1.

During spring break, I've begun to read a few articles about Computer Science Education, and especially those dealing with the difficulties encountered by beginners. I am clearly not at the end of this process, but I have already looked at a few dozens of them, and there are a few very interesting ones. I have uploaded them in a public folder: https://copy.com/UdZIPVAaOkzUVxsI. There is also a public Evernote notebook where I put a lot of links, especially from Bret Victor for now (but there are other interesting stuff) https://www.evernote.com/pub/lemebfr/brownproject. Both of them are updated regularly.

2.

After these readings, I think that the issues we are going to encounter are threefold. The problems encountered by would-be programmers are complex indeed, and therefore can really be resolved only by a mixture of syntax, user interface, and programming environments improvements.

Syntax (and Compiler)

Apart from the usual things, such as simplicity and easiness to learn, here are a few constraints that we need to think of:

1. *Ideally fit to multiple ways of learning / programming styles*. Some people learn and do assignments by copying and pasting, and there are good reasons to think this will not go away with the rise of open source tools, online tutorials, and the general tendency to copy other news orgs. Adopting functional programming could be, for instance, a good way of tackling this reality. Some people program more by trial and error: a reactive, always-compiling GUI would potentially be more interesting (see below). If we try to adopt the principles of Flow Based Programming (see below), we probably may need to be able to divide the code architecture into very small units.
2. *Fit to multiple working environments*. Newsrooms have different workflows and different tools. Odds are anyone who wants to code in a newsroom will have to incorporate different tools into his programming workflow. Of course, technologies such as JSON make stuff much easier. That said, we should think of integrating the language with other ones. Should we make this a sub-thing of Python? Should we let user execute a bunch of lines of code of other programming languages into ours? To which extent should we facilitate the copying and pasting of functions and instructions written by other users?
3. *Made for journalists, not computer scientists*. What this means is perhaps hiding the math as much as possible, and perhaps creating and including as standards the most useful data structures. This may be dangerous…

GUI

A few thoughts about user interfaces:

1. Generally, the beginners use two types of software to write their first lines of code. They can use Scratch, or Codecademy, which are very cool software to understand the very basic principles of programming, but require a transition to more professional tools afterwards. They also can use dryer programming softwares, such as Eclipse or vim, that are definitely not as user-friendly, and require a steeper, and for some unreachable, learning curve. Really, it is too bad that no tool has really bridged the two worlds.
2. That said, professional tools become better and better. There are countless attempts and trials of making development and debugging easier. Autocompletion, for instance, is super cool, and more and more GUIs detect errors before compilation. Clearly, if we want to do a GUI, it would have to be on the side of professional tools.
3. There are some important features lacking in these GUIs. One of them is the permanent compilation. Xcode Playgrounds really is amazing and doable. Plus, if you look at it, Playgrounds have been conceived not only as a learning tool (you can mix text, code and interactive results in a special version of Playground files), but also as a great tool for developers who just want to sketch some programming ideas. It really is a good programming abstraction for everyone.
4. Another important feature is live preview of the result, by which I mean permanent live preview. Adobe Brackets apparently does that: you see the result of the file you are working on instantaneously, and the part of the file you are hovering with your mouse is even indicated. Live rendering is an enormous area of investigation. Bret Victor's "Learnable Programming" (available in the Copy and the Evernote folders) argues that one of the big problems of live rendering is that you are not able to grab and move the elements you created. Generally, it would be cool to be able to do the same modification on your code in a multiplicity of ways, in order to conform to the variety of users' styles and practices. Especially if we consider that the majority of the products made with our tool will be somehow displayed.
5. Other issue mentioned by Victor: it would be important to make the hidden state tangible. For instance, that would mean seeing the dataset or the different variables, and their evolution in real time. Also, manipulations of code should be tangible (transform constant values with a slider, being able to transform lines of code into a function or a loop by grabbing them, etc.)

Other questions I have not quite thought through: Should we make a physical notebook-looking interface, with code written in a monospaced but serif font? Should it be web-based? Should it integrate flow-based programming à la FlowHub or by having a tab in which are stacked all your different streams of data? Oh, and should we think of making a publication platform? How are we going to export the outputs of the program?

Development Environment

Development Environments are surprisingly, and almost absurdly, dry. When people want to begin to program, they generally will have to discover more than the computational thinking and the syntax. They will have to learn how to use a command-line environment. This is dry and sometimes discouraging.

In some ways, it is absurd to hold command-line environment as the ultimate environment for programmers. There are things that one can do more powerfully in command-lines (such as scripting), but, for other tasks, command-line is just worse. If you want to search for libraries and extensions, you need to search on Google or Github because there rarely are a Web Store or an App Store of a programming language. If you want to install a library, you need to install multiple dependencies one after the other and copy and paste different lines in your terminal, whilst praying for the support of the version of your OS.

