+

{lambda talk}

Alain Marty Engineer Architect
Villeneuve de la Raho, France
marty.alain@free.fr

ABSTRACT

The {lambda way} project is a web application built on two engines, {lambda talk} and {lambda tank}. {lambda talk} is a purely functional language unifying authoring, styling and scripting in a single and coherent Lisp-like syntax, working in {lambda tank}, a tiny wiki built as a thin overlay on top of any web browser. In this paper we forget {lambda tank}, mainly a PHP engine managing text files on the server side, and progressively introduce {lambda talk}, a Javascript engine evaluating code in realtime on the client side. The making of {lambda talk} is done in three stages:

- 1) we define the minimal set of rules making {lambda talk} a programming language, *complete even if unusable*,
- 2) we progressively add numbers, operators, data and control structures making {lambda talk} *more usable*,
- 3) we finally build on the browsers' full functionalities a set of *libraries* making {lambda talk} *usable and much more efficient*.

As a guilding line, at each stage, we compute with a total precision the factorial of **5** and **50**:

```
5!  = 120
50! = 30414093201713378043612608166064
      768844377641568960512000000000000
```

In a last section, APPENDICE, some explanations are given on the {lambda talk}'s Javascript implementation.

KEYWORDS

- Information systems~Wikis
- Theory of computation~Regular languages
- Theory of computation~Lambda calculus
- Software and its engineering~Functional languages
- Software and its engineering~Extensible Markup Language

INTRODUCTION

{lambda talk} expressions are written in an editor frame, evaluated in real time, displayed in the wiki's viewer frame, then saved and published on the WEB. « *A wiki is a web application which allows collaborative modification, extension, or deletion of its content and structure.*[1] » The father of this concept, Ward Cunningham[2], gives a simple and clear introduction to {lambda talk}:

- **1)** Away from curly braces {} *words are just words*:

```
Hello World!
-> Hello World!
```

- **2)** Expressions are written in a *prefix notation*:

```
2+3 is equal to {b {+ 2 3}}
-> 2+3 is equal to 5
```

- **3)** Functions are created with *lambda* and named with *def*:

```
{def SMART_ADD
 {lambda {:a :b}
  :a+:b is equal to {b {+ :a :b}}}}
-> SMART_ADD

{SMART_ADD 2 3}
-> 2+3 is equal to 5
```

These examples use the "+" Math operator, the "b" HTML/CSS markup operator and Javascript numbers. In the following we want to introduce {lambda talk} built upon the deepest foundation possible, *simple words*, and we will ignore everything but words until the third section.

1. WORDS

We present the structure and evaluation of a {lambda talk} expression.

1.1. expressions

{lambda talk} is mainly built on three rules freely inspired by the *λ-calculus*[3]. An expression is defined recursively as follows:

```
expression is [word|abstraction|application]*
```

where

```
1) word         is [^\s{}]*
2) abstraction  is {lambda {word*} expression}
3) application  is {abstraction expression}
```

An expression is made of `words`, `abstractions` and `applications` where 1) a `word` is any character except spaces "`\s`" and curly braces "`{}`", 2) an `abstraction` is the "process" (called a *function*) selecting a sequence of `words` (called *arguments*) in an `expression` (called *body*), 3) an `application` is the process calling an `abstraction` to replace selected `words` by some other `words` (called *values*).

The evaluation follows these rules:

1. words are not evaluated,
2. abstractions are evaluated before applications,
3. an abstraction is evaluated to a single word, a reference to an anonymous a function stored in a global dictionary, initially empty,

4. an application is progressively evaluated from inside out, to a sequence of words,
5. the evaluation stops when all expressions have been reduced to a sequence of words.

What can we do with that?

1.1.1. words

```
Hello World
-> Hello World
```

Words are not evaluated and are displayed as they are.

1.1.2. abstraction

```
{lambda {o a} oh happy day!}
-> _LAMB_6
```

The abstraction selects `o` and `a` as characters whose occurences in the expression `oh happy day!` are to be replaced by some future values, and returns the reference to an anonymous function.

1.1.3. application (1)

```
{{lambda {o a} oh happy day!} oOOOo aaAAaa}
-> oOOOoh haaAAaappy daaAAaay!
```

The abstraction is defined and immediately called. The abstraction is first evaluated to a word, say `_LAMB_6`, the application `{_LAMB_6 oOOOo aaAAaa}` gets the given values, calls the abstraction which makes the substitution and returns the result, `oOOOoh haaAAaappy daaAAaay!`.

1.1.4. application (2)

```
{{lambda {z}
 {z {lambda {x y} x}}}
  {{lambda {x y z}
   {z x y}} Hello World}}
-> Hello
```

```
{{lambda {z}
 {z {lambda {x y} y}}}
  {{lambda {x y z}
   {z x y}} Hello World}}
-> World
```

These expressions return respectively the first and the second words of `Hello World`, recalling a pair and its accessors. Let's trace the evaluation leading to Hello:

- 1) Nested lambdas are first evaluated:

```
1: {{lambda {z} {z {lambda {x y} x}}}
    {{lambda {x y z} {z x y}} Hello World}}
2: {{lambda {z} {z _LAMB_1}}
    {_LAMB_2 Hello World}}
3: {_LAMB_3 {_LAMB_2 Hello World}}
where
  _LAMB_1 replaces {lambda {x y} x}
  _LAMB_2 replaces {lambda {x y z} {z x y}}
  _LAMB_3 replaces {lambda {z} {z f1}}
```

- 2) Then simple forms are evaluated:

```
1: {_LAMB_3 {_LAMB_2 Hello World}}
2: {_LAMB_3 {{lambda {x y z} {z x y}} Hello
World}}
```

`{{lambda {x y z} {z x y}} Hello World}` is a *partial application* replacing "x" and "y" by "Hello" and "World", creating a new lambda waiting for the third value, `{lambda {z} {z Hello World}}`, and returning a reference, `_LAMB_4`.

```
3: {_LAMB_3 _LAMB_4}
4: {{lambda {z} {z _LAMB_1}} _LAMB_4}
5: {_LAMB_4 _LAMB_1}
6: {{lambda {z} {z Hello World}} _LAMB_1}
7: {_LAMB_1 Hello World}
8: {{lambda {x y} x} Hello World}
9: Hello
```

To sum up on lambdas:

- `1) lambdas` are *first class* functions,
- `2) lambdas` accept *partial function application*: a lambda called with a number of values lesser than its arity memorizes the given values and returns a new lambda waiting for the rest.
- `3) lambdas` *don't create closures*, inner lambdas have no access to outer lambdas' arguments, there is no lexical scoping, no nested environments, no free variables.

Lambdas are pure black boxes, independant of any context, as are mathematical functions.

