

The Visi Lanugage

David Pollak

December 26, 2011

Contents

1	Introduction	5
2	Motivation	7
3	Language Samples	9

Chapter 1

Introduction

This document describes the Visi language. Visi is an open source language that blends concepts from spreadsheets, scripting languages, functional languages such as Haskell and OCaml, and other systems. The goal of Visi is to be accessible for Excel power users, yet be “correct” such that runnable code should be substantially bug-free. Visi forms the basis for the Visi.Pro platform that allows programming on iPads and those programs can run on iPads, iPhones, as well as in the cloud.

Features of Visi include:

- Guaranteed serializability of user-created data structures (like Erlang) such that data structures can seamlessly migrate across address spaces.
- Visually pleasing syntax that makes simple programs as well as very complex programs easy to understand. The syntax is whitespace-oriented and eschews curly-braces, line ending markers, etc.
- Persistent (immutable) data structures except for certain data structures that do not escape the boundaries of library calls (everything that “end users” see is immutable).
- Clear demarcations of side effects (sources [input]/sinks [output] and references) such that the order and locus of computations is invisible to the programmer until a commit operation to a sink or reference occurs.
- A type system that is simultaneously powerful and invisible. End users cannot write type annotations. Library authors can construct very complex type expressions that are evaluated at compile time to insure code correctness. Programs will not be allowed to run unless they pass the type checker. Note that getting error messages “right” will be a serious challenge for Visi.
- Visi will be self-hosted with a built-in IDE like Smalltalk. The initial IDE will be Mac OS X based.

- Visi will support incremental development such that changes can be made to “running” programs like changes can be made to a spreadsheet and the changes are immediately reflected. However, Visi programs will be compiled to various more efficient representations including converting Visi lambda calculus representations to GHC intermediate representations such that Visi programs can be compiled to any supported GHC back-ends including LLVM, native code, and potentially JavaScript. Visi can also be run interpreted on iOS devices.
- Visi, like spreadsheets, performs computations when external state changes and updates outputs. This makes Visi ideal for writing systems that rely on external data feeds and Visi can trigger events when new data from data feeds is received.

This document is an evolving description of the Visi language as well as a discussion/justification for the design decisions.

Chapter 2

Motivation

Do we need yet another computer language? Is there currently a “Cambrian explosion” of computer languages and why Visi?

Well, yes. Most computer languages, especially the ones that are cropping up these days, seem to re-visit ideas of past computer languages. They make minor syntactic changes or in other ways make small alterations to the kind of basic concepts that have been around computing for years.

There are notable exceptions. Clojure embraced Lisp syntax, but fundamentally changes mutable state into an issue of time (see http://www.artima.com/articles/hickey_on_time.html). Scala made material advances in computer languages by blending a rich mostly consistent type system with object oriented programming. Most other languages that have been introduced this millennium are minor variants on Smalltalk or C++ or Java.

Visi takes a different approach. Visi approaches the problem of describing how a computer should respond to input in a similar manner to spreadsheets. Visi approaches computing from the perspective of a dependency graph when outputs change as inputs change and only outputs that depend on a particular input are recomputed when a particular input changes. The dependency graph is intuitive to anyone who has ever put together an Excel spreadsheet (or a 1-2-3 spreadsheet or even a VisiCalc spreadsheet.)

More broadly, my motivation for Visi is to create a language that fundamentally changes the way people program computers such that Visi is a language oriented to humans rather than a veneer on top of computing machinery. Visi is not a tool for writing compilers (although it will be mostly self-hosting). But, instead, Visi is a tool for normal people to describe relationships such that a network of computers can perform calculations based on external input and generate predictable, correct output. My motivation for Visi is to change the landscape of computer languages the way that VisiCalc changed the language and computing landscape in 1979.

Chapter 3

Language Samples

Let's take a look at some Visi language samples.

First, "Hello, World!":

```
"Greeting" = "Hello , World!"
```

The lefthand side of the equation has quotes around it, meaning that it's a "Sink". A Sink is output that can be wired up to a user interface or some other external output.

Next, let's take a number from a "Source" (an input), add 1 to the number and send it to a "Sink" called "Plus One":

```
?number  
"Plus One" = number + 1
```

It's easy to define functions:

```
addOne n = n + 1 // a function that adds 1 to the input  
?number  
"Plus One" = addOne number
```

And functions can be recursive:

```
fact n = if n == 0 then 1 else n * fact n - 1  
res = fact 10 // 3628800
```

Syntactically, variables (and local functions) need only be offset by spaces from the upper level declaration:

```
f n = // add 33 to the input
  v = 33 // the variable v is set to 33
  n + v // return the result because it's the last line of the function
```

In action:

```
f n =
  v = 33
  n + v
res = f 3 // res == 36
```

Functions can be local as well:

```
f n = // calculate the factorial of the input
  fact n = if n == 0 then 1 else n * fact(n - 1)
  fact n
```

Inner functions can shadow outer function. The function in the nearest scope, wins. Also partially applied functions can be passed as parameters:

```
fact n = n & "hello" {- proper scoping: fact is not the inner fact -}
f n = {- Test partially applied functions -}
  // a local fact function that is visible within this function only
  fact n = if n == 0 then 1 else n * fact(n - 1)
  app n fact // apply the fact function to the input

{- Apply the function f to the value -}
app v f = f v

res = f 8 // 40320.0
```

Partially applied functions close over local scope:

```

f b = {- test that the function closes over local scope -}
  timesb n m = n * b * m
  timesb {- partially apply the function which closes over
          the scope of the 'b' parameter -}

app v f = f v

q = f 8 // return a partially applied function

z = f 10 // return another partially applied function

res = (app 9 (app 8 q)) - ((z 8 9) + (z 1 1)) // -154

```

Rolling input, functions and output together:

```

{- Input -}
?taxRate // source the tax rate
?taxable
?nonTaxable

{- Computations -}
total = subtotal + tax
tax = taxable * taxRate
subtotal = taxable + nonTaxable

{- Output -}
"Total" = total // sink the total
"Tax" = tax // sink the tax

```

And the type checker insures you don't mix types:

```

f = 3
d = f & "hi" // fails... can't mix a number with a String

```

The typer supports the identity function and type variables:

```

q n = n
f n = if true then n else (q n) // both q and f are generic function

```

Generic functions, even at multiple levels of recursion can be correctly typed:

```
a n = b n
b n = c n
c n = a n // a, b, and c are all generic functions
```

Functions can be mutually recursive without any hints to the type checker (like OCaml's `let rec`):

```
isOdd n = if n == 1 then true else not (isEven (n - 1))
isEven n = if n == 0 then true else not (isOdd (n - 1))

// define the not function
not n = if n then false else true

// the result is true (9 is odd)
res = isOdd 9
```

More language samples as Visi evolves.