# Yet Another Wiki!

Alain Marty Engineer Architect
66180, Villeneuve de la Raho, France
marty.alain@free.fr

## ABSTRACT

« **An environment where the markup, styling and scripting is all S-expression based would be nice.** » would have said the father of LISP, **John McCarthy [1]**. It's the goal which was given to the **lambdaway project**: 1) build a small wiki environment, (**alphawiki**), and 2) define a small syntax, (**lambdatalk**), which allows markup, styling and scripting based on S-expressions.

## Keywords

Lisp, Javascript, Regular Expressions, CMS, wiki.

## 1. INTRODUCTION

### 1.1 The context

Web browsers can parse data (HTML code, CSS rules, scripts, ...) stored on the server side and display rich multimedia dynamic pages on the client side. Some HTML functions, (texarea, input, form, ...) associated with script languages (PHP, Javascript, Regular Expressions, ...) allow interactions with these data leading to web applications like blogs, **wikis** and CMS. Hundreds of engines have been built, managing files on the server side and interfaces on the client side, such as Wordpress, Wikipedia, Joomla,.... Syntaxes, like the *de facto* standard **Markdown syntax**, have been proposed to simplify and unify the markup and the styling but give no help on the scripting side. Some recent works have been done in this direction, for instance:

- **Skribe [5]** a text-processor based on the SCHEME programming language dedicated to writing web pages,
- **HOP [6]** a Lisp-like progamming language for the Web 2.0, based on SCHEME,
- **BRL [7]** based on SCHEME and designed for server-side WWW-based applications.

All of these projects are great and powerful. With the plain benefit of existing SCHEME implementations they make a strong and Lisp-like junction between the mark-up (HTML/CSS) and programming (JS, PHP,...) syntaxes. But these tools are devoted to **developers**, not to **users** or **web-designers**. It's the the **lambdaway project**'s goal to give all of them a common environment.

### 1.2 The lambdaway project

Why such a project? What is the current state? Who is concerned?

#### *1.2.1 why?*

1) When **Ward Cunningham [2]** invented the concept of **wiki** in 1995, a kind of online text-editor, he had in mind the powerful functionalities of an amazing software created in 1987 for Apple[Inc] by Bill Atkinson and Dan Winkler, **HyperCard** as the environment + **HyperTalk** as the language (both killed by Steve Jobs in 2001!). Nowadays, there are a lot of wiki engines which are well integrated in the browsers (the best known being Wikipedia) but the languages/syntaxes used for editing are **far from** being comparable to the **HyperTalk** powerful and user friendly language.

2) When **Brendan Eich [3]** created in 1995 the **Javascript** language for the Netscape browser, he had in his mind the powerful functionalities of the **LISP** language created in 1958 by **John Mc Carthy** at MIT. But this language, which can be considered as a LISP in C clothes, is working at the low level of the browser which is **far from** being comparable to the **HyperCard** nice and user friendly environment.

The result is not actually the **online HyperCard+HyperTalk** Ward Cunningham was dreaming of!

#### *1.2.2 what?*

**alphawiki** is a small **wiki** coming with a small language, **lambdaTalk**:

1) **alphawiki** tries to fill the gap between the complex DOM and the user with a gentle interface similar to HyperCard's one : pages are cards with text containers, pictures and buttons.

2) **lambdatalk** tries to fill the gap between the complex Javascript language and the user, with a simple and unified notation coming from LISP, used for creating rich texts, structured pages and dynamic content.

3) The couple **alphawiki** and **lambdatalk** intends to be an **easy to use online HyperCard+HyperTalk**, very small and as most elegant as possible.

#### *1.2.3 who?*

**alphawiki** is intended to be a tool for **the writer, the designer and the coder**, in a collaborative work for creating and sharing on internet, complex chunks of rich, structured and dynamic data :

1) **the writer** - who may be a "newby" - is (supposed to be) an expert in his domain and he brings the information ; with a reduced set of tags, he can fill pages with minimally structured and enriched informations (titles, paragraphs, lists, images, bold, italic, ...)

2) **the designer** - who may be "smart" - strengths the information and gives it the best shape for the best communication ; with the plain set of HTML/CSS functionalities, he can compose rich and sophisticated pages,

3) **the coder** - who may be a "ninja" - extends the functionalities as needed ; on the top, he can build new tools (a Table of Content, a worksheet, a paint or draw tool, math functions, a lisp console, ...)