We perhaps need to ask more from library developers. Not only to write their instructions in "real" English, but perhaps also other things. For instance, to include visual examples of their libraries for the documentation or for the autocompletion in the GUI. Or to indicate all the dependencies so they can be installed in one click. I may be mistaken and not having seen such development environments, but it seems so absurd to me that installing libraries is so painful and counter-intuitive to the novice.

Plus: An introductory course?

Of course, one of the goals of this language is pedagogy. Perhaps we should think about developing a specific form of courses for this language. There is A LOT of material has been written about programming courses, and Mark has a lot to say about it. However, I was kind of scared to think about a programming language that would only be great to *learn*. As shown by Scratch, they sometimes are very attractive, but ultimately dumbed-down versions of programming that cannot be used on a sustainable basis.

Instead, what we would need is a PL great to learn but mostly *great on the long-run*. In a sense, the part of the learning curve we have to tackle is not the very left of it. After all, so many things have been done over the last decades to find metaphors for the basic concepts of programming that I don't really believe it is an interesting area of investigation. What is most important to tackle is what comes just after, when it comes to actual programming (and often debugging). In the Copy folder, you can find an article entitled "How Novices Tackle Their First Lines of Code in an IDE" which kinda says that focusing on the very first steps is probably not worth the energy.

That does not mean we should not think about a potential class. The education perspective is one of the most important part of the project, if not the most important one. So I don't know to which extent we should try to develop educational tools. Plus, there would be a lot of questions. Should we put an emphasis on J-School / academic learning, or more on self-paced, professional, potentially newsroom-enforced learning? Should we integrate tutorials in the GUI? Should we think about optimizing the language for teachers?

Anyway, there are a lot of potential directions. I spoke with Gabe about that, and we agree that there is no way we can now know which aspect of the project to prioritize. The issue is that some of the solution we could propose would be intertwined in the syntax, GUI and environment aspect, so prioritization may not be easy.

3.

Since the project is primarily about a PL, I first tried to know what syntactic solutions could be envisioned. This is what I'm describing in the rest of this document. Of course, that's a very rough draft. Even though I tried to check when possible that the ideas were not implemented in other PLs, the originality of the ideas may just be the result of my ignorance, and Gabe's extensive knowledge would be quite useful there. Also, for the sake of clarity, and because I was also thinking of what the code would render on a GUI IDE, I highlighted the code.

There a lot of details we perhaps need to determine, such as whether to make the language loosely typed, or if we need to reduce the number of types to a few ones, or if we need to make it heavily object-based, or closer to languages like Clojure, etc. Here, I really wanted to focus on what could make a language adapted to journalists.

Get rid of mathematical notation

The first, and most important feature, of the syntax, is the ability to express instructions in a non mathematical way, but in a true English-like way. The classic mathematical function notation is:

```
name_of_function(arg1, arg2)
Or: name_of_function arg1 arg2
(if you assume the first word is the name of the function)
```

This is a completely counter-intuitive way if you have a non-mathematical background. As Bret Victor mentioned, sometimes some functions who work that way are extremely cryptic, because it basically asks arguments without any context. As a consequence, when you see that kind of function for one of the first times, when you skim through your code, or when you are debugging, you tend to replace this notation by a literary translation in your head. Why not adopting this altogether, especially for social scientists and journalists? This literary notation would be:

```
Performs an operation with (arg1) and (arg2)
    Or: (arg1) is processed with (arg2)
Or: (arg1) and (arg2) are processed together
```

Again, this notation is, in a lot of ways, and especially for journalists, much better than mathematical notation. It is way more legible, it is easier to debug, and it actually provides a literary look. A lot of developers would tend to say that mathematical abstraction is inherent to computer science, and that departing from this is an heresy, but I frankly don't buy it. Especially when you can approximate mathematical notation in a literate way, as I've tried above.

To the best of my knowledge (and to the best of Google's knowledge), I have seen no programming language fulling embracing this possibility as a tool for clarification for social scientists. The closest I have seen is Inform (inform7.com), which is a programming language for interactive fiction, and, less relevantly, Chef*, which basically is an esoteric language expressing functions as cooking instructions. HyperTalk and AppleScript are also very English-like, but they still do adopt the mathematical notation for user-created functions.

* (www.dangermouse.net/esoteric/chef.html)

How does it concretely differ?

