The Alonzo Language Specification

Alonzo is a language heavily based on Lambda Calculus' syntax, and minimalism. Alonzo also uses Haskell's idea of purity, although not quite to that extent. And because of Alonzo's use of Scheme-eschew style macros, very little cannot be implemented in the language itself. Alonzo does not aim to be Object-Oriented in any way, only supporting algebraic data types (for example, Haskell's Option type) and enumerations. Functions are values, and are proper lambdas. All functions are curried automatically, as in Haskell, except that missing arguments are represented as an underscore. There are no operators defined by the language, just functions, although in the standard library, infix macros are defined for operators. Lists are not built into the language, but instead are implemented in it. There are no loops of any kind, except the usual map, filter, reject, each, etc. Recursion is highly recommended for tasks that require indefinite amounts of looping. Since there are no operators, there are no special characters that you cannot put in variable names. Even if you define an infix macro, it will be delimited by whitespace. Alonzo uses Scheme-like brace, paren, and curly brace inter polarity, but Alonzo has some best practices when using one or the other type of brace, to give it a more C-like feel, but otherwise is pretty much exactly the same as lambda calculus. Functions only ever take one argument, and return functions that take the other arguments. There are not to be any extensions to the original runtime after the language is finished. Any additional features *will* be added to the standard library as macros, always. The language is implemented in Rust for maximum portability. Any operating system should be able to use it. Alonzo compiles to LLVM, and is parsed by a Top Down Operator Precedence (Based on Douglas Crockford's article). There will never, in any reimplementation of the language be any other features than these in the language:

1. True Lambdas
2. Variable Assignment
3. Strings
4. Macros

> **Note**: Comments are a macro, and so are lists and data types and enumerations. Almost all usual language features are a macro or a function.

1. Functions

Lambdas are essential for any Turing-complete language, as lambda calculus can prove. The definition of a lambda is a function that has no name, but instead can be passed around as easily as a number or boolean. A lambda, when created, captures its surrounding environment in a closure. Alonzo has been heavily inspired by Lambda Calculus, as even professed by its name. The syntax we have chosen is very similar to the lambda calculus syntax, but slightly influenced by C to make it more readable to the average programmer. Here is an example of a normal lambda in Alonzo:

```
{λx . λy . λz .
  xyz ↦ +(x, y, z)
  *(xyz, 9)
}(1, 2, 3)
```

As you may have noticed, those λ characters require unicode. This is not necessary if your editor or environment is not supportive to that type of unicode programming, but it is recommended, because it makes much more clear your intentions to make a lambda similar to that of a lambda calculus one. Also note that functions are curried, as professed by the lambda syntax, just as in Lambda Calculus. Parenthesis and curly braces and square brackets are all completely equivalent, but to avoid a lispy feel, we mostly use curly braces to surround blocks, parenthesis to call functions and wrap expressions, and square brackets to go around the list macro. This is so that users can use most of the visual cues they are used to. Dots are like open curlys that are all closed by the last curly. This is very convenient. Without dots, Alonzo's syntax would be flooded with brackets, curlys, and braces! Here is the non-unicode version:

```
{\\x . \\y . \\z .
  xyz := +(x, y, z)
  *(xyz, 9)
}(1, 2, 3)
```

Here is an example of currying:

```
+1 ↦ {λx . λy . λz .
  +(x, y, z)
}(1, _, _)
+4 ↦  +1(4, _)
+9 ↦  +4(9) ; => 14 (:
+3nums ↦  {λx . λy . λz .
  +(x, y, z)
}
+3nums(1, 2, 3) ; => 6
```

Lambdas with no arguments use an underscore in place of the argument, such as `{λ_ . 1 + 1}.` Functions are called with no arguments by using empty parens (`()`)

1.1 Side-Effects (Put the Poop in The Box)

Side effects are not allowed in normal functions at all. All functions must return the same thing when given the same arguments as previously. This is similar to Haskell, except that instead of using monads, which I find very confusing and hard to understand for beginners, we use a sort of 1/2 monad:

```
main ↦ {do .
  println "Hello, World!"
}
```

Side effects are only allowed in a do-block, and do blocks can only be assigned to the main variable, which is called immediately if it is there. The last return value of the program is otherwise used. Which brings me to one point: The last expression in a do-block or lambda is the return value, which is highly convenient for functional programming.

1.2 The Poop in The Box

So, we can do side-effects safely and cleanly (for the most part) but, what can we do, and how twill it work? Lets see an example. We will make a guessing game. The computer will take a random number (a side-effect, because it does not always return the same thing), feed it into a function that formats the question. Then, back in main, the rest of the work is done:

```
main ↦ {do .
   println "Hello, User."
   println "In this game, you are to guess the computer's number,
from 1 to 100."
   r ↦ &rand 1, 100 ; The ampersand symbol signifies side-effect
functions. These functions cannot be called outside of the {do.}
construct.
   println format(r)
   print ">>"
   in ↦ &read()
```

```
    {{if stoi(in) == r println "Horray! You guessed my number!"}
     {else println "No! That's not my number!"}}
}
```

As you can see, there is no need to be able to do many side-effects in many different places in your system. We have studied the coding patterns of many well-respected programmers, and even in languages that allow side-effects anywhere, they are mostly centered around a single section of the code that then dispatches and chains together many utilities. If this idea of limited side effects scares you, go look at some real-world recommendations for modular code! Plus, this idea of mostly pure functions makes the whole thing easier to unit test.

> **Note:** Functions are not lazy, because we have macros, but lists are very lazy, since they are macros. If you want laziness use macros, and for eager evaluation, use functions. All language constructs are built with macros and lambdas, so by nature everything is lazy. Have fun being lazy!

