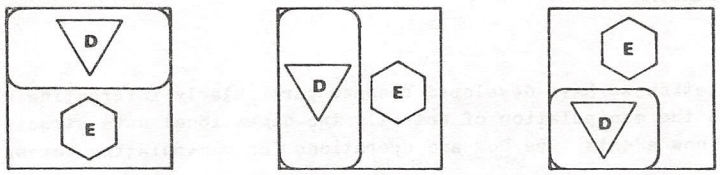


The append function gives a simple example of list manipulation:

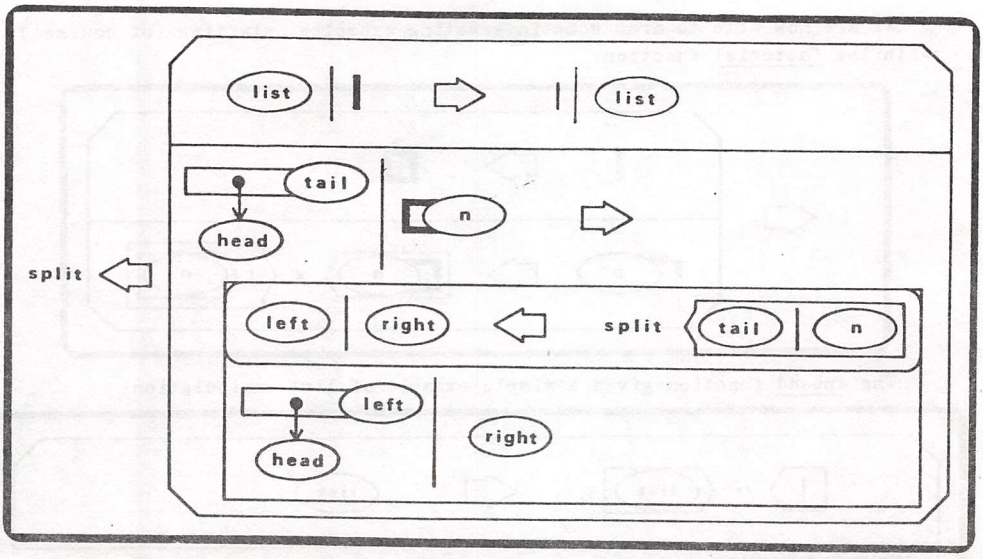


Note how the arrows have been reversed in this example, according to a previously defined abbreviation.

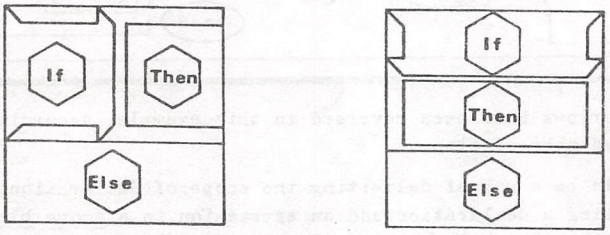
There should be a way of delimiting the scope of declarations. This is done by inserting a declaration and an expression in a scope block; the expression is then the only part of the program having access to the exports of the declaration:



An example of scope block is given in the following split function, which splits a list at a given position, returning the two halves as results. Note that every function takes a single argument and returns a single value, but values may be pairs, tuples (i.e. multiple pairs) or lists, simulating the effect of functions with multiple arguments and results.



We complete our set of programming constructs with the graphical representation of if-then-else:

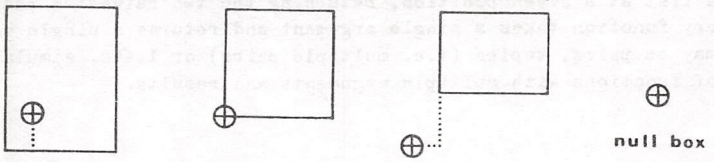


The else part may contain another if-then-else without need to enclose it in a surrounding frame.

Boxes

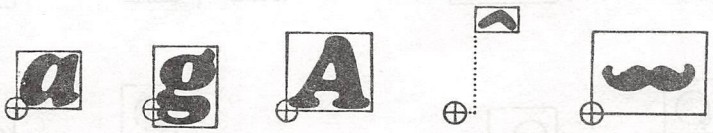
The notation we have developed becomes particularly interesting when applied to the manipulation of actually two-dimensional data structures. We introduce now a data type box and operations for manipulating boxes.