1.1.5. application (3)

```
{{lambda {n} {{lambda {g n} {g g n}} {lambda {g
n} {{lambda {p t f g n} {{{p n} {{lambda {x y z}
{z x y}} t f}} g n}} {lambda {n} {{lambda {n} {n
{lambda {x} {lambda {z} {z {lambda {x y} y}}}}}
{lambda {z} {z {lambda {x y} x}}}}} n}} {lambda
{g n} {{lambda {n f x} {f {{n f} x}}} {lambda {f
x} x}}} {lambda {g n} {{lambda {n m f} {m {n
f}}} n {g g {{lambda {n} {{lambda {z} {z {lambda
{x y} x}}} {{n {lambda {p} {{lambda {x y z} {z x
y}} {{lambda {z} {z {lambda {x y} y}}} p}
{{lambda {n f x} {f {{n f} x}}} {{lambda {z} {z
{lambda {x y} y}}} p}}}}} {{lambda {x y z} {z x
y}} {lambda {f x} x} {lambda {f x} x}}}}} n}}}}
g n}} n}} {lambda {f x} {f {f {f {f {f x}}}}}}}
-> _LAMB_167
```

At this point, *you should believe* that **this unreadble expression** evaluated to an anonymous function *is* the factorial of **5**, **5! = 120**, computed using its recursive definition:

```
fac(n) = 1 if n == 0 else fac(n) = n*fac(n-1)
```

1.2. names

In order to make code more readable we introduce a second special form `{def word expression}`, with which we will populate the *global dictionary* with *constants* and give *names* to anonymous functions. Let's rewrite with names the previous examples.

1.2.1. words

Any sequence of words can be given a name:

```
{def HI Hello World}
-> HI
```

```
HI {HI}
-> HI Hello World
```

Note that the word **`HI`** out of curly braces {} is not evaluated. Remember that, in a spreadsheet, one must write **`=PI()`** to get the value associated to **PI**.

1.2.2. abstractions

An anonymous functions can be given a name:

```
{def GOOD_DAY
 {lambda {:o :a} :oh h:appy day!}}
-> GOOD_DAY
```

1.2.3. application (1)

That makes several applications easier:

```
{GOOD_DAY oOOOo aaAaa}
-> oOOOoh haaAaappy day!

{GOOD_DAY ♠ ♥}
-> ♠h h♥ppy day!
```

Note: arguments and their occurences in the function's body have been prefixed with a colon "**:**". Doing so prevents unintentional substitutions in the function's body, for instance the word **day** hasn't been replaced by **daaAaay** or **d ♥ y**. Escaping/marking arguments, for instance prefixing them with a colon "**`:`**", is highly recommended if not always mandatory. We will do it systematically and we add this constraint to the previous rules: « *In lambda expressions arguments **`arg`** must at least be tagged by some escaping character, for instance **`:arg`**, or for a better security, bracketted between two, for instance **`:arg:`***.

1.2.4. application (2)

```
{def CONS {lambda {:x :y :z} {:z :x :y}}}
-> CONS
{def CAR  {lambda {:z} {:z {lambda {:x :y}
:x}}}}
-> CAR
{def CDR  {lambda {:z} {:z {lambda {:x :y}
:y}}}}
-> CDR

{CAR {CONS Hello World}}
-> Hello
{CDR {CONS Hello World}}
-> World
```

In fact we just have built and used a *pair* and its *accessors*. Where LISP[4] and SCHEME[5] use a closure to define **`cons`**:

```
(def cons (lambda (x y) (lambda (z) (z x y))))
(def car  (lambda (z)   (z (lambda (x y) x))))
(def cdr  (lambda (z)   (z (lambda (x y) y))))
```

{lambda talk} uses partial application. There is no outer environment storing accessible values, values are stored inside lambdas.

1.2.5. application (3)

We rewrite the example 1.1.5. using two names:

```
{def FOO {lambda {:n} {{lambda {:g :n} {:g :g
:n}} {lambda {:g :n} {{lambda {:b :t :f :g :n}
{{{:b :n} {{lambda {:x :y :z} {:z :x :y}} :t
:f}} :g :n}} {lambda {:n} {{lambda {:n} {:n
{lambda {:x} {lambda {:z} {:z {lambda {:x :y}
:y}}}} {lambda {:z} {:z {lambda {:x :y} :x}}}}}
:n}} {lambda {:g :n} {{lambda {:n :f :x} {:f
{{:n :f} :x}}} {lambda {:f :x} :x}}} {lambda {:g
:n} {{lambda {:n :m :f} {:m {:n :f}}} :n {:g :g
{{lambda {:n} {{lambda {:z} {:z {lambda {:x :y}
:x}}} {{:n {lambda {:p} {{lambda {:x :y :z} {:z
:x :y}} {{lambda {:z} {:z {lambda {:x :y} :y}}}
:p} {{lambda {:n :f :x} {:f {{:n :f} :x}}}
{{lambda {:z} {:z {lambda {:x :y} :y}}} :p}}}}}
{{lambda {:x :y :z} {:z :x :y}} {lambda {:f :x}
:x} {lambda {:f :x} :x}}}}} :n}}}} :g :n}} :n}}
}
-> FOO

{def BAR
 {lambda {:f :x} {:f {:f {:f {:f {:f :x}}}}}}}
-> BAR

{FOO BAR }
-> _LAMB_8
```

We notice that **`FOO`** is applied to **`BAR`**, an anonymous function applying *5 times* **`:f`** to **`:x`**. We can now better understand that the result is a reference to an anonymous function which might be associated to the factorial of **5**. And we guess that applying **50 times `:f`** to **`:x`** would lead to an anonymous function associated to the factorial of **50** ... provided we had a huge memory and thousands years before us!

Concluding this first section we note that, until now, {lambda talk} knows nothing but text substitution and that the dictionary contains no built-in primitive. In the following section, *still without using any Javascript Math object*, we progressively build **numbers**, **operators**, **data** and **control structures** to compute **5!** and **50!**.

2. NUMBERS

Following *"Collected Lambda Calculus Functions"*[6] we progressively add numbers, operators, data and control structures.

