# Spark - Pre-Birth of a Modern Lisp

Author: Shlomi Fish

Revision History
Email ASF

## Table of Contents



# Introduction

Spark is a Modern dialect of Lisp currently being planned. This document is not a formal functional (much less technical) specification for it, but rather a briandump of some of the conclusions I (= Shlomi Fish) have reached about the fundamentals of its behaviour. Nevertheless, some preliminary (and still subject to change) specification of it will be given and some code examples will brought.

Beside contemporary Lisp dialects such as Common Lisp, Scheme and Arc, Spark draws a lot of inspiration from other modern languages, paradigms, and technologies including Perl 5, Perl 6, Python, Ruby, Java and Haskell.

# Some Spark Essentials

These are some of the guiding elements in the design of Spark.

# Spark is not another implementation of Scheme (or Common Lisp)

There are far too many implementations of Scheme out there, and probably too many of Common Lisp. However, that is beside the point that we did not come to praise the existing dialects of Lisp, by implementing them again.

Spark will be a completely different dialect of Lisp. It won't be compatible with either Scheme (or any of its implementations), Common Lisp or even with Arc. It will still be Lisp, though, as we hope you will see.

# Spark will be a dynamic (so-called "scripting") programming language

Spark will be an alternative to such languages as Perl, Python, Ruby, PHP, Tcl, Lua or io-language, which are all very different but have APIs to achieve similar ends and are used for similar tasks. Like them and Lisp (which is one of the oldest dynamic languages) it will be able to determine a lot of behaviour at run time and will support dynamic "eval", call-by-name , run-time typing, run-time change of type for a datum, multiple dispatch, polymorphic macros and other features of Lisp and other dynamic languages.

Some people have been referring to Perl and friends as "scripting languages" but that implies they are only useful for scripts. See:

http://xoa.petdance.com/Stop_saying_%22script%22

While that article by Andy Lester illustrates the problem with labelling programming languages as "scripting languages", I still think saying "script" and "scripting" is a valid way to distinguish a trivial program from an application. For example `/usr/bin/gcc` is essentially a script written in ANSI C, which just passes controls to the various compilation stages. When we type `gcc` at the command line, we are running this script that does the hard work of doing the compilation for us. (`/usr/bin/gcc` should not be confused with GCC, the GNU Compiler Collection which is a compilation framework for C, C++ and other languages, and is a crucial piece of the open-source UNIX-like operating systems infrastructure).

For an "in your face" anti-thesis to the aversion to call languages scripting languages see Larry Wall's "Programming is Hard. Let's Go Scripting":

http://www.perl.com/pub/a/2007/12/06/soto-11.html

# Spark aims to be popular and be actively used for real-world tasks

While other general purpose Lisps such as Common Lisp, Scheme, Arc or Clojure have been influential and have some followers and users, none of them are actively used with the same popularity as Perl, Python, Ruby or PHP are. Spark aims to be a popular lisp dialect which will be actively used for real-world tasks, not just toy or experimentation code.

Eventually, it is our hope that some people will get paid to maintain Spark code. Some of them against their best preferences, like some people now are maintaining Perl 5, PHP or even Python code while preferring a different language. (Simply because it puts bread on their table, and they cannot get paid to write something else.)

Python? Yes! Google for example has been hiring several Perl 5 or even Perl 5 and Perl 6 programmers and instructing them to program mostly in Python. I've known two or three of them myself. Many of them tolerate or even like Python enough to program in it, but many of them still prefer Perl and consider

themselves as Perlers at heart. But Google gives very good conditions and many Perl programmers (despite a huge demand for them) are treated badly or have advanced to better paying positions. And naturally a lot of the “younger generation” (most of them) bought the hype that you shouldn’t need to learn Perl if you already know PHP, or Python or Java or whatever.

In any case, Spark aims to become roughly as popular as any of Perl/PHP/etc. It is not guaranteed that we succeed , but we still would like to try. What can we do to increase our chances?