A box is a rectangle with a reference point (⊕). Boxes always have horizontal and vertical sides, i.e. they cannot be oblique.

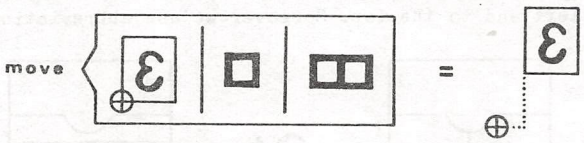


There is a set of basic boxes which we assume here to contain characters and graphical symbols. In general, basic boxes may contain very complex

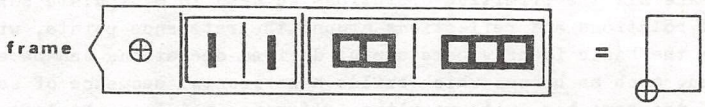
pictures, generated by direct graphical interaction, or by an adequate set of graphical functions (e.g. splines).



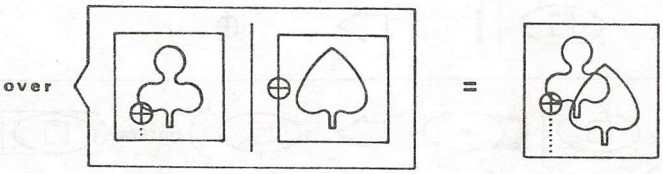
Furthermore, there is a set of primitive operations on boxes:
- size: takes a box and returns its size as a pair of numbers (x,y);
- refpoint: takes a box and returns a pair of numbers which are the displacement of the reference point from the lower left corner of the box;
- move: takes a box and two numbers, and moves the box with respect to its reference point by the quantity specified by the numbers, e.g.:



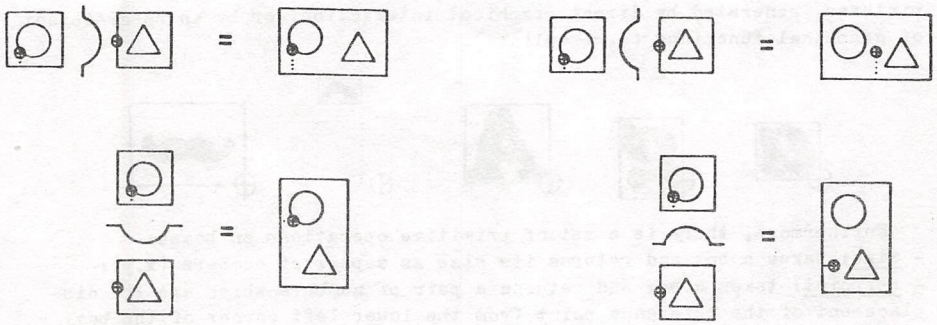
- frame: takes a box and two pairs of numbers, and replaces the rectangle of the box with the rectangle specified by the four numbers (the first two are the lower left corner and the second two are the upper right corner):



- over: takes two boxes and overlaps them identifying their reference points. The result is the smallest rectangle enclosing both the argument rectangles, with reference point in the common reference point, e.g.:

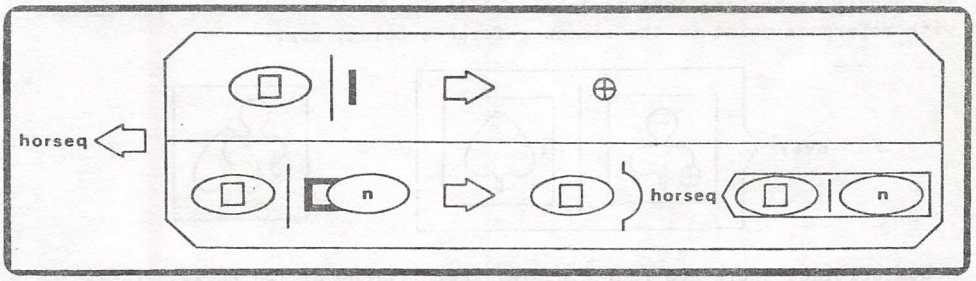


- composition: there are four kinds of compositions of two boxes, denoted by horizontal and vertical "bumped" bars (see below). Horizontal composition is obtained by placing the two boxes so that the right side of the first box is aligned on the same vertical line as the left side of the second box, with reference points aligned on the same horizontal line. The result is the smallest box enclosing both arguments, where the reference point is the reference point of the first box if the "right-bumped" composition is used, or the reference point of the second box if the "left-bumped" composition is used. Similarly for vertical composition, which connects in the top-to-bottom direction.