2.1. numbers

We define the so-called *Church numbers*:

```
{def ZERO  {lambda {:f :x} :x}}
-> ZERO
{def ONE   {lambda {:f :x} {:f :x}}}
-> ONE
{def TWO   {lambda {:f :x} {:f {:f :x}}}}
-> TWO
{def THREE {lambda {:f :x} {:f {:f {:f :x}}}}}
-> THREE
{def FOUR  {lambda {:f :x} {:f {:f {:f {:f
:x}}}}}}
-> FOUR
{def FIVE  {lambda {:f :x} {:f {:f {:f {:f {:f
:x}}}}}}}
-> FIVE
```

Applied to a couple of any words, we get strange things:

```
{ZERO . .} -> .
{ONE  . .} -> (. .)
```

```
{TWO  . .} -> (. (. .))
{FIVE . .} -> (. (. (. (. (. .)))))
```

We define the function `CHURCH` which translates Church numbers in a more familiar shape:

```
{def CHURCH
 {lambda {:n}
  {{:n {lambda {:x} {+ :x 1}}} 0}}}
-> CHURCH
{CHURCH ZERO} -> 0
{CHURCH ONE}  -> 1
{CHURCH FIVE} -> 5
```

Note: the `CHURCH` function is built on numbers, **[0,1]**, and a function, **'+'**, coming with Javascript, which are not supposed to exist at this point. Consider that it's only for readability.

2.2. operators

Based on Church numbers, which are **iterators by themselves**, we can easily define and test a first set of operators:

```
{def SUCC {lambda {:n :f :x} {:f {{:n :f} :x}}}}
-> SUCC
{def ADD {lambda {:n :m :f :x} {{:n :f} {{:m :f}
:x}}}}
-> ADD
{def MUL {lambda {:n :m :f} {:m {:n :f}}}}
-> MUL
{def POWER {lambda {:n :m} {:m :n}}}
-> POWER

{CHURCH {SUCC ZERO}}        -> 1
{CHURCH {SUCC ONE}}         -> 2
{CHURCH {SUCC THREE}}       -> 3
{CHURCH {ADD TWO THREE}}    -> 5  // 2+3
{CHURCH {MUL TWO THREE}}    -> 6  // 2*3
{CHURCH {POWER THREE TWO}}  -> 9  // 3^2
```

Building "opposite" functions like `PRED`, `SUBTRACT`, `DIVIDE` is not so easy - and Church himself avoided them in the primitive version of λ-calculus. The answer was given by Stephen Cole Kleene[7], the father of Regular Expressions: *Church numbers can be used to **iterate** and pairs to **aggregate***. This is how:

- we define a function `PRED.PAIR` getting a pair `[a,a]` and returning a pair `[a,a+1]`,
- the function `PRED` computes `n` iterations of `PRED.PAIR` starting on the pair `[0,0]` and leading to the pair `[n-1,n]` and returns the first, `n-1`:

```
{def PRED.PAIR {lambda {:p}
 {CONS {CDR :p} {SUCC {CDR :p}}}}}
-> PRED.PAIR
{def PRED {lambda {:n}
 {CAR {{:n PRED.PAIR} {CONS ZERO ZERO}}}}}
-> PRED

{CHURCH {PRED FIVE}}
-> 4
```

2.3. « To Iterate is Human, ...

We already have all what is needed to evaluate complex expressions like **1*2*3*...*n**. Inspired by the `PRED` operator:

- we define a function `ITER.PAIR` getting a pair `[a,b]` and returning a pair `[a+1,a*b]`,
- the function `ITER` computes `n` iterations of `ITER.PAIR`, starting on the pair `[1,1]` and leading to the pair `[n,n!]` and returns the second, `n!`

```
{def ITER
 {def ITER.PAIR
  {lambda {:p}
   {CONS {SUCC {CAR :p}}
         {MUL {CAR :p} {CDR :p}}}}}}
 {lambda {:n}
  {CDR {{:n ITER.PAIR} {CONS ONE ONE}}}}}
-> ITER

{CHURCH {ITER TWO}}   -> 2
{CHURCH {ITER THREE}} -> 6
{CHURCH {ITER FOUR}}  -> 24
{CHURCH {ITER FIVE}}  -> 120
```

2.4. ... to Recurse, Divine »[8]

If we want to define the factorial using its recursive mathematical definition:

```
fac(n) = 1 if n == 0 else fac(n) = n*fac(n-1)
```

we need to build a few boolean operators:

```
{def TRUE {lambda {:z} {:z {lambda {:x :y}
:x}}}}
-> TRUE
{def FALSE {lambda {:z} {:z {lambda {:x :y}
:y}}}}
-> FALSE
{def IF {lambda {:x :y :z} {:z :x :y}}}
-> IF
{def ISZERO {lambda {:n}
  {:n {lambda {:x} FALSE} TRUE}}}
-> ISZERO
```

Note that `TRUE, FALSE, IF` are aliases to `CAR, CDR, CONS`.

Here is the tricky part! We remember that all expressions except abstractions are evaluated *eagerly*: functions' arguments are *called by value* and not *called by name*. Inside the `IF` function every arguments are evaluated before the call and this would lead to an *infinite loop* in a recursive process. A workaround is to use abstraction to introduce *manually* some kind of lazyness. This is an answer:

```
{def FAC
 {lambda {:n}
  {{lambda {:b :t :f :n}
   {{{:b :n} {IF :t :f}} :n}}
    {lambda {:n} {ISZERO :n}}    -> :b
    {lambda {:n} ONE}            -> :t
    {lambda {:n}
      {MUL :n {FAC {PRED :n}}}} -> :f
    :n                           -> :n
}}}
-> FAC

{CHURCH {FAC FIVE}}
-> 120
```

We can see that expressions `{ISZERO :n}` and `{MUL :n {FAC {PRED :n}}}` are hidden via lambdas behind names `:b` and `:f`. As long as `{:b :n}` is evaluated to `false`, `{IF :t :f}` returns the word `:f` which is **then** evaluated to `{MUL :n {FAC {PRED :n}}}` and the process recurses until zero, leading to `{* 5 {* 4 {* 3 {* 2 1}}}} = 120`.

Let's introduce some **Y-combinator** making recursive an almost recursive function:

```
{def Y {lambda {:g :n} {:g :g :n}}} -> Y

{def IFTHENELSE {lambda {:b :t :f :g :n}
  {{{:b :n} {IF :t :f}} :g :n} }}
-> IFTHENELSE

{def ALMOST_FAC {lambda {:g :n}
  {IFTHENELSE
   {lambda {:n} {ISZERO :n}}
   {lambda {:g :n} ONE}
   {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
   :g :n }}}
-> ALMOST_FAC

{CHURCH {Y ALMOST_FAC FIVE}}
-> 120
```

Let's mix the both:

```
{def YFAC {lambda {:n}
 {{lambda {:g :n} {:g :g :n}}
 {lambda {:g :n}
  {IFTHENELSE
   {lambda {:n} {ISZERO :n}}
   {lambda {:g :n} ONE}
   {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
   :g :n}}
   :n}}}
-> YFAC

{CHURCH {YFAC FIVE}}
-> 120
```