2. Variable Assignment

Variables are immutable, and all structures, such as arrays and datatypes are therefor immutable. Alonzo uses Lexical scoping with variables and functions. Variables are assigned like this:

```
set('foobar', 2)
```

but in the standard library a macro for assignment is defined to make it easier and clearer:

```
foobar ↦ 2
```

The format is `<identifier> := <expression>` where an identifier is any sequence of characters, symbols, or numbers that is not already used.

Variables can have numbers that come first, because numbers are defined inside the language anyway. All infix operators that you see are macros. So are lists, numbers, conditionals, and booleans. To make a contract of a type of a variable just use a special form of comment:

```
foobar::int
foobar ↦ 2
```

This special comment only comments one identifier, which it then associates with the identifier on the other side, and then it checks to make sure that the 2 is an Int. Since variables are immutable, this isn't really that much use, but for people who want it, it's there. (written as a macro).

2.1 Type Checking Example

Type checking is not really *necessary* for the language, but for the comfort and security of the very few people that have been coming from statically-typed languages, it might be of some familiarity, and anything familiar is good when learning a language as strange as this. Plus, static types, when dynamic under the hood, acting as just inline documentation and extra checks, can actually add extra expressiveness to a language! Here is a good example:

```
; concat: A function that takes two strings and returns another
string that is those two strings together
```

This is fine documentation, and very clear and concise for english, but then, look at this:

```
concat::string(A), string(B) -> string(C)
; where C = A + B
```

The first line of code is actually evaluated by a macro, which sets up a wrapper function that does type checks around the original function, using the names supplied as the variable names for the arguments for the wrapper function. The second line, a comment, expresses some extra human-read information about that function. This also works for variables, since named functions *are* variables that just happen to have a lambda in them. But since variables cannot be reassigned, type checking on variables is rarely needed. If you need to change the value of something, make it an argument to a function, and recur on that function with the changed value. There are many options to change values of names, but you will be surprised how little and rarely you need to change the value of something. Inside the main function, inside the do macro that is assigned to the main function, you can change the values of variables and produce side-effects, but there and nowhere else. As Yehuda Katz and Tom Dale say:

> ... put the poop in the box, so none has to see it ...

That is a paraphrase, of course.

3. Strings

Strings are created between " or '. There is no string templating or concatenation or substitution or slice built in, but all that is possible inside the standard library, with macros. Strings can be indexed though, and indexes that go off the end of the string return an empty string, and setting of the end concatenates the character. You can only get and set by characters in strings in Alonzo.

```
get 0 , "Hello, World!"
get 5 , 'Hello, World!'
```

3.1 An Example of Concatenating Strings

Concatenating strings is built into the standard library, but since The whole point of the Alonzo language is to see how much we can do with the bare minimum, lets see how we concatenate strings. We will use the get, set and numbers from the standard library though. This is basically just an interesting thought exercise to introduce you to the Alonzo language.

> **Note**: In Alonzo, it is recommended to use unicode for the λ symbols. It is recommended that you place the `\\` word for autocomplete as λ. And the `:=` for autocomplete as ↦. Unicode is more expressive than plain ascii, so this is much clearer, plus looking more like the lambda-calculus that Alonzo is based on.

```
concat ↦ {λs1 . λs2
       end-of-str1 ↦ str_length(s1)
       rec ↦ {λon-char . λs
              {{if on-char < str_length(s2)
                     rec on-char + 1, set(end-of-str1 + on-char,
                                      get(on-char, s2),
                                      s)}
              {else s}}
       }
       rec(s1)
}
```

4. Macros

Macros are the central feature in Alonzo, because they are the reason that it is possible to write such a tiny language and yet get a readable, high-level functional language. Scheme has macros, but they are not obvious about their use, and not very easy to read their definition. So, I have decided to use the Rust or Sweet.js style syntax for macros. Macros are going to be one of the most sophisticated features of this language. Macros in Alonzo are very simple pattern matches, with capitalized variables for unknown peices, which get put into those variables. Macros are lazy, and get the structure of their arguments to match on, not the argument result. Macro bodies are the transformed code, with no code that you do not want put into the translation. Alonzo is evaluated like this:

4.1 Macro Syntax

Macros use a very intentionally simple syntax. Here is a crude implementation of `+`.

```
macro + {
   binary rule { A:int _ B:int } => {
         add(A, B)
   }
   unary rule { _ A } => {      ; The underscore is for matching the
actual macro's name. You can also do macros that match things that
do not contain their names, for useful things like code
optimization.
         abs(A)
   }
}
```

5. FAQ

5.1.1 How do I identify data types?

Built-In datatypes can be identified by the type function, but user-defined types cannot.

5.1.2 Will Alonzo be able to load files in the future?

File loading will probably not be implemented, because of the certain imperativeness and unsafeness that comes with that. In the future, they might be added though.

5.1.3 Why the Rust Programming Language vs. C or C++?

Rust is a very modern, flexible, and functional programming language, with closures, lambdas, and many other good features, as well as direct bindings for LLVM, and A good package manager and fast benchmark speed. It is also very robust, and type-safe, as well as avoiding all the dangerous pitfalls of C. It is also as fast or faster then C. C++ is also far to strange, dangerous, imperative, and just plain impalpable.

5.1.4 Will Alonzo implement feature X in Y time?

No. Not ever. Implement it yourself. We have macros, and a very flexible syntax. And, do you really need X??

5.1.5 How do I learn Alonzo?

Since the majority of programmers aren't functional programmers, and this is a very hard-core functional language, as well as not being marketed very well, and being very small, you can either contact me, put up an issue, write a tutorial yourself, read the rust code, or read the standard library.