The three levels share the same language, **lambdaTalk**, in an easy learning curve smoothing the frontiers between the writer, the designer and the coder.

### 1.3 In this paper

The 100kb **alphawiki's engine** can be easily installed on any **ISP**. It's mainly built on **two small "cylinders"**:

- **1) PHP.php**: on the server side a 460 lines PHP code does everything about **pages data**, reading and writing, security, administration, ...
- **2) JS.js**: on the client side a 1000 lines JS code manages the **user interface** and contains the code **interpreter**.

The present contribution will forget **alphawiki** and focus on the interpreter, **lambdatalk**, following two levels:

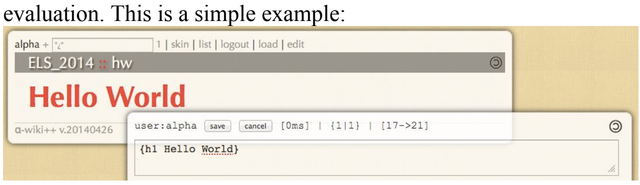
- 1) **using lambdatalk**'s primitive functions, for web-designers,
- 2) **coding lambdatalk**, with user functions, for coders,

each one viewed on both sides,

- 1) the underlying Javascript code's analyzis,
- 2) some examples of user lambdatalk code.

## 2. USING LAMBDATALK (level 1)

An alphawiki website is made of several pages sharing the same appearance. Each page can be edited (by authorized users), the result being displayed in real-time, following the **wiki-code** evaluation. This is a simple example:




The wiki-code is a string made of **plain text** containing **words** and **symbolic-expressions**.

**{first {first {first {first {first rest} } } } }**

**Symbolic-expressions** are nested expressions of the form **{first rest}** where **first** is a word belonging to a dictionary (or a symbolic-expr returning such a word), and **rest** is a string of values and symbolic-expressions.

Words are ignored by the interpreter. Symbolic-expressions are evaluated according to a **primitive function's dictionary**. For instance in the following string, the dictionary's functions "**b**" (for bold) and "**i**" (for italic) will be applyed to the red words:

```
I am a simple word,
I am a {b fat word},
You are a  {b {i fat italicized word}},
This is a product : {* 1 2 3 4 5 6}.
```

displays:

```
I am a simple word,
I am a  fat word,
You are a fat italicized word,
This is a product : 720.
```

The choice of **curly braces "{}"** instead of Lisp's standard **round parentheses "()"** comes from the wiki page context where **round parentheses** have to be used for other usages then **bracketing** symbolic-expressions.

The **JS interpreter's single task** is to **translate** the wiki-code in the HTML+CSS+JS syntaxes known by the browser. It will be up to the **browser's engine to do the hard work** to interpret and display the result. Lambdatalk and HTML sharing the **same tree structure** makes the task rather straightforward.

The next section describes the **tiny but complete JS code (level 1)**, and two of the hundred of functions belonging to the primitive's dictionary.

### 2.1. JavaScript code (level 1)

The wiki-code is caught and evaluated by the **evaluate()** function working on a **dictionary** containing primitive functions.

In the approach followed by the majority of LISP interpreters, the input string is tokenized and transformed in a tree structure, generally a nested array. The tree structure is recursively walked through and the "leaves" are evaluated. In such an approach strings must be quoted. In a wiki context where the main content is made of plain text, such a constraint must be avoided. Thus, **lambdatalk** followed a different approach based on a single loop working on a single Regular Expression inspired from **Steven Levithan [4]**.

#### *2.1.1 the evaluate() function*

```
function evaluate(str) {
  str = preprocessing( str );
  str = eval_special_forms('if', str);
  str = eval_special_forms('lambda', str);
  str = eval_special_forms('def', str, true);
  str = eval_sexprs( str );
  str = postprocessing( str );
  return str;
};
```

We ignore the **preprocessing()** and **postprocessing()** functions which don't bring here relevant informations, and the greyed lines which will come back at the level 2. The **eval_sexprs()** function's complete code can be written in this very compact shape:

```
function eval_sexprs(str) {
 while (str != (str = str.replace(
  /{([^s{}]*)(?:[s]*)([^{}]*)}/g,
   function(m, f, r) {
    return (dico.hasOwnProperty(f))?
     dico[f].apply(null,[r]):
     '('+f+' '+r+')';
   })));
 return str;
}
```

But, in order to better understand the eval_sexprs()'s mechanism, we are going to split it in three parts.