Throwing away the name, let's define and immediately call the lambda on the value **5**:

```
{CHURCH
 {{lambda {:n} {{lambda {:g :n} {:g :g :n}}
  {lambda {:g :n}
   {IFTHENELSE
    {lambda {:n} {ISZERO :n}}
    {lambda {:g :n} ONE}
    {lambda {:g :n} {MUL :n {:g :g {PRED :n}}}}
    :g :n}} :n}} FIVE}}
-> 120
```

Finally, let's replace all constants by their lambda based values to get a pure λ-calculus expression made of `words`, `abstractions` and `applications`:

```
{CHURCH {{lambda {:n} {{lambda {:g :n} {:g :g
:n}} {lambda {:g :n} {{lambda {:b :t :f :g :n}
{{{:b :n} {{lambda {:x :y :z} {:z :x :y}} :t
:f}} :g :n}} {lambda {:n} {{lambda {:n} {:n
{lambda {:x} {lambda {:z} {:z {lambda {:x :y}
:y}}}} {lambda {:z} {:z {lambda {:x :y} :x}}}}}
:n}} {lambda {:g :n} {{lambda {:n :f :x} {:f
{{:n :f} :x}}} {lambda {:f :x} :x}}} {lambda {:g
:n} {{lambda {:n :m :f} {:m {:n :f}}} :n {:g :g
{{lambda {:n} {{lambda {:z} {:z {lambda {:x :y}
:x}}} {{:n {lambda {:p} {{lambda {:x :y :z} {:z
:x :y}} {{lambda {:z} {:z {lambda {:x :y} :y}}}
:p} {{lambda {:n :f :x} {:f {{:n :f} :x}}}
{{lambda {:z} {:z {lambda {:x :y} :y}}} :p}}}}}
{{lambda {:x :y :z} {:z :x :y}} {lambda {:f :x}
:x} {lambda {:f :x} :x}}}}} :n}}}} :g :n}} :n}}
{lambda {:f :x} {:f {:f {:f {:f {:f :x}}}}}}}}
-> 120
```

Concluding this section, using nothing but words and text replacement processes, forgetting limitations of Javascript numbers and so theoretically regardless of its size and with a total precision, we can compute the factorial of any natural number. But if computing **5!** is *relatively* fast, computing **50!** would still be too long! It's time to remember that we can use the power of modern browsers to make things easier and much more faster!

3. {LAMBDA TALK}

In this section Church numbers and their related operators built as user defined functions are *forgotten* and replaced by primitive functions built on the browser's foundations. We use Javascript's numbers, Math operators and functions, HTML tags, CSS rules, SVG and more. We add *aggregate datas* like `pairs, lists, arrays` and some others specific to the wiki context. We add new special forms, `[if, let, quote, macro]`. Note that there is no `set!` special form, {lambda talk} is purely functional. This is the current dictionary:

> ***DICTionary:*** *(250) [debug, browser_cache, lib, eval, apply, <, >, <=, >=, =, not, or, and, +, -, *, /, %, abs, acos, asin, atan, ceil, cos, exp, floor, pow, log, random, round, sin, sqrt, tan, min, max, PI, E, date, serie, map, reduce, equal?, empty?, chars, charAt, substring, length, first, rest, last, nth, replace, cons, cons?, car, cdr, cons.disp, list.new, list, list.disp, list.null?, list.length, list.reverse, list.first, list.butfirst, list.last, list.butlast, array.new, array, array.disp, array.array?, array.null?, array.length, array.item, array.first, array.last, array.rest, array.slice, array.concat, array.set!, array.push!, array.pop!, array.unshift!, array.shift!, array.reverse!, array.sort!, @, div, span, a, ul, ol, li, dl, dt, dd, table, tr, td, h1, h2, h3, h4, h5, h6, p, b, i, u, center, hr, blockquote, sup, sub, del, code, img, pre, textarea, canvas, audio, video, source, select, option, object, svg, line, rect, circle, ellipse, polygon, polyline, path, text, g, mpath, use, textPath, pattern, image, clipPath, defs, animate, set, animateMotion, animateTransform, br, input, iframe, mailto, back, hide, long_mult, turtle, drag, note, note_start, note_end, show, lightbox, minibox, editable, forum, lisp, BN.DEC, BN.new, BN.+, BN.-, BN.*, BN./, BN.%, BN.pow, BN.compare, BN.negate, BN.abs, BN.intPart, BN.valueOf, BN.round, BN.fac, sheet, sheet.new, sheet.input, sheet.output, SMART_ADD, HI, GOOD_DAY, CONS, CAR, CDR, BAR, ZERO, ONE, TWO, THREE, FOUR, FIVE, CHURCH, SUCC, ADD, MUL, POWER, PRED.PAIR, PRED, ITER.PAIR, ITER, TRUE, FALSE, IF, ISZERO, FAC, Y, IFTHENELSE, ALMOST_FAC, YFAC, FACT, TFAC.rec, TFAC, BI.bigint2pol.rec, BI.bigint2pol, BI.pol2bigint.rec, BI.pol2bigint, BI.simplify.rec, BI.simplify, BI.pk, BI.p+, BI.p*, BI.tfac.r, BI.tfac, castel.interpol, castel.sub, castel.point, castel.build, svg.dot, p0, p1, p2, p3, red_curve, green_curve, QUOTIENT, SIGMA, PAREN, mul, TITLE, WRAP, ref, back_ref, space, COLUMNS]*

where can be seen the user defined functions starting at `SMART_ADD`, the first constant created for the current document.

In order to illustrate some of these new capabilities we will write effective recursive factorials, compute big numbers, play with tabular data in a spreadsheet, explore intensive computing with javascripts, build regular expressions based macros.

3.1. recursion

In the previous section we have seen how tricky it was to write a recursive algorithm. We had to build *manually* a lazy behaviour. Using the third `{if bool then one else two}` special form and its built-in *lazy evaluation* the way is opened to efficient recursive algorithms. It's now possible to write the factorial function following its mathematical definition:

```
{def FACT
 {lambda {:n}
  {if {< :n 0}
   then {b n must be positive!}
   else {if {= :n 0} then 1
   else {* :n {FACT {- :n 1}}}}}}}
-> FACT

{FACT -1} -> n must be positive!
{FACT 0}  -> 1
{FACT 5}  -> 120
{FACT 50} -> 3.0414093201713376e+64
```

Let's write the tail-recursive version:

```
{def TFAC
 {def TFAC.rec
  {lambda {:a :n}
   {if {< :n 0}
    then {b n must be positive!}
    else {if {= :n 0} then :a
    else {TFAC.rec {* :a :n} {- :n 1}}}}}}
 {lambda {:n} {TFAC.rec 1 :n}}}
-> TFAC

{TFAC 5}  -> 120
{TFAC 50} -> 3.0414093201713376e+64
```

