Park - Park is, is not, is too, is not Arc

Shlomi Fish <shlomif@iglu.org.il>

Revision History

Revision 3	21 June 2006	shlomif

Started working on this document after forking the template of an older one.

Table of Contents

Park - Park is, is not, is too, is not Arc

Park [1] is a new dialect of Lisp, based on Paul Graham's Arc effort. Park is in fact a malicious and incompatible fork of Arc. A fork not of the code, which is still not available to the public, but of the concept and motivation.

The origin of the name is with an initial implementation for Arc that I wanted to write for Parrot, called PArc or Parc. However, then I realised I disagree with Graham about some of his language design elements. A few days ago I came with a revelation that it was hopeless waiting for Mr. Graham to actually release either a final spec to the public of Arc or a actual working code. Arc was announced at the LL1 conference at Novemeber 2001. It's been almost 5 years from now, and there's still not a working code or a coherent spec. Furthermore, Graham stopped adding the ideas people sent him to the emails collection containing them (including one that I sent a few years ago, which I know he is aware of because he responded to my message).

Thus enter Park. The name stems from "Parc" but is actually an English world. One can say that Park is a LISP that starts with "P", or that working with it is like a stroll in the Park. Park has the design goals of Arc but:

[1] The "Park - Park is, is not, is too, is not Arc" slogan is temporary. The permanent slogan so far seems like it would be "Park - a LISP that starts with P".

1. Is different and incompatible.
2. Has a working (albeit so far informal) spec.
3. Encourages participation from the community (using such web resources as a Subversion repository, a Wiki, etc.)
4. Encourages people to create and hack on implementations for it.
5. Would try to get version 0.2.0 of the SPEC (the first stable version of the language) out of the door as soon as possible, so people can create implementations that they can rely on.

One note about myself. My name is Shlomi Fish and I'm a software enthusiast and a writer. I am not the ideal person to design Park, but I believe and hope that I have the right attitude. If you happen to know better about somethings, please post your corrections to the mailing lists to the mailing list. Corrections in the form of patches against the Docbook/XML source are preferable. Several of them or even just one that is accepted will get you a Subversion commit access. And naturally everyone can edit the wiki.

Why am I Doing it?

> All language designers are arrogant. Comes with the territory.
>
> —Larry Wall

Why am I doing it? For several reasons:

1. Because it's fun. This is by itself a good reason.
2. Because I found that designing your own language is one of the best ways to learn more about the original languages it is based on. When I designed the Perl dialect "Rindolf", I learned that some features I suggested for it were already doable in Perl 5.
3. Because something good may eventually come out of it. The existing popular dialects of LISP are Common LISP, Scheme (which some people claim is not entirely Lispy) and some domain-specific Lisp dialects like Emacs Lisp and Autolisp, which are not useful for general programming.

   The Common Lisp and Scheme standards still exhibit some relics of the Lisp distant past, where it was expected to run on such now obscure systems as the PDP-10 and ITS or the LISP Machine. They do not correspond to the modern's world "All the world is a VAX" and "Everything on top is a UNIX" reality. (Which has persisted for better or for worse).

   The implementations themselves do little to solve this problem, as they differ in the APIs they provide for such basic mechanisms that programmers of other dynamic languages (like Perl, Python, PHP, Ruby or Tcl) take for granted like Random File I/O, Sockets, Regular Expressions, CGI Programming and Web Automation, Graphical User-interface, Database Connectivity etc.

Why a new Dialect?

Paul Graham explains it very well in the Arc FAQ, and in "Being Popular". Park is Arc done faster, and hopefully also better.