Paul Graham has written about what makes a language popular. Arguably, he failed to make Arc as popular as he wanted, even after it was released (and may have violated some of his own guidelines), but the guidelines in the article are still sound (ad-hominem anecdote put aside).

One can spend a lot of time expanding on Graham’s points and adding more stuff (including possibly good timing and luck and possibly a lot of financial backing can help, which most open-source enthusiasts don’t have).

However, it is important to note that this is our ultimate goal. Some languages didn’t have a lot of financial backing before they got popular - Perl, Python, PHP and Ruby are examples of it. Can we do the same?

None of the P-languages are perfect. Someone on #scheme gave me a moderate list of must have items out of which Perl, Python, PHP and Ruby each violate at least one and Scheme and Common Lisp none. I can quote him here as the channel is publicly logged.

I’m not going to try to violate any of them (or any of the many common sense “never-do-it”’s that PHP alone has violated, to say nothing of the rest) on purpose in hope it will save me from the evil eye of having a decent language. But I don’t think languages succeed because they are bad but because they offer at least one awesome advantage. For example Perl 1 offered the ability to write pseudo-shell scripts in a much more robsut and powerful way than awk, sed, sh and csh and a much easier way than ANSI C did. (Although still very primitive by the standards of Perl 5.). By the time Perl 4 was abandoned in favour of Perl 5 , Perl 4 was already the de-facto standard for clueful sys admins who valued their sanity. And I recall when Perl 5 was the only real solution for writing CGI scripts that you could use because I was hired to work as a web developer at that time.

PHP was similar enough to Perl 5 to fool people into believing it was adequate (while messing up the internal behaviour and core language royally) and offered more ubiquity as far as setting up on web-hosts was concerned (which Perl 5 is still trying to catch up). You can now download very powerful and gratis content management systems in PHP (and other gratis web applications) and run them out of the box on your cheap hosting and call yourself a webmaster, which would be hard with any other UNIX-based solution. PHP as a language has many issues, but they got some parts of the implementation needs right, and this is what made it popular.

We need to find some good reasons for people to become hooked on Spark. Spark probably won’t be perfect because I am not perfect and don’t know all there is to know about language design. However, neither are Scheme and Common Lisp. But even if Spark doesn’t have so many quirks like PHP or Perl (or even Python) it should be an attractive platform for people to experiment with, write scripts in daily, write toy programs and serious applications, and then one day we may get a killer app like Ruby’s Ruby on Rails.

(Hopefully, we won’t get the added RoR hype, which I admit is based on viral marketing and community efforts, but was hype nonetheless. Perl and PHP were never strongly hyped, and they seem to have been doing fine without it.)

# Spark does not aim to compete with C and friends

Spark does not aim to fill the same ecological niche as ANSI C, C++ Objective-C, etc. - much less assembly. ANSI C and friends have been the de-facto standard for writing applications when some or all of certain constraints have been met:

http://www.shlomifish.org/philosophy/computers/when-c-is-best/

While their use have lately been diminishing somewhat due to the increasing attractiveness of the Java or .NET frameworks or the various dynamic languages (Perl/Python/PHP/Ruby/etc.) they are nevertheless still very much in vogue and even the backends for the more high-level virtual machines are written in C and C++ .

There have been some efforts to compete with C and C++ on their own turf such as D or Ecere (and earlier efforts such as Ada 95, or Object Pascal) and they can be commended for that, but unlike them, Spark does not aim to replace C in most of the valid use cases for a C-like language.

# Spark will have a rich type system but won't be strongly typed

Like Common Lisp, Python, Ruby and Perl 6 and to some extent unlike Perl 5, Spark will have a rich type system. However, it won't be strongly typed like Haskell. If Spark had been going to be strongly typed, it could no longer have been considered a Lisp, and I happen to like dynamic typing.

A variable can be assigned different values with different types during its run-time, and functions would be able to accept variables of any type (unless they specifically forbid it).

The Spark type system will be extendable at run time, and will be analogous to its Object Oriented Programming (OOP) system. As a result, one would be able to call methods on pieces of data, on expressions, on S-expressions, and on functions, macros, classes and method declarations, and on their application.