The recursive part is called by a *helper function* introducing the accumulator "**:a**". {lambda talk} doesn't know lexical scoping - the **TFAC.rec** inner function is global - and this leads to some pollution of the dictionary. The *Y-combinator* mentionned above, *making recursive an almost-recursive function*, will help us to discard this helper function. The *Y-combinator* and the almost-recursive function can be defined and used like this:

```
{def Y {lambda {:f :a :n} {:f :f :a :n}}}
-> Y

{def ALMOST_FAC
 {lambda {:f :a :n}
  {if {< :n 0}
   then {b n must be positive!}
   else {if {= :n 0} then :a
   else {:f :f {* :a :n} {- :n 1}}}}}}
-> ALMOST_FAC

{Y ALMOST_FAC 1 5}
-> 120
```

We can do better. Instead of applying the Y combinator to the almost recursive function we can define a function composing both:

```
{def YFAC {lambda {:n}
 {{lambda {:f :a :n}
  {:f :f :a :n}}
   {lambda {:f :a :n}
    {if {< :n 0}
     then {b n must be positive!}
     else {if {= :n 0} then :a
     else {:f :f {* :a :n} {- :n 1}}}}}} 1 :n}}}
-> YFAC

{YFAC 5}
-> 120
{YFAC 50}
-> 3.0414093201713376e+64
```

We can `map` this first-class function to a sequence of numbers:

```
{map YFAC {serie 0 20}}
-> 1 1 2 6 24 120 720 5040 40320 362880 3628800
39916800 479001600 6227020800 87178291200
1307674368000 20922789888000 355687428096000
6402373705728000 121645100408832000
2432902008176640000
```

It's much fast but there is a last point to fix: `{FAC 50}`, `{TFAC 50}` and `{YFAC 50}` return a rounded value **3.0414093201713376e+64** which is obvioulsly not the exact value. We must go a little further and build some tools capable of processing big numbers.

3.2. big numbers

The way the Javascript Math object is implemented puts the limit of natural numbers to 2^{54}. Beyond this limit last digits are rounded to zeros, for instance, as we will demonstrate later, the four last digits of 2^{64} **= {pow 2 64} = 18446744073709552000** should be **1616** and are rounded to **2000**. And beyond 2^{69} natural numbers are replaced by float numbers with a maximum of 15 valid digits. In order to overcome this limitation we come back to the definition of a natural number: *A natural number* $a_0a_1...a_n$ *is the value of a polynomial* $\Sigma_{i=0}^{n}a_ix^i$ *for some value of x, called the base*. For instance $12345 = 1*10^4+2*10^3+3*10^2+4*10^1+5*10^0$. We build a set of user defined functions defining `addition, multiplication` of polynomials and some helper functions:

```
{def BI.bigint2pol
 {def BI.bigint2pol.rec
  {lambda {:n :p :i}
   {if {< :i 0}
    then :p
    else {BI.bigint2pol.rec
          :n
          {cons {charAt :i :n} :p} {- :i 1}}}}}
 {lambda {:n}
  {BI.bigint2pol.rec :n nil {- {chars :n} 1}}}}
-> BI.bigint2pol

{def BI.pol2bigint
 {def BI.pol2bigint.rec
  {lambda {:p :n}
   {if {equal? :p nil}
    then :n
    else {BI.pol2bigint.rec
           {cdr :p} {car :p}:n}}}}
 {lambda {:p}
  {let { {:q {list.reverse :p}} }
   {BI.pol2bigint.rec {cdr :q} {car :q}}}}}
-> BI.pol2bigint

{def BI.simplify
 {def BI.simplify.rec
  {lambda {:p :q :r}
   {if {and {equal? :p nil} {= :r 0}}
    then :q
    else {if {equal? :p nil} then {cons :r :q}
    else {BI.simplify.rec
          {cdr :p}
```

```
          {cons {+ {% {car :p} 10} :r} :q}
          {floor {/ {car :p} 10}} }} }}}
 {lambda {:p}
  {BI.simplify.rec {list.reverse :p} nil 0}}}
-> BI.simplify

{def BI.pk
 {lambda {:k :p}
  {if {equal? :p nil}
   then nil
   else {cons {* :k {car :p}}
              {BI.pk :k {cdr :p}}} }}}
-> BI.pk

{def BI.p+
 {lambda {:p1 :p2}
  {if {and {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {equal? :p1 nil} then :p2
   else {if {equal? :p2 nil} then :p1
   else {cons {+ {car :p1} {car :p2}}
              {BI.p+ {cdr :p1} {cdr :p2} }}}}}}}
-> BI.p+

{def BI.p*
 {lambda {:p1 :p2}
  {if {or {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {not {cons? :p1}}
   then {BI.pk :p1 :p2}
   else {BI.p+ {BI.pk {car :p1} :p2}
               {cons 0 {BI.p* {cdr :p1}
:p2}}}}}}}
-> BI.p*
```

Using this set of functions we can now compute the factorial of natural numbers of any size with an exact precision:

```
{def BI.tfac
 {def BI.tfac.r {lambda {:n :p}
   {if {< :n 1}
    then :p
    else {BI.tfac.r {- :n 1}
          {BI.simplify
           {BI.p* {BI.bigint2pol :n} :p}}}}}}
 {lambda {:n}
  {BI.pol2bigint
   {BI.simplify
    {BI.tfac.r :n {BI.bigint2pol 1}}}}}}
-> BI.tfac

{BI.tfac 50}
-> 30414093201713378043612608166064 7
   68844377641568960512000000000000
```

We finally reached the goal: with the subset of special forms `[lambda, def, if]`, the `cons` and `lists` structures, and *without any external library* we have *effectively* computed the exact value of **50!** But we need to go a little further.

3.3. when {lambda talk} calls Javascript

Until now user defined functions were exclusively created in the {lambda talk} syntax, with a large speed penalty when comes intensive computation. We can add any Javascript code using the {script ... Javascript code ...} special form. And when the set of user defined functions written in {lambda talk} or Javascript syntaxes increazes in size, it's time to externalize code in some other wiki page used as a **library** and called via a (require library_name). This helps {lambda talk} to stay minimal, coherent, orthogonal, and any user to create, add and maintain his own specific **library**. In the following we illustrate some of these capabilities.

3.3.1. the lib_BN library

Jonas Raoni Soares Silva[9] has written a small (150 lines) and smart Javascript library, `BigNumber`, ready to be called via {lambda talk} wrapping functions, everything being stored in another wiki page, `lib_BN`. We just write the factorial function using the multiplicate operator `BN.*` redefined for big numbers:

```
{def BN.fac
 {lambda {:n}
  {if {= :n 0}
   then 1
   else {BN.* :n {BN.fac {- :n 1}}}}}}
-> BN.fac
```

and call it on the number **50**:

```
{BN.fac 50}
-> 30414093201713378043612608166064 7
   68844377641568960512000000000000
```

Obviously it's the fastest and best choice!