What Park is based on

The following languages have influcened Park:

1. Arc and Scheme. And to some extent Common LISP.

2. Perl 5 and Perl 6.
3. Python.
4. Smalltalk and Ruby.

Elements of Design

Arc is Object Oriented (Everything is an Object)

Paul Graham wrote Why Arc isn't Especially Object-Oriented.. His conclusion for why this is the case are:

> I personally have never needed object-oriented abstractions. Common Lisp has an enormously powerful object system and I've never used it once. I've done a lot of things (e.g. making hash tables full of closures) that would have required object-oriented techniques to do in wimpier languages, but I have never had to use CLOS.
>
> Maybe I'm just stupid, or have worked on some limited subset of applications. There is a danger in designing a language based on one's own experience of programming. But it seems more dangerous to put stuff in that you've never needed because it's thought to be a good idea.

Well, maybe it's because I'm younger, but having worked with Perl 5 extensively, I've used its oject system (and some OO-support modules) a lot. While I have inherited some classes in Perl 5, I also often use an object just to make sure it can be instantiated.

Furthermore, having learned a bit about Perl 6 and Ruby, I believe that making everything an object, can be advantageous because one can more easily inherit from it, and add methods (and multimethods) to constants or regular variables (e.g: “6->my_func();”), or even easily override default behaviour.

So there we go. Note that Park will not be overly-Object-Oriented (OOO). One would be able to declare global functions and variables, and write scripts or modules without wrapping it in Objects. (Read what Damian Conway has to say about the Hello World program in Java for why I don't like it).

Here are some features of the object system:

1. Supports multiple-inheritance.
2. Methods can be added to classes at run-time (like Smalltalk or Ruby).
3. Supports traits (or as the Perl 6 people call them "Roles").
4. The fields of every object will be stored as an associative array, accessible using a special method. (This associative array will also be an object since everything is. I wonder how the Python people have solved it, but it's probably doable.)
5. The member variables of every object will be stored as an associative array, accessible using a special method. (This associative array will also be an object since everything is. I wonder how the Python people have solved it, but it's probably doable.)
6. Similarly to Perl5 and Python (and as opposed to Java, C++ or PHP) there won't be any built-in "public", "protected", "private", etc. access-control for methods of objects by default. Such things can probably be implemented by tweaking the method class and/or the class meta-class, or in different ways completely, but I don't find it appropriate for Park.

7. Multi-methods will be supported. Multimethods are functions that are dispatched according to more than one object based on the best matching multimethod signature. The syntax for this would be the mcall function (short for multi-call):

```
(mcall my-multi-method ([ object1 object2 object3])
    optional-param1 optional-param2 optional-param3
)
```

The standard way to invoke a method is to use the `call` function, which propagates according to only a single argument:

```
(def-method "MyClass::init"
    ((this)
        (set (this sum) 0)
    )
)
(def-method "MyClass::add"
    ((this n)
        (+= (-> this sum) n)
    )
)
(def-method "MyClass::getsum"
    ((this)
        (-> this sum)
    )
)
(set new-summer (new MyClass))
(call add new-summer 5)
(call add new-summer 7)
(call add new-summer 13)
(print (call getsum new-summer) "\n")
```

This will print 24 and a newline.

Note that usually the call would be redundant, and you could use a simple function call. However, such a function call would be subject to other Perl 5-like calling semantics.

Lexical Scoping and Dynamic Scoping

Scheme and Common Lisp which are the most modern popular LISP dialects use lexical scoping, and since Perl 5 adopted it, it has been become very popular and mainstream, outside Lisp.

The default localization of values in Park would be lexical. However, having used Perl 5's `local`, I am aware of several good uses for dynamic scoping in a mostly lexical scoping language. So, there will be a way to localise variables dynamically in Park.

One killer feature like that would be a way to do something like:

```
(do-using-local-state
    (
        (set a "hello")
        (my-func-with-side-effects)
```

```
    )
    (
        # Rest of code here.
    )
)
# a is no longer hello and (my-func-with-side-effects) effects' are reversed.
```

Of course, this feature is still pie-in-the-sky, and I'm waiting for a good analysis of complexity and/or efficient working implementation.

