Diagram Based Languages

DaS

It is often believed that programming languages come in only 2 forms: (1) textual and (2) visual.

Visual Programming has become to mean pixel-based image processing.

I have implemented an intermediate form of programming. Something between (1) textual and (2) pixel-based image processing.[1]

I call it DaS – Diagram as Syntax. It is *diagrammatic programming*. I used to call this "visual programming", but that phrase has come to mean something else, something much harder to implement.

aha

I backed into a number of design principles. I will do my best to summarize them below...

- Most programming languages and compilers are based on glyphs. A *character* is a small bitmap. A *character* is a *glyph.* Programming languages are based on grids of non-overlapping glyphs. Language design has been driven by programming-editor capabilities, e.g. emacs, Vim, VisualStudio. Fixed-font programming editors determine the shape of programming languages (not the other way around). Programming editors are frozen in the 1950's, whereas non-programming editors (e.g. Word) have advanced to include variable-sized fonts, diagrams, images, etc.
- Modern hardware is not constrained to editing non-overlapping glyphs in a grid.
- It is OK to mix text and diagrams. A diagrammatic language does not need to be 100% diagrams. Some concepts, e.g. concepts like "a = b + c", are better expressed as text.
- Diagrams for concurrency need only a few overlapping glyphs [2]– (1) boxes, (2) arrows, (3) text.
- Diagrams for StateCharts need only a few overlapping glyphs – (1) ellipses (or rounded boxes), (2) curved lines, (3) text
- Backtracking is OK.
- PROLOG can be used as a parser.
- A single app can use more than one paradigm, e.g. if PROLOG backtracking is used for parsing, the rest of the app is not constrained to use only PROLOG

1 "Visual Programming" also meant, for a while, software development using GUIs. MFC and Visual BASIC were touted as "visual programming".

2 Here, I am using the word "glyph" to mean "atomic graphical symbol". See also https://en.wikipedia.org/wiki/Glyph. Unlike character glyphs, atomic graphical elements are not fixed size, but, like character glyphs they can be bound to a single input gesture (e.g. a keystroke). There are not many of these kinds of glyphs in any given notation (e.g. about 3).

- DSLs everywhere – multiple DSLs can (should[3]) be used in one project. It is OK to build source-to-source converters that let the base language do the heavy lifting – this makes building DSLs much less onerous.
- Hierarchical composition – makes Software Architecture, DI[4] easier and, more expressive, and promotes *divide & conquer*[5]
- I want to derive interesting information about a diagram (code). For example, I want to know the (x,y) for a box, I want to know (x,y) for the start-point of a line, I want to know (x,y) for the end-point of a line. I can used backtracking pattern matching to derive some of this information.
- Current PLs (programming languages) are based on the concept that, in (x,y),"x" is a character position and "y" is a line number. Characters are strictly non-overlapping and sequential. I would say that characters are not 2D, but something less, like 1.5D. Current hardware can do better – (x,y), *can* be represented as pixel coordinates and full 2D glyphs can overlap.
- I need to ensure that my "editor" gives me enough information. The requirements for a diagrammatic programming editor are different from the requirements for a business editor (like Word, VISIO, Draw.io, etc.).
- SVG and XML based editors and diagram editors produce diagrams that contain much of this needed information, but they also contain lots of noise (aka syntactic sugar). It might be better to build one's own diagram-programming editor, but in the meantime, Draw.io, yEd, etc., might suffice.
- I am more concerned with DI than with Maintenance Engineering, Efficiency Engineering, Test Engineering, etc., etc. From this perspective – i.e. DI - I don't care about the efficiency of parsers that employ backtracking, as long as it doesn't keep me waiting, on my computer.
- Hierarchical composition – keeps things small. O(3) doesn't matter when things are small. O(3) still runs "fast enough".

Box-And-Arrow Diagrams and Concurrency

The canonical form of "visual programming" - which I call DaS – is the box-and-arrow diagram. It is a network diagram wherein nodes are functions instead of full-blown computers.

Most attempts at implementing box-and-arrow diagrams have failed (as far as I know). Most attempts at "visual programming" have failed (as far as I know – visual programming can produce pretty pictures, but do not address PLs).

3 See my essay "DSLs - The Future of Computing"
4 Design Intent
5 See my essay "Divide And Conquer"

I have been successfully using box-and-arrow diagrams in production since the mid-1990’s. FBP[6] has been using box-and-arrow diagrams since the 1960’s. UNIX® pipelines are a degenerate form of box-and-arrow-diagrams.

Why have most attempts at box-and-arrow diagrams failed?

Concurrency[7].

Box-and-arrow diagrams do not work well in the sequential paradigm.

All of the working instances of box-and-arrow diagrams – that I know of – treat boxes as being concurrent components and treat arrows as pipelines of concurrent messages between (concurrent) components.

UNIX® pipelines and existing implementations of FBP use heavy-weight *threads* to implement concurrency. I implemented the concurrent paradigm without using *threads*. I used something like closures.

6 <ref to FBP>
7 See my essay “Concurrency is a Paradigm”