Note: Using this library, we can control that the value of 2^{64} given by the Javascript Math object, *{pow 2 64} = 18446744073709552000* is not its exact value *{BN.pow 2 64} = 18446744073709551616*!

3.3.2. the lib_sheet library

A spreadsheet is a good illustration of functional languages, (Simon Peyton-Jones [10]). A spreadsheet is an interactive computer application for organization, analysis and storage of data in tabular form. The basic idea is that each cell contains the input - *words and expressions* - and displays the output. Calling a set of Javascript and {lambda talk} functions stored in a wiki page, `lib_sheet`, and writing {sheet 4 5} displays the following table of 5 rows and 4 columns of editable cells:

Editing cell L5C4:

{+ {IJ -3 0} {IJ -2 0} {IJ -1 0}}

NAME	QUANT	UNIT PRICE	PRICE
Item 1	10	2.1	21
Item 2	20	3.2	64
Item 3	30	4.3	129
.	.	TOTAL PRICE	214

[local storage]

Two specific functions are added for linking cells:

- `{LC i j}` returns the value of the cell L_iC_j as an *absolute reference*,
- `{IJ i j}` returns the value of the cell $L_{[i]}C_{[j]}$ as a *relative reference*. For instance writing `{IJ -1 -1}` in `L2C2` will return the value of `L1C1`.

Datas are stored in the browser's *localStorage*. The **[local storage]** button opens a window where the spreadsheet can be alternatively edited in a JSON format:

```
["{b NAME}","{b QUANT}","{b UNIT PRICE}","{b
PRICE}","Item 1","10","2.1","{* {IJ 0 -2} {IJ 0
-1}}","Item 2","20","3.2","{* {IJ 0 -2} {IJ 0
-1}}","Item 3","30","4.3","{* {IJ 0 -2} {IJ 0
-1}}","","","{b TOTAL PRICE}","{+ {IJ -3 0} {IJ
-2 0} {IJ -1 0}}","4"]
```

3.3.3. graphics

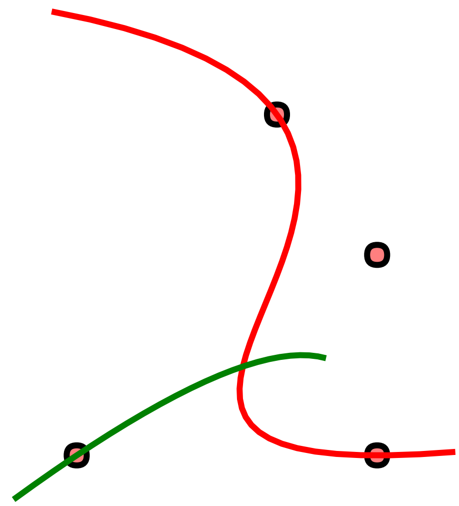
The **de Casteljau** recursive algorithm[11] allows drawing Bezier curves of any degree, i.e controlled by any number of points. Defining points as pairs and control polylines as lists, we build a small set of {lambda talk} user defined functions feeding the points attributes of SVG polylines:

```
{def castel.interpol {lambda {:p0 :p1 :t}
 {cons {+ {* {car :p0} {- 1 :t}}
          {* {car :p1} :t}}
       {+ {* {cdr :p0} {- 1 :t}}
          {* {cdr :p1} :t}}
}}}
-> castel.interpol

{def castel.sub {lambda {:l :t}
 {if {equal? {cdr :l} nil}
  then nil
  else {cons
   {castel.interpol {car :l} {car {cdr :l}} :t}
   {castel.sub {cdr :l} :t}}}}}
-> castel.sub

{def castel.point {lambda {:l :t}
 {if {equal? {cdr :l} nil}
  then {car {car :l}} {cdr {car :l}}
  else {castel.point {castel.sub :l :t} :t}}}}
-> castel.point

{def castel.build {lambda {:l :a :b :d}
 {map {castel.point :l} {serie :a :b :d}}}}
-> castel.build

{def svg.dot {lambda {:p}
 {circle {@ cx="{car :p}" cy="{cdr :p}" r="5"
            stroke="black" stroke-width="3"
            fill="rgba(255,0,0,0.5)"}}}}
-> svg.dot
```

For instance the following code:

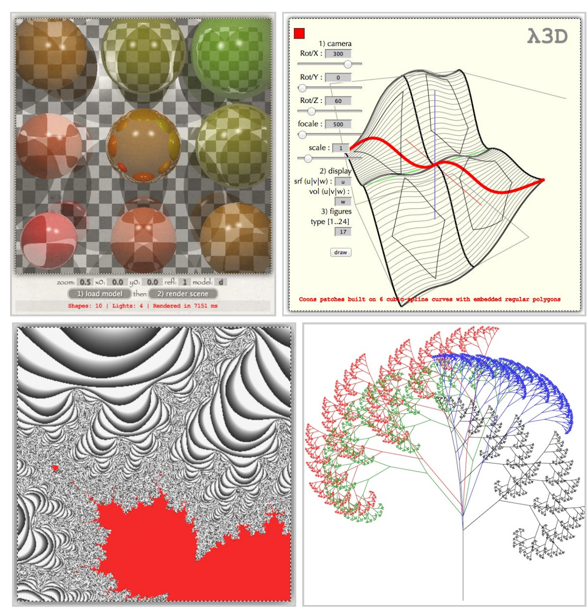
```
{def p0 {cons 150 80}}  -> p0
{def p1 {cons 200 150}} -> p1
{def p2 {cons 50 250}}  -> p2
{def p3 {cons 200 250}} -> p3
{def red_curve {castel.build
  {list {p0} {p1} {p2} {p3}}
   -0.3 1.1 {pow 2 -5}}}
-> red_curve
{def green_curve {castel.build
  {list {p2} {p1} {p3}}
   -0.1 0.6 {pow 2 -5}}}
-> green_curve

{svg.dot {p0}}
{svg.dot {p1}}
{svg.dot {p2}}
{svg.dot {p3}}

{polyline {@ points="{red_curve}"
  stroke="red" fill="transparent"
  stroke-width="3"}}
{polyline {@ points="{green_curve}"
  stroke="green" fill="transparent"
  stroke-width="3"}}
```

draws a cute **λ** in an SVG frame:



3.3.4. intensive computing

For intensive computing, it's obvioulsly more efficient to call the underlying language, Javascript. These are screenshots of {lambda tank}'s pages dedicated to **ray-tracing,**[12] **curved shapes modeling**[13]**, fractals**[14]**, turtle graphics drawing**[15].



