λ Informal description of the λanguage

Before anything, we should have a clear picture about what we're trying to achieve. It's a good idea to put together a rigorous description of the grammar, but I'm going to keep things more informal in this tutorial, so here is the λanguage by examples:

```
# this is a comment

println("Hello World!");

println(2 + 3 * 4);

# functions are introduced with `lambda` or `λ`
fib = lambda (n) if n < 2 then n else fib(n - 1) + fi

println(fib(15));

print-range = λ(a, b)             # `λ` is synonym to
                if a <= b then {  # `then` here is op
                  print(a);
                  if a + 1 <= b {
                    print(", ");
                    print-range(a + 1, b);
                  } else println("");        # newlin
                };
print-range(1, 5);
```

> Note above that identifier names can contain the minus character (`print-range`). That's a matter of personal taste: I always put spaces around operators, I don't like much camelCaseNames and the dash is nicer than the underscore. The nice thing about writing your own language is that you can do it as you like it. :)

The output is:

```
Hello World!
14
610
1, 2, 3, 4, 5
```

Welcome! (login)

How to implement a PL

- » Introduction
- » [λanguage description]
- » Writing a parser
  - ✓ Input stream
  - ✓ Token stream
  - ✓ The AST
  - ✓ The parser
- » Simple interpreter
  - ✓ Test what we have
  - ✓ Adding new constructs
  - ✓ How fast are we?
- » CPS Evaluator
  - ✓ Guarding the stack
  - ✓ Continuations
    - ✓ Yield (advanced)
- » Compiling to JS
  - ✓ JS code generator
  - ✓ CPS transformer
    - ✓ Samples
    - ✓ Improvements
  - ✓ Optimizer
- » Wrapping up
- » Real samples
  - ✓ Primitives
  - ✓ catDir
  - ✓ copyTree sequential
  - ✓ copyTree parallel
  - ✓ In fairness to Node
  - ✓ Error handling

The λanguage looks a bit like JavaScript, but it's different. First, there are no statements, only expressions. An expression returns a value and can be used in place of any other expression. Semicolons are required to separate expressions in a “sequence”. The curly brackets, `{` and `}`, create such a sequence, and it's itself an expression. Its value is what the last expression evaluates to. The following is a valid program:

```
a = {
  fib(10);  # has no side-effects, but it's computed
  fib(15)   # the last semicolon can be missing
};
print(a); # prints 610
```

Functions are introduced with one of the keywords `lambda` or `λ` (they are synonyms). After the keyword there must be a (possibly empty) parenthesized list of variable names separated with commas, like in JavaScript — these are the argument names. The function body is a single expression, but it can be a sequence wrapped in `{…}`. There is no `return` statement (there are *no statements*) — the last expression evaluated in a function gives the value to return to its caller.

There is no `var`. To introduce new variables, you can use what JavaScripters call “IIFE”. Use a `lambda`, declare variables as arguments. Variables have function scope, and functions are closures — like in JavaScript.

Even `if` is itself an expression. In JavaScript you'd get that effect with the ternary operator:

```
a = foo() ? bar() : baz();           // JavaScript
a = if foo() then bar() else baz();  # λanguage
```

The `then` keyword is optional when the branch starts with an open bracket (`{`), as you can see in `print-range` above. Otherwise it is required. The `else` keyword is required if the alternative branch is present. Again, `then` and `else` take as body a single expression, but you can `{`group`}` multiple expressions by using brackets and semicolons. When the `else` branch is missing and the condition is `false`, the result of the `if` expression is `false`. Speaking of which, `false` is a keyword which denotes the only falsy value in our λanguage:

```
if foo() then print("OK");
```

will print "OK" if and only if the result of `foo()` is NOT `false`. There's also a `true` keyword for completion, but really everything which is not `false` (in terms of JavaScript's `===` operator) will be

interpreted as true in conditionals (including the number `0` and the empty string `""`).

Also note above that there is no point to demand parentheses around an `if`'s condition. It's no error if you add them, though, as an open paren starts an expression — but they're just superfluous.

A whole program is parsed as if it were embedded in curly brackets, therefore you need to place a semicolon after each expression. The last expression can be an exception.

Well, that's our tiny λanguage. It's not necessarily a good one. The syntax looks cute, but it has its traps. There are a lot of missing features, like objects or arrays; we don't concentrate on them because they're not essential for our journey. If you understand all this material, you'll be able to implement those easily.

In the next section we'll write a parser for this λanguage.