1. *The structure of an instruction is extremely flexible* and is basically determined by the user.
2. *Instructions begin with a capital letter*, and can use punctuation to make code more legible (see below).
3. *Functions are defined not by a name, but by a pseudocode*. It echoes, interestingly, a paradox in Bret Victor's essay. Victor throws the idea of displaying explanations of code when the mouse comes over it (e.g., over Processing code `ellipse(x, y, a, b)`, a text like "Draw an ellipse at x,y with height a and length b"). But why should we have a programming language on one part and an explanatory language on the other part? It does not really make sense, especially if we consider source code as something that primarily needs to be read by humans. Therefore, it would be more interesting to make the explanatory language the *actual* programming language.

What would it look like?

For instance, here is what some included Python functions would look like, if translated in a literary way:

```
Python function                 Literary function (arguments in parentheses)
---------------                 --------------------------------------------
abs(x)                          Absolute value of (a number)
bin(x)                          (an integer) as binary
bool(x)                         (an object) as boolean
cmp(x,y)                        Compare (an object) and (an object)
enumerate(x)                    Enumerate (a list)
filter(function, iterable)      Filter all the true elements from (a function)
hash(object)                    Hash (an object)
hex(x)                          Hexadecimal value of (an integer)
input([prompt])                 Prompt the user [with message (a string)]
len(s)                          Length of (an object)
```

We could go even further and try to formulate some functions (or sentences, or utterances) as interrogations. Functions that return boolean are particularly fit for this.

```
Python function                 Literary function (arguments in parentheses)
---------------                 --------------------------------------------
any(iterable)                   Is any element of (a list) true?
issubclass(class, classinfo)    Is (a class) a subclass of (a class)?
```

Punctuation

One of the most salient differences between journalists and mathematicians / computer scientists is the usage of punctuation. For the latter ones, the variables are placed between parentheses; therefore, parentheses create placeholders for values. Generally, journalists do not *write* like that. Parentheses are there to add additional informations. Placeholders, in the contrary, are between [ ] brackets, (e.g. *[Editor's note: blabla]*). Why not use the same syntax?

```
name = Ask user input with "What is your name?"
(after asking [something] is optional)

Print "Hello [name]!" (Parentheses are great because,
                      then,
                      comments can run on multiple lines)
```

Translations

The other advantage or such an approach is that we could also think of translating the programming language in foreign languages. In French:

```
prénom = Demande à l'utilisateur "Quel est votre prénom ?"
(les instructions sont impératives et non descriptives)

Affiche "Bonjour [prénom] !"
```

Perhaps we could go even further in the approach. Inform is an imperfect but worthy example of a really English-based language. It may not be applicable to data journalism, but its radical aspect makes it really interesting.

Markdown-style syntax

Markdown-style syntax could be useful. We could definitely think about, as in IPython Notebook, or in Literate CoffeeScript, merging text and code in the same file, and using Markdown for that. However, perhaps more interesting would be to think code *as the text* (since it is expressed in a literate way), and Markdown-style notation as a way to distinguish different elements. For instance, one could imagine stars as a way to indicate to the compilers where variable values are. What would it look like, in the end? Probably something like this:

```
true_false = {True, True, True, False} ({ are delimiting lists)

boolean = Are all the elements of *true_false true? (* being optional)
Print *boolean (=> false)
```

Examples

Let's imagine we want to the equivalent of Python's all() function. all() returns true if all the elements of a list are true, and false if not. The way to express the function in plain, literary English is "Are all the elements of [the list] true?", with a question since a boolean is returned. What we basically the function to do is (a) to scan every element of the list and, if this element is not true, or not a boolean, to return false, (b) if the function has not returned false, to return true. Literary speaking, that would look like something like this:

> *"Are all elements of the list true?"* means that,
>
> For each element in the list, if this element is not true or not a boolean, return false. If nothing has been returned, return true.

In code, we could convert it into this:

Simple version:

```
"Are all elements of [list the_list] true?" means:

  For (each) element in the_list,
    If the type of element is not boolean,
    Or If element is not true,
      Return false

  Return true
```

Complete, commented version:

```
Are all elements of [list the_list] true?, (are quotes necessary?)
all [list the_list],                  (possible to even define it as a "normal" function)
Is [the_list] completely true? means: (not proper English, but we can accept that :) )

  (commas are optional and can be replaced by a colon)

  For (each) element in the_list, (each is added)
    If the type of element is not boolean, (again, is not is equivalent to != or =/=)
    Or If element is not true,
      Return false

  (If, after all these verifications, nothing has been returned)
  Return true
```

The advantage of that way of doing is that it does not breaks with computational thinking. The syntax is very similar to any other major programming language, the language still uses the notion of function (even though formulated differently), and it still is a sequence of instructions. Perhaps this similarity is not contrastive enough with programming in general, but this could be a nice way to offer a language that could be used in parallel to other programming languages (Actually, the working title I have in my head for the language is Trojan).