3.3.5. mathML

$$i\hbar \frac{\partial \psi}{\partial t}(x,t) = \left(mc^2\alpha_0 - i\hbar c \sum_{j=1}^{3} \alpha_j \frac{\partial}{\partial x_j} \right) \psi(x,t)$$

The Dirac equation in the form originally proposed by Dirac {lambda talk} forgets the **MathML** markup set which is not implemented in Google Chrome[16]. A set of functions, *exclusively built on standard HTML and CSS rules*, can be defined to render Math Symbols. For instance the above **Dirac equation** is not a picture but the result of the code below:

```
i{del h}{QUOTIENT 30 ∂ψ ∂t}(x,t) = {PAREN 3 (}
mc{sup 2}α{sub 0} - i{del h}c {SIGMA 30 j=1 3}
α{sub j}
{QUOTIENT 30 ∂ ∂x{sub j}} {PAREN 3 )} ψ(x,t)
```

calling three user defined {lambda talk} functions:

```
{def QUOTIENT
 {lambda {:s :num :denom}
  {table
   {@ style="width::spx;
             display:inline-block;
             vertical-align:middle;
             text-align:center;"}
 {tr {td {@ style="border:0 solid;
                   border-bottom:1px
solid;"}:num}}
 {tr {td {@ style="border:0 solid;"}:denom}} }}}
-> QUOTIENT

{def SIGMA
 {lambda {:s :one :two}
  {table
   {@ style="width::spx;
             display:inline-block;
             vertical-align:middle;
             text-align:center;"}
 {tr {td {@ style="border:0 solid;"}:two}}
 {tr {td {@ style="border:0 solid;
                   font-size:2em;
                   line-height:0.7em;"}Σ}}
 {tr {td {@ style="border:0 solid;"}:one}} }}}
-> SIGMA

{def PAREN
 {lambda {:s :p}
  {span {@ style="font:normal :sem arial;
                  vertical-align:-0.15em;"}:p}}}
-> PAREN
```

3.4. what about macros?

A language without macros is not a *true* language, isnt'it? {lambda talk} macros bring (*a little bit of*) the power of **regular expressions** directly in the language.

3.4.1. make it variadic

{lambda talk} comes with some variadic primitives, for instance `[+, -, *, /, list, ...]`. But at first sight, user functions can't be defined variadic, for instance:

```
{def mul {lambda {:x :y} {* :x :y}}} -> mul
{* 1 2 3 4 5}    -> 120 // * is variadic
{mul 1 2 3 4 5} -> 2   // 3, 4, 5 are ignored
```

In order to make `mul` variadic we glue values in a list and define an helper function, `variadic`:

```
{def variadic
 {lambda {:f :args}
  {if   {equal? {cdr :args} nil}
   then {car :args}
   else {:f {car :args}
            {variadic :f {cdr :args}}}}}}
-> variadic

{variadic mul {list 1 2 3 4 5}}
-> 120
```

But it's ugly and doesn't follow a standard call. We can do better using a macro:

```
1) defining:
{macro {mul* (.*?)}
    to {variadic mul {list €1}}}

2) using:
{mul* 1 2 3 4 5}
-> 120
```

Now `mul*` is a variadic function which can be used as any other primitive or user function, except that it is not, as in most Lisps, a first class function.

3.4.2. titles, paragraphs & links

As a last example, {lambda talk} comes with a predefined small set of macros allowing writing without curly braces titles, paragraphs, list items, links:

```
_h1 TITLE ¬
  stands for: {h1 TITLE}

_p Some paragraph ... ¬
  stands for: {p Some paragraph ...}

[[PIXAR|http://www.pixar.com/]]
  stands for: {a {@
href="http://www.pixar.com/"}PIXAR}

[[sandbox]]
  stands for: {a {@ href="?
view=sandbox"}sandbox}
```

These simplified alternatives, avoiding curly braces as much as possible, are fully used in the current document.

CONCLUSION

{lambda talk} takes benefit from the extraordinary power of modern web browsers, simply adding a coherent and unique syntax, *without re-inventing the wheel*, just using existing tools, HTML/CSS, the DOM and Javascript. *Standing on the shoulders of such giants*, {lambda talk} can be built as a minimal regexp based implementation of the λ-calculus, where the repeated substitutions inside the code string overcomes limitations of regular language, where the lack of closure is balanced by the built-in partial application functionality, where a dictionary initially empty can be extended "inline" via user defined libraries. More can be seen in the following APPENDICE.

The {lambda way} project is a thin overlay - about 100kb - built upon any modern browser, proposing a small interactive development environment, {lambda tank}, and a coherent language, {lambda talk}, without any external dependencies and thereby easy to download and install on a web account provider running PHP. From any web browser on any system, complex web pages can be created, enriched, structured and (algorithms) tested in real time on the web. The current document has been created in this wiki page, `http://lambdaway.free.fr/workshop/?view=oxford` then directly printed from the browser as a PDF document.

.

Alain Marty, 2017/07/28

APPENDICE

In this section we present the minimal set of JavaScript functions necessary and sufficient to implement `abstractions`, `applications`, `definitions` and the `ifthenelse` control structure.

1. evaluation

Working on the client side the {lambda talk} evaluator is a Javascript **IIFE** (Immediately Invoked Function Expression), `LAMBDATALK`, returning the public function `eval()`. This function is called at every keyboard entry and replaces the string code by its evaluation, without building any Abstract Syntaxic Tree.

```
var LAMBDATALK = (function() {
var eval = function(str) {
  str = pre_processing(str);
  str = abstract_lambdas(str); // abstraction
  str = abstract_defs(str);    // abstraction
  // some other special forms
  str = eval_forms(str);       // application
  str = post_processing(str);
  return str;
};
return {eval:eval}
})();
```

2. application

In a single loop, using a single regular expression[17], simple forms `{first rest}` are recursively evaluated from the leaves to the root and replaced by words. The evaluator stops when simple forms are reduced to a sequence of words, actually a valid HTML code sent to the browser's engine for the final evaluation and display. Using a *regular expression based window*, the evaluator literally loops over the code string, skips the words and progressively replaces *in situ* forms by words. *The repeated substitutions inside the code string overcomes limitations of regular language*. A kind of Turing machine[18] ...

```
var eval_forms = function( str ) {
  while (str !=
(str=str.replace(leaf,eval_leaf)))
    ; // does nothing!
  return str
};
var leaf = /\{([^\s{}]*)(?:[\s]*)([^{}]*)\}/g;
var eval_leaf = function(_,f,r) {
  return (DICT.hasOwnProperty(f)) ?
          DICT[f].apply(null,[r]) : '('+f+'
'+r+')'
};
var DICT = {}; // initially empty
```