In Spark "everything will be an object", but unlike Java, it won't be overly-OO. One won't need to instantiate a class and declare a method just to print "Hello World" on the screen. This will work:

```
$ spark -e '(say "Hello World")'
```

Or:

```
$ spark -e '(print "Hello World\n")'
```

Or:

```
$ spark -e '(-> "Hello World" say)'
```

# Is a simple macro that converts `(# obj method args)` to `(method obj args)` and is used for some syntactic sugar.

# Spark will be capable of being used for Scripting

While Spark should not be called a scripting language, just as the name is a misnomer for Perl 5 or for PHP, it should in fact be capable of writing scripts, including command line scripts at the prompt or the REPL (Read-Eval-Print-Loop). Here are some examples for command line scripts. Most of these are taken from the descriptions in Peteris Krumins' "Famous Awk One-Liners Explained" series (which is now in the process of being augmented with "Famous Perl One-Liners Explained"). I'm not going to study the Awk implementations due to lack of knowledge in Awk and lack of will to learn it as I already know Perl 5 - its far superior superset, but I'll implement something similar in Spark (Hopefully, Peteris will feature a "Famous Spark One Liners Explained" feature in his blog someday too).

## Line Count:

```
$ spark -e '(foreach (fh ARGV) (++ i)) (say i)' [Files]
```

## Line Count Reloaded

```
$ spark -e '(: say len getlines fh ARGV)' [Files]
```

`(: … )` serves the same purpose as Haskell's `$` - to chain function calls without too many nested parameters. So this script is equivalent to saying:

```
$ spark -e '(say (len (getlines (fh ARGV))))' [Files]
```

But is shorter and cleaner.

## Double-space a file

```
$ spark -pe '(say)'
```

(Think Perl)

## Number lines in each file

```
$ spark -ne '(say "${^LINENUM} ${^LINE}")'
```

Here we can see the string interpolation of variables in action. `${….}` interpolates a single variable, while `$()` is an S-expression. Aside from that spark will also have sprintf, sprintf with named conversions similar to Python and something as similar as possible to Perl's Template Toolkit (while still being Sparky). I find Common Lisp's `format` to be hard to understand and much less flexible than Template Toolkit so I'm going to drop it.

Like in Perl 5 the `^VARNAME` variables are reserved and are usually in all-capitals. Unlike Perl 5 (or Common Lisp), we are not a Lisp-2 and we use the same symbol namespace for everything (like Scheme). So we can put assign a lambda (could be `(fun … )`, `(sub … )`, `(lambda …)` or `(function … )` - all exact synonyms) to a variable and call it with a value:

```
(my square)
(:= square (fun (x) (* x x))) ; Or (<- square) but I'm hazy about (= square)
(say (square 5)) ; Prints 25 followed by a newline.
```

Like in Perl 5, however, a symbol table has an arbitrary amount of slots which we can put values. So we can say:

```
(say :to (fh STDERR) "Warning, Will Robinson.")
```

Which will print to STDERR.

In the case of the `:to` named parameter to `(say)` (or to `(print)` or to `(printf)` or whatever we have or define) we don't need the explicit `(fh …)` (= file handle) call but it won't hurt.

## Note about command line magic

Out of convenience the `-e` and the rest of the `-p`, `-n` , etc. flags will involve some magic manipulation of the S-expression inside -e or inside the script. It also loads some convenient modules. However, sometimes we may wish to convert a command line script to a full application. That's what the `--dump-code=code.spark` flag is for. It dumps the code of the program to a file containing code that can be run with just `spark code.spark`.

For example:

```
$ spark --dump-code=say.spark -pe '(say)'
$ cat say.spark
(no strict) ; you should probably remove that.
(use re)
(use cmd-loop)
(cmd-loop.set-implicit-print 1)
(say)
$
```

Like all examples here, this is just for the sake of the illustration. Until version 1.0.0 comes out, everything can change. But the concepts will remain the same.