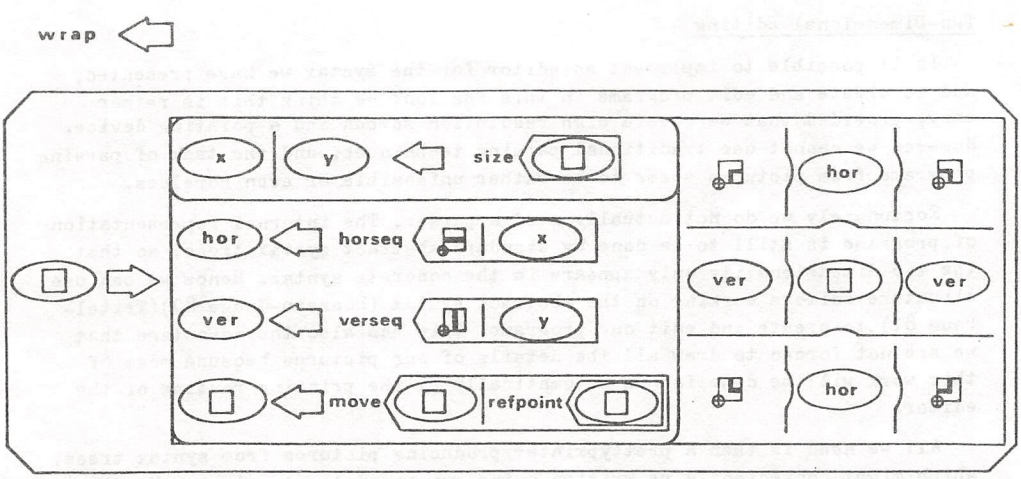


Compositions are not in general associative, hence by convention they associate to the left and to the top. Moreover we use abbreviations like:

These are all the primitive operations we need to manipulate boxes (we might add rotations and reflections around the reference points, without affecting the basic ideas). Some useful derived operations can be easily programmed, such as horseq which builds a horizontal sequence of several copies of the same box, with resulting reference point to the left (and similarly for verseq, with resulting reference point to the bottom):

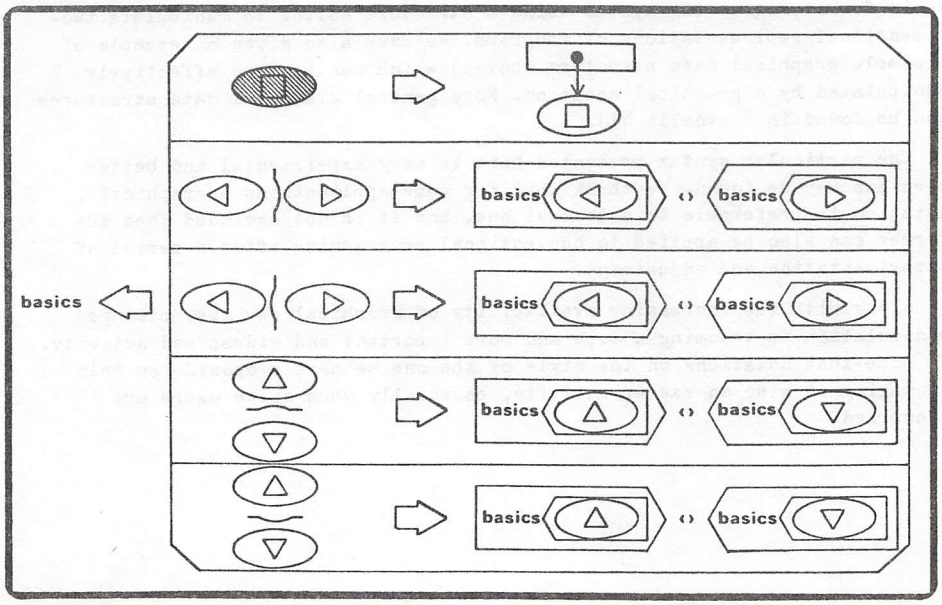


Here is a function, called wrap which puts a frame around a box, leaving the reference point in the lower left corner:



The little basic boxes composing the frame in the figure above have unitary size. Note that the reference point of the argument box has to be renormalized to the lower left corner (to place the frame in the correct position): the result of the refpoint operation is used to move the box.

In order to use case analysis on boxes we adopt the convention that a shaded variable matches any arbitrary basic box. We can then write, for example, the following basics function which builds a list of all the basic components of a (compound) box:



Two-Dimensional Editing

Is it possible to implement an editor for the syntax we have presented, and to create and edit programs in this fashion? We think this is rather easy, provided that we have a high-resolution screen and a pointing device. However we cannot use traditional parsing techniques, and the task of parsing programs from pictures seems to be rather unfeasible or even hopeless.

Fortunately we do not actually need a parser. The internal representation of programs is still to be done by standard abstract syntax trees, so that the two-dimensionality only appears in the concrete syntax. Hence we can use structure editors working on the abstract syntax (Donzeau-Gouge 80)(Teitelbaum 81) to create and edit our programs. This has also the advantage that we are not forced to draw all the details of our pictures because most of this work will be done for us automatically by the printing routine of the editor.

All we need is then a prettyprinter producing pictures from syntax trees, which might conveniently be written using our box data structures. We might also have a more conventional prettyprinter, producing formatted text, and we could merge the graphical and textual notations according to our taste. However, in the most extreme situation, we would only work on pictures, carefully hiding the abstract syntax representation, so that the user could have the feeling of manipulating pictures directly.

Conclusions

We have described a graphical syntax for a small functional language, avoiding semantic considerations whenever possible. These ideas may be the basis for a programming system using a structure editor to manipulate two-dimensional representations of programs. We have also given an example of a simple graphical data structure (boxes) which can be more effectively manipulated by a graphical notation. More general graphical data structures can be found in (Cardelli 81).

The particular syntax presented here is very experimental and better versions may be found. We think that for some applications a graphical notation is preferable to a textual one, and it is not excluded that the former can also be applied to conventional programming after a period of experimentation and adjustment.

Along with the increasing availability of graphical devices, picture manipulation is becoming a more and more important and widespread activity. We hope that notations on the style of the one we have proposed can help in making it also an easier activity, especially when naive users are concerned.

References

(Burstall 80) R.M.Burstall, D.B.MacQueen, D.T.Sannella: "Hope: an experimental applicative language". Proc. 1980 LISP Conference, Stanford.

(Cardelli 81) L.Cardelli, G.Plotkin: "An algebraic approach to VLSI design". in J.P.Gray (ed.): VLSI 81. Academic Press.

(Cardelli 82) L.Cardelli: "PaperBox: an applicative text formatter". Unpublished program documentation.

(Donzeau-Gouge 80) V.Donzeau-Gouge, G.Huet, G.Kahn, B.Lang: "Programming environments based on structured editors: the Mentor experience". Report 26, INRIA.

(Gordon 79) M.J.Gordon, R.Milner, C.P.Wadsworth: "Edinburgh LCF". Lecture Notes in Computer Science, n.78. Springer-Verlag.

(Knuth 79) D.E.Knuth: "TEX and Metafont". Digital Press.

(Lakin 80) F.Lakin: "Computing with text-graphic forms". Proc. 1980 LISP Conference, Stanford.

(Milne 76) R.E.Milne, C.Strachey: "A theory of programming language semantics". Chapman and Hall.

(Milner 78) R.Milner: "A theory of type polymorphism in programming". JCSS, vol.17, n.3.

(Teitelbaum 81) T.Teitelbaum, T.Reps, S.Horwitz: "The why and wherefore of the Cornell Program Synthesizer". Proc. of the ACM symposium on Text Manipulation.