3. abstraction

Special forms `{lambda {arg*} body}` are matched and evaluated **before** simple forms and replaced by a reference to an anonymous function added to the dictionary. The following code demonstrates that:

- **1) lambdas** are **first class** functions,
- **2) lambdas** accept **partial function application**, when called with a number of values lesser than their arity, they memorize the given values and return a lambda waiting for the rest,
- **3) lambdas don't create closures**, inner lambdas have no access to outer lambdas' arguments, there is no lexical scoping, no environment, no free variables. Like mathematical functions lambdas are pure black boxes.

```
var abstract_lambdas = function(str) {
  while ( str !== ( str =
    form_replace(str,'{lambda',
abstract_lambda)));
  return str
};

var abstract_lambda = function(s){
 s = abstract_lambdas( s );  // nested lambdas
 var index = s.indexOf('}'),
     args = supertrim(s.substring(1,
index)).split(' '),
     body = s.substring(index+2).trim(),
     name = '_LAMB_' + LAMB_num++,
     reg_args = [];
 for (var i=0; i < args.length; i++)
   reg_args[i] = RegExp( args[i], 'g');
 body = abstract_ifs( body ); // {ifthenelse}
 DICT[name] = function() {
  var vals =
      supertrim(arguments[0]).split(' ');
  return function(bod) {
   bod = ifthenelse( bod, reg_args, vals );
   if (vals.length < args.length) {
    for (var i=0; i < vals.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
    var _args=args.slice(vals.length).join(' ');
    bod = '{' + _args + '} ' + bod;
    bod = abstract_lambda(bod);// return a
lambda
   } else {                    // return a form
    for (var i=0; i <  args.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
   }
   return bod;
  }(body);
 };
 return name;
};

var form_replace = function(str,sym,func,flag){
  sym += ' ';
  var s = catch_form( sym, str );
  return (s==='none')?
     str:str.replace(sym+s+'}',func(s,flag))
};

var catch_form = function( symbol, str ) {
  var start = str.indexOf( symbol );
  if (start == -1) return 'none';
  var d0, d1, d2;
  if (symbol === "'{")      { d0=1; d1=1; d2=1;}
  else if (symbol === "{") { d0=0; d1=0; d2=1;}
  else          { d0=0; d1=symbol.length; d2=0;}
  var nb = 1, index = start+d0;
  while(nb  > 0) { index++;
         if ( str.charAt(index) == '{' ) nb++;
    else if ( str.charAt(index) == '}' ) nb--;
```

```
  }
  return str.substring( start+d1, index+d2 )
};
```

4. definition

Special forms `{def name expression}` are matched and evaluated **before** simple forms and replaced by `name` as a reference to `expression` added to the dictionary.

```
var abstract_defs = function(str, flag) {
  while ( str !== ( str =
    form_replace( str, '{def', abstract_def,
flag ))) ;
  return str
};
var abstract_def = function (s, flag) {
  flag = (flag === undefined)? true : false;
  s = abstract_defs( s, false );
  var index = s.search(/\s/), // match spaces
      name  = s.substring(0, index).trim(),
      body  = s.substring(index).trim();
  if (body.substring(0,6) === '_LAMB_') {
    DICT[name] = DICT[body];
    delete DICT[body];
  } else {
     body = eval_forms(body);
     DICT[name] = function() { return body };
  }
  return (flag)? name : '';
};
```

5. ifthenelse

Special forms `{if bool then one else two}` are matched in lambda's bodies and replaced by the reference to an array `[bool, one, two]`. When the lambda is called with some values, `one` or `two` is returned according to the **bool** value.

```
var abstract_ifs = function(str) {
  while ( str !== ( str =
    form_replace( str, '{if', abstract_if ))) ;
  return str
};
var abstract_if = function(s){
  s = eval_ifs( s );
  var name = '_COND_' + COND_num++;
  var index1 = s.indexOf( 'then' ),
      index2 = s.indexOf( 'else' ),
      bool = s.substring(0,index1).trim(),
      one = s.substring(index1+5,index2).trim(),
      two = s.substring(index2+5).trim();
  COND[name] = [bool,one,two];
  return name;
};
var eval_ifs = function(bod, reg_args, vals) {
   var m = bod.match( /_COND_\d+/ );
   if (m === null) {
     return bod
   } else {
     var name = m[0];
     var cond = COND[name];
     if (cond === undefined) return bod;
     var bool=cond[0], one=cond[1], two=cond[2];
     if (reg_args !== undefined) {
      for (var i=0; i < vals.length; i++) {
       bool = bool.replace(reg_args[i],
vals[i]);
       one = one.replace(reg_args[i], vals[i]);
       two = two.replace(reg_args[i], vals[i]);
      }
     }
     var boolval = (eval_forms(bool)==='true')?
                   one : two;
     bod = bod.replace( name, boolval );
     return eval_ifs( bod, reg_args, vals )
   }
};
```

REFERENCES

[1] The Wiki way: http://dl.acm.org/citation.cfm?id=375211
[2] Ward_Cunningham: http://ward.asia.wiki.org/view/testing-microtalk
[3] A Tutorial Introduction to the Lambda Calculus (Raul Rojas): http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
[4] Lisp: http://www.cs.utexas.edu/~cannata/cs345/Class%20Notes/06%20Lisp.pdf
[5] Scheme: https://mitpress.mit.edu/sicp/full-text/book/book.html
[6] Collected Lambda Calculus Functions: http://jwodder.freeshell.org/lambda.html
[7] Stephen_Cole_Kleene: https://en.wikipedia.org/wiki/Stephen_Cole_Kleene
[8] L. Peter Deutsch https://fr.wikipedia.org/wiki/L._Peter_Deutsch https://sites.google.com/a/gertrudandcope.com/info/Publications/Patterns/C--Report/SpaceIII
[9] Jonas Raoni Soares Silva http://jsfromhell.com/classes/bignumber
[10] Simon_Peyton_Jones: https://www.microsoft.com/en-us/research/people/simonpj/#
[11] De_Casteljau's_algorithm: http://www.malinc.se/m/DeCasteljauAndBezier.php
[12] raytracing: http://lambdaway.free.fr/workshop/?view=raytracing
[13] pForms: http://lambdaway.free.fr/workshop/?view=pforms
[14] fractal: http://lambdaway.free.fr/workshop/?view=mandel
[15] turtle: http://lambdaway.free.fr/workshop/?view=turtle_tree
[16] google-subtracts-mathml-from-chrome: https://www.cnet.com/news/google-subtracts-mathml-from-chrome-and-anger-multiplies/
[17] Regular Expressions: http://blog.stevenlevithan.com/archives/reverse-recursive-pattern
[18] Turing machines implemented in JavaScript http://www.turing.org.uk/book/update/tmjavar.html