We encourage Perl, Ruby and other dynamic languages with rich command line interfaces, to steal the `--dump-code` idea. Maybe one day someone will become a multi-millionaire from selling a 300K Lines program that evolved from a simple `spark/perl/ruby --dump-code=code…. -e ….` invocation (after a successful plain `-e` invocation).

# Spark will have nested namespaces

Spark will have a similar namespace system to Perl 5, with nested namespaces, and the ability to selectively import symbols from namespaces at run-time. Similarly to http://metacpan.org/release/Sub-Exporter and as opposed to C++ where importing symbols from namespaces is an all-or-nothing operation, and so mostly unusable.

As opposed to Java one would be able to import several symbols from a namespace at once and grouped by tags. `Sub-Exporter` gives much more than that for Perl 5, but I don't recall all the details from the slides I saw about it. `:-)` .

Like Perl 5 one will be able to import symbols at run-time.

As opposed to Perl 5, classes won't be automatically associated with namespaces, and a namespace may contain one or more classes (or none). Like CPAN and unlike Java (`org.apache.jakarta`…), we will not enforce namespace purity, but hopefully there will be a better mechanism than the current CPAN and PAUSE (Perl Authors Upload Server) to be able to fork, spin-off, or branch CSAN distributions or choose between competing alternatives. CPAN6 (orthogonal to Perl 6) is worth a look for some ideas, as is "The Zen of Comprehensive Archive Networks"

# Spark will be more succinct than most Lisps, but not overly terse

One thing notable about Scheme and Common Lisp is that many identifiers and keywords there are excessively long. `(string#integer)` , `(lambda)` , `(concatenate)` , etc. Yes, you can easily assign aliases to them, but it:

1. Takes time to write and more time to maintain.

2. Will require you to carry and update a meta-syntactic package in all your code.

3. Would make your code harder to understand to the unfamiliar.

So Spark will provide short identifiers for most common operations (either macros or functions) by default. There won't be too many long aliases because they will increase the core and require more time to learn and become familiar with for the uninitiated, but some of them will be considered.

For example, a variation on my favourite Scheme statement is:

```
(vector-set! myarray idx (+ (vector-ref myarray idx) 2))
```

Which in Perl is:

```
$myarray[$idx] += 2;
```

And in Spark would be:

```
(+= (myarray idx) 2)
```

Which isn't much worse than Perl. We're not trying to beat Perl 5, Perl 6 or much less J in code Golfing in every case - this isn't a peeing contest. We're just trying to make easy tasks easy.

Furthermore, while +=, -=, /= and friends will be defined there will be a macro (that will be used defined by the prelude and used there)

And as you see we will try very hard that every sane expression can become an lvalue.

And in case, you're worried there will be a "\+\+" and "--" operators too, with post-increment/post-decrement and pre-increment/pre-decrement variations.

Finally, by inspiration from Arc, we decided to do something about excessive parens. So we will have a `(with (k1 v1 k2 v2 k3 v3) …. )` scoping instead of the unwieldy Scheme `(let*)` and `(letrec)` (both will be easily replaceable by +(with…) `with some macro or VM trickery.). And we'll have a C-style for-loop instead of the obscure +(do…)` and a while loop, and a Perl 5/Perl 6-style foreach loop, and maybe other loops too. And you can always use recursion.

However we're not going down the Arc route of assigning extremely short, and hard to pronounce and grep for identifiers. `(fn)` How do you pronounce it? fnn…. There's no such sound in Hebrew, so it's verboten by your Hebrew-speaking overlords. We like `(fun…)` because it puts the fun back in function ("Functional!! Parallelism!!!!" - oh wait! Wrong language.), and we like `(sub …)` because it puts the "sub" back in subroutine. And all Hebrew speakers will rejoice because they can pronounce `"cat"` exactly like `"cut"` and Perl like Pérl and Lisp like Leesp, and they can pronounce TeX and LaTeX with a honest-to-god khaph (or a Heth if they put their mind to to it.), and the God of the Israelite Programmers saw there were only 5 and a half vowels and he was pleased.