##### *2.1.1.1 the main loop*

The **eval_sexprs()** function is a **single loop** using a **single Regular Expression** to catch symbolic-expressions, and a **do_apply() function** to replace them by their value:

```
function eval_sexprs(str) {
 while (str != (str =
          str.replace(loop_rex, do_apply)));
 return str;
}
```

##### *2.1.1.2. the main regular expression*

The **loop-rex** Regular Expression is **carefully designed** to catch **{first rest}** symbolic-expressions :

```
var loop_rex =
         /{([^s{}]*)(?:[s]*)([^{}]*)}/g;
-
1) /          start of the regexp
2) {         begins with a {
3) ([^s{}]*) everything except "s{}": first
4) (?:[s]*)  zero or several spaces
5) ([^{}]*)   everything except "{}"   : rest
6) }         ends with a }
7) /          end of the regexp
8) g          go next
-
```

##### *2.1.1.3. the do_apply() function*

Given a symbolic-expression **{first rest}**, the **do_apply()** function applies **first** to **rest**, if it belongs to the dictionary called **dico**, or returns **as it is** the *invalid* symbolic-expression, if not.

```
function do_apply() {
 var first = arguments[1], rest = arguments[2];
 if (dico.hasOwnProperty(first))
  return dico[first].apply(null,[rest]);
 else
  return '('+ first +' '+ rest +')';
};
```

#### *2.1.2. dictionary*

The dictionary contains a hundred of primitive Javascript functions. The complete dictionary's content can be seen in the file JS.js.

##### *2.1.2.1 primitives*

```
HTML:
1) main tags:
@, div, span, a, [ul, ol, li], [dl, dt,
dd], [table, tr, td], pre, a, img, canvas,
iframe, embed, input, script, style
2) some others (sugar):
h1 to h6, p, b, i, u, center, br, hr, sup,
sub, del, blockquote,..

Math operators and functions:
1) math operators:
>, < , >=, <=, =, not, or, and, +, *, -,
/, %,
2) JS Math object functions:
abs, acos, asin, atan, atan2, ceil, cos,
exp, floor, log, random, round, sin, sqrt,
tan, pow, min, max, PI, E,

alphawiki's custom extensions:
lib, date, note, note_start, note_end,
show, lightbox, back, drag, listing, lisp,
lc, sheet, forum, editable, require,
include, first, rest, nth, length, serie,
map, reduce,

MathML tags: they are included but are not
recognized by Chrome
math, mrow, mfrac, mo, mi, mn, msup, msub,
msubsup, msqrt, munder, mover,..
```

### *2.1.2.2 primitive's code*

Here are given 3 illustrating examples.

```
1) the '*' math operator :
dico['*'] = function() {
  var args = arguments[0].split(' ');
  for (var r=1, i=0; i< args.length; i++)
    if (args[i] !== '')
      r *= args[i];
  return r;
};

2) HTML tags: this function builds functions
on the HTML tags set:
var htmltags = ['div','span',.., etc...];
for (var i=0; i< htmltags.length; i++) {
 dico[htmltags[i]] = function(tag) {
  return function() {
   var attr =
        arguments[0].match(/@ @[sS]*?@ @/);
   if (attr == null)
    return '< '+tag+' >'+arguments[0]+
                              '< /'+tag+' >';
   arguments[0] =
   arguments[0].replace( attr[0], '' )
               .trim();
   attr = attr[0].replace(/^@ @/, '')
                .replace(/@ @$/, '');
   return '< '+tag+' '+attr+' >'+
                arguments[0]+'< /'+tag+' >';
  }
 }(htmltags[i]);
}

3) the '@' function catches all the
HTML tags attributes and CSS rules:
dico['@'] = function () {
 return '@ @' + arguments[0] + '@ @'
};

Note : actually there is no space inside
the previous couples @ @.
```

### *2.1.3. steps & speed*

According to the main evaluation loop, symbolic-expressions are evaluated **from the leaves upto the root**. Here are given basic examples of the evaluation steps:

#### *2.1.3.1. evaluating a sequence of words:*

```
0: {center {b an {u underlined word}}}
1: {center {b an < u >underlined word< /u >}}
2: {center < b > an
           < u >underlined word< /u >< /b >}
3: < center > < b > an < u >underlined word
                   < /u >< /b >< /center >
```

This valid HTML expression will be given back to the browser's engine for evaluation.

#### *2.1.3.2. evaluating a math expression:*

```
0: {sqrt {+ {* 3 3} {* 4 4}}}
1: {sqrt {+ 9 16}}