Park would be Fully POSIX

Park will be expected to run on a Unix or a Unix-like system (such as Win32) and will provide APIs for such common POSIX and P-Languages operations as random file I/O, sockets, regular expressions, manipulating ASCII text, Unicode and other encodings. It may also have platform-specific modules for managing processes, creating new processes, etc. (as is the case for Perl 5, for example.)

All of these things will eventually be part of the Park spec, and their APIs would be common to all implementations.

Thoughts about using Monads for Files

> I understood Monads for exactly 5 minutes. Then the understanding left my head. It was too hard for me to keep there.
>
> — An anonymous Perl 5 and Pugs hacker about Haskell's Monads.

As cool as Monads are I'm skeptical about using them in files by default. I know how to use File I/O in Haskell, but still don't understand when Monads are needed. My personal Haskell style so far made no use of them except for file Input and Output.

One will be able to implement Monads in Park and use them for file I/O. There may even be a default Monad class. However, the default file Input/Output API will either be a reflection of the POSIX one, or alternatively something similar in spirit to PerlIO or the Python or Ruby files and sockets I/O classes.

Above this API one can build as many abstractions to his liking, including Monads.

Bootstrapping, Self Hosting and C Hosting

Bootstrapping is the process of implementing a virtual machine by implementing it using an existing virtual machine. So, for example perl 5, CPython, ruby, php and the Tcl interpreter are all written in ANSI C. That way they can be bootstrapped to run on any system for which a C compiler (such as the Open-Source gcc is available).

A related concept is Self-hosting, where a virtual machine's compiler or interpreter is written in itself. So for example, gcc is written in C and requires a C compiler to be built.

For C it is not really a problem, because one can use gcc or a different C compiler to cross-compile enough programs for it to run there conveniently. However,

Constants in Park

> "I am not in Rome."
>
> — An anonymous designer of his own DOS 8086 Assembly written in itself, and used to write other programs, in which numbers that start with 0 are hexadecimal.

I encountered some Scheme implementations in which identifiers were case-sensitive, and some in which they were case-insensitive. To match what is the norm nowadays, and becase I like it better, Park requires case-sensitive identifiers.

Numbers in Park can be decimal, octal (if they start with "0"), or hexadecimal (if they start with 0x). Following Perl's lead, one will be able to use underscores (_) within numbers (e.g: `100_000` instead of `100000`).

Strings, Quoting and Interpolation

String Constants

String constants will be written in a large correspondence to the C, Perl 5 and Perl 6 conventions. For example, a "\n" will designate a newline. The exact type and quoting of the string will be determined by a few optional Perl 6-like modifiers. Here are a few examples:

```
# The (qq ... ) is not really needed but it doesn't hurt.
(my mystring (qq "Hello there, Hackers! Welcome to Park"))
(print mystring "\n")
(my regex (m :p5 "H..kers"))
(say (=~ mystring regex)) # Prints "Hackers"
(my conv (tr "H" "B"))
(say (=~ mystring conv)) # Prints "Bello there, Backers!..."
(my newstring mystring)
(=~! newstring conv)
# newstring is now "Bello there, Backers!..."
# (You gotta love S-expressions.)
```

String Interpolation

There was a very interesting presentation in an Israeli Perl Mongers meeting about strings, quoting and interpolation in other languages and Perl. The presentation noted among else that in other languages one has to write relatively ugly code like `"Hello" & Name & "! What's up?"` rather than the Perl `"Hello $Name! What's up?"`.

In Perl 5 the interpreter has some heuristics for determining the end of the interpolated expression, which can be any arbitrary Perl expression. Perl 6 goes one step further and parses the string and translates it into Perl 6 code, so one can safely include arbitrarliy complex code there. (In a similar fashion to the Korn shell "$(...)" construct).

What I plan is to have such interpolation in Park as well for some types of strings. The syntax will look something like that:

```
(my i 5 j 7)
(print "${i} + ${j} = $(+ i j)\n")
```

As expected this prints "5 + 7 = 12" (followed by a newline).

Another thing I'd like to have is shell or Perl 5-like here documents (`&&"EOF" ... EOF`)