Seriously now, I don't like (fn) because it's hard to pronounce, doesn't sound right when you read it to your mind's ear, and is obnoxious. While being succinct is a noble goal, picking psychologically-sound and intuitive conventions is also important. I recall searching the Arc tutorial and documentation for a `(not)` function only to found it was spelled `(no)`:

```
(if (no soup) (print "soup is false"))
```

"If no soup". Oh no, no! No soup for you. **For one year!!!** We're going have (not) like everybody else, and also a "!" alias .

```
(if (not soup) (print "soup is false"))

(if (! soup) (print "soup is false"))
```

So we're going to borrow stuff from Arc, but only when it makes sense. Spark should have an up-to-date documentation manual right from the very start, which will be kept as up-to-date as possible, and naturally will have automated tests, which would serve as automatically verifiable examples. Arc really had none, and often you needed to read the implementation code, or one of the example web applications.

# Spark will be written in plaintext

What do I mean? `sbcl` forces to write a long command line just to run a program from the command line and exit. Arc was not better and could not execute a program directly. I added this functionality to the Arc git repository myself.

No, the REPL won't be gone in spark, and good old `(load)` will still be there. But you can also do:

```
$ cat hello.spark
(say "Hello World!")
$ spark hello.spark
Hello World!
$
```

Or put /usr/bin/env spark or whatever in your sha-bang and it will work.

A spark program will do its thing, execute to the finish and gracefully exit.

We will still have a REPL, that can be used from the command-line or from within Emacs SLIME or within IDEs such as Eclipse. Even Vim/gvim may get a REPL if it gets something like an embedded shell like Emacs has.

Spark programs can be abstract syntax trees, a network of objects, some compiled bytecode or flying unicorn ponies who drop candy. Nevertheless, they are still read from text. If you want to change the state of REPL until you forget what it has now or it changes unbeknowest to you ("Parallelism!!!!") it's an option. But you can still write your tests, run, debug, change, run debug, etc. Hopefully with automated tests for extra bonus points in Software Management sainthood.

# Regexps and other important elements have dedicated syntax

But don't worry - it's in the "re" module and it uses read-macros/ char-macros / text macros. With the help of such macros one can even create a parser for Ruby-style syntax (or Perl's - `;-)`), but it'll be actively discouraged.

So for example:

```
$ spark -pe '(~ ^LINE (re.s {ba([zro])(\s+)mozart} ma$1$2bozart))'
```

And it will replace the first `baz[whitespace]mozart` with `maz[whitespace]bozart` , etc. The ~ operator is similar to Perl 5's =~ or perl-5.10.0's or Perl 6's ~~ or in that it does a smart matching of a datum (which could be a list) to an abstract operation.

We can also use other delimiters instead of {…} in the `(re.s…)` read macro:

```
$ spark -pe '(~ ^LINE (re.s /ba([zro])(\s+)mozart/ ma$1$2bozart))'
```

# C/Perl/etc. conventions

1. Spark will be case-sensitive
2. Unicode (UTF-8) aware-and-safe
3. With C-style escapes - backslash does the right thing.

# Spark will not encourage a proliferation of implementations

As you all know Lisp is a family of languages, which includes Lisp, Scheme Arc, Emacs Lisp, etc. and some people may say also Dylan, and that Perl 5, Perl 6, Ruby, Python and most other modern languages have many Lispisms in them up to being able to translate many programs written in Lisp to them line by line. (See Paul Graham Revenge of the Nerds for the inspiration)

However, some people on #scheme told me that Scheme, due to the proliferation of incompatible implementations, was not one Lisp dialect, but a family of languages called "Scheme" with a common denominator. While I'm all for Germanic languages being a major sub-division of Indo European languages, also with several mutually incomprehensible languages, I'm not sure I want a "Scheme programming language"-family within Lisp. It generally shouldn't happen with "man-made" and computer-understood languages that are more under our control.

As a result, I'd like Spark to remain a single language with only a few implementations, possibly only one for each target virtual machine (e.g: Parrotcode, the JVM, or the .NET CLR, or a C-based interpreter). Spark will be defined and compatible even in its internals, its foreign function interface, and "standard library" (which will also have something more like CPAN is for Perl 5, where every J. Random Hacker can upload their own INI parser, under a different namespace), and core functionality.

Spark will have an open-source source code (GPL-compatible BSD-style or possibly partially Artistic 2.0 in case some of the code is derived from Parrot code), which naturally can be span-off, branches, and forked. However, none of them pose a threat to the fact that the Spark implementation will remain unified.

If someone changes Spark in incompatible ways, it may either die, or forked into a new language. This language will also be Lisp and may be Spark-like but it won't be Spark. Perl 5 which only has one major implementation (`perl5` - currently at `perl-5.10.0`), recently span off kurila which is fork of perl 5 that is incompatible with it and with Perl 5, on purpose. Nevertheless, while Kurila may be considered a a language in the Perl family, it is not Perl 5 any more than Perl 4 , Perl 6 , Sleep or whatever are. So Perl 5 has still not become a family of incompatible implementations.

Another factor that will dissuade people from creating multiple implementations of Spark is that as opposed to Scheme, creating a Spark implementation from scratch is not going to be trivial. It's not that Spark will be needlessly complixified, but that it would be needfully complex to implement to be an expressive, feature-rich and high-quality language.

To quote Bjarne Stroustrup (the creator of C++ ) from his FAQ question about Java

> Much of the relative simplicity of Java is - like for most new languages - partly an illusion and partly a function of its incompleteness. As time passes, Java will grow significantly in size and complexity. It will double or triple in size and grow implementation-dependent extensions or libraries. That is the way every commercially successful language has developed. Just look at any language you consider successful on a large scale. I know of no exceptions, and there are good reasons for this phenomenon. [I wrote this before 2000; now see a preview of Java 1.5.]
>
> — Bjarne Stroustrup *FAQ Question about Java*

(I should note that, like many other FOSS hackers, I normally prefer ANSI C over C++ , and am not a big fan of C++ for most stuff. However, Stroustrup is a wickedly smart guy, and despite whatever faults his language may have, he speaks straight, and what he says here seems to make a lot of sense).

Spark hopefully won't be as complex as C++ is today from the beginning, but will also be more complex than Scheme to allow for better expression and faster development. It also doesn't aim to be an incremental improvement over Scheme (or Common Lisp) which seems to be the case for Arc and Clojure, but rather

something like Perl 5 was to Perl 4 or Perl 6 is to Perl 5 : a paradigm shift, which Lispers and non-Lispers alike will appreciate.

## The first version of Spark will not be the ultimate Lisp

Some features of Common Lisp or other Lisps will be absent in Spark, some things will be harder to do than Common Lisp or even other Lisps or other non-Lisp programming languages, and some things will not work as expected at first (bugs, etc.). A lot of it will be caused due to the fact that the primary author of this document does not consider himself a Scheme expert (and is very far from being a Common Lisp expert) and just likes Lisp, Perl 5 and other languages enough to want to promote them.

As a result, some estoric features of the popular Lisp languages today or some languages that he has not fully investigated yet, won't be available at first. This is expected given his ignorance, enthusiasm and anxiety to get something out of the door first.

While he would still be interested in learning about whatever core library or meta-programmatic features other languages have that may prove useful for the core Spark language (or alternatively cool APIs that you think should be ported to Spark). But he has little patience to learn entire languages "fully" (if learning any non-trivial language fully is indeed possible) before starting to work on Spark. And often ignorance is a virtue.

So the first versions of Spark will still have some room for improvement. Most of it may hopefully be solvable using some meta-syntactic or meta-programming user-land libraries (as is often the case for Lisps and other dynamic languages). As for the rest, we could consider them bad design decisions that still add to the language's colour and make it a bit more interesting to program in than a 100% perfect language. Sometimes perfection is in imperfection.