Let's take another example. In this IPython Notebook, Brian Keegan investigates the idea that the movies that pass the Bechdel test make more money.* Let's compare his code in Python to a code in this language.

*http://nbviewer.ipython.org/github/brianckeegan/Bechdel/blob/master/Bechdel_test.ipynb

```
In [3]:  numbers_df = pd.read_csv('revenue.csv',encoding='utf8',index_col=0)
         numbers_df['Released'] = pd.to_datetime(numbers_df['Released'],unit='D')
         numbers_df.tail()
```

```
----------

numbers_df = Read file 'revenue.csv' with encoding utf-8 beginning at 0
(since we need a star to indicate a variable,
 the encoding can be expressed without quotes)

Standardize *numbers_df[Released] to time in days
(the function modifies directly the variable)

Tail of *numbers_df

----------
```

```
In [5]:  movie_ids = json.loads(urllib2.urlopen('http://urldetest.com').read())
         imdb_ids = [movie[u'imdbid'] for movie in movie_ids]
         print u"There are {0} movies in the Bechdel test corpus".format(len(imdb_ids))
```

```
----------

movie_ids = Load JSON from [Read URL 'http://urldetest.com']
(brackets are here for clarity, in a Ruby way)

imdb_ids = For (each) movie in movie_ids,
               Return *movie[imdbid]

Print "There are [Length of *imdb_ids] movies in the Bechdel test corpus"
```

Plus, it is possible, I think, to apply functional programming principles to this syntax.

```
tw_d = Get Twitter data from the last 10 minutes mentioning "Obama"
tw_d2 = Sentiment analysis on *tw_d

(or, if the function "Sentiment analysis" has no list as parameter)
tw_d2 = Apply Sentiment analysis on *tw_d
(in Python, this would be tw_d2 = map(sentimentAnalysis, tw_d))
```

Embrace diversity

Another particularity of writing is that every writer has a distinctive writing style. Of course, programmers have different programming styles, but they all use the exact same functions and expressions. This may be a detail, but I guess some people will understand, and formulate in their minds, the same functions or operations differently. For instance, testing that a variable is a string can be expressed in multiple ways:

```
If the type of var_name is string
  If type of var_name is string
     If var_name is a string
(I'm not even including operations such as var_name.typeOf() ==.)

Or:

x = 2
2 -> x
x <- 2
x becomes 2
x is now 2

Or: (cf. example above)

Are all elements of a_list true?
          all a_list
   Is a_list completely true?
```

Literary speaking, there are multiple ways to express the same thing. Why not in a programming language, as long as this multiplicity is fairly bounded and limited? In general, this would probably mean that a good autocompletion in an IDE would be necessary so that the user could be comfortable with the multiplicity of the syntax.

We could also imagine to include optional words. For instance, if you want to know what the median of a list is, you could put '*what is* ' as an optional word:

```
what is Median of a_list
=> 32
Median of a_list
=> 32
```

Reducing errors

One of the biggest pains for would-be developers, especially coming from social sciences, are the number of errors and bugs they encounter. Actually, after having looked at "Investigating Novice Programming Mistakes, Educator Beliefs vs Student Data" (in the Copy folder), it is possible to see that some of the most common mistakes can be kind of corrected by the design of the PL.

For instance, one of the most common errors is mistaking '==', '=', '.equals()', etc. Sometimes, the meaning of these characters are inverse from one programming language to another (e.g. Java and Ruby). Even though the distinction between = and == is normal, the distinction between == and equals() is fairly absurd, since it generally is the same operation. We could imagine to merge those operations or, even better, let users choose what keyword they want to use.

```
If var_name == "Brown" / If var_name equals "Brown" / If var_name is "Brown"
```

Among other useful things would be a simplification of some number operations. That may be dangerous though, since it would imply, sometimes, an automatic conversion from integer to float and vice-versa.

```
Is -num equal to num * -1 (I think it works that way in COBOL)
=> true

var1 = 11
var2 = 2
division = var / var2 (=> 5.5 instead of 5, not to confuse users)
division_integer = var // var2 (as in Ruby)

var3 = 10
var4 = 2.0
division2 = var3 / var4 (=> 5.0, converted as integer)
```

That being said, a lot of the errors, instead of be corrected mainly by PL changes, could be better corrected if caught by the compiler / IDE. Therefore, that may be an interesting thing to look at from a non-language way.