# Why Lisp Has Been Unpopular

Mark Jason Dominus gives a case study called "Why Lisp Will Never Win" in his "Twelve Views of Mark Jason Dominus". He gives the following awk one-liner:

```
awk 'BEGIN {FS=":"}; $6=="/sbin/nologin" {print $1}' /etc/passwd
```

Which a member of comp.lang.lisp suggested that Lisp should implement something similar to. The response was that "with only a couple of new utility macros & functions", it could become:

```
(with-lines-from-file (line "/etc/passwd")
    (let ((fields (string-split line :fs #\:)))
        (when (string= (aref fields 5) "/sbin/nologin")
            (format t "~A~%" (aref fields 0))))))
```

Which as Dominus notes is a little over 2.5 over the length of the awk program. (but still required these macros). Naturally no one will opt to write it instead. So the problems with Common Lisp is ground-up verbosity, lack of common idioms for commonly performed tasks, and lack of motivation to use it for common, everyday (sometimes even throwaway code).

So what will it look like in Spark?

```
spark -inaF/:/ '(if (= (^F 5) "/sbin/login") (say (^F 0)))' /etc/passwd
```

(we borrowed the `(array idx)` notation from Arc, because an array ref object is overloaded with a method to get the index.

One can find other resources about what makes a language popular here:

1. http://www.paulgraham.com/popular.html - “Being Popular”

2. http://www.paulgraham.com/power.html - “Succinctness is Power”

# Some Implementation Details

These are some implementation details that are still subject to change:

## Virtual Machine

I’m leaning towards making the initial Spark implementation use Parrot as its virtual machine. LLVM was suggested, and while it seems nice and powerful, it requires the compiler front end to be written in C/C++ which will cause the time to market to grow considerably. I don’t rule out a later implementation of Spark to LLVM, but during the initial implementation we would like to change things rapidly and quickly, and C or C\++ will slow us down considerably.

I’m not keen on using the Java Virtual Machine, due to Java’s long historical reputation of being “enterprisey” and non-hacker friendly (see http://www.paulgraham.com/javacover.html ), and due to the fact it has a slow startup time, and that it feels very “sluggish”/non-responsive. Again, I don’t rule out a future port of Spark to the JVM.

The Parrot VM seems very suitable for dynamic languages, and it is progressing nicely. The Parrot languages Page already lists several implementation of Scheme which can serve as the basis for a Spark implementation, so I’d like to start there.

## Spark Licensing

Since we’re building on Parrot, the licence of the non-original code will be the Artistic License 2.0, which is free, open-source, GPL-compatible and somewhere between a weak copyleft licence (e.g: the LGPL) and a permissive licence (e.g: the 2-clause or 3-clause BSD licences). The original code will be written under the MIT/X11 licence, which is a very permissive BSD-style licence that specifically allows sublicensing. To avoid legal confusion, every file should contain an explicit “Licensing” notice to indicate under which license it is.

## Test-driven development

As opposed to Arc, which shipped with no automated tests, Spark will be developed in a Test-driven development fashion. Namely, it will have a comprehensive test suite that will need to fully pass upon any commit to the trunk (or “master” or whatever the main branch is called).

The code of the tests is not expected to be authoritative for how the final version of the language will behave. Rather, some future design decisions will require changing the code of a lot of the tests accordingly.

I still don’t have a clear idea of how to design a lot of “big picture” Spark design decisions. While I believe that design is good, I also think that Spark should be designed incrementally, and that we can expect many design decisions to change. Test-driven development, while accepting the fact that often a lot of testing code will need to be modified, will allow us to do that.

## Documentation

It is our plan to keep documentation for the Spark language using POD, PseudoPod, AsciiDoc or a similar language so people will be able to learn it, without the need to delve into many tests or the core code

itself. The documentation will be kept mostly up-to-date, but we can expect it to grow somewhat out-of-sync with the code.

We're not planning to make exhaustive documentation - for example, the internals of the front-end will not be very well documented, as they will tend to get out-of-sync with the code, and in general the code should be structured to be self-documenting and easy to understand using refactoring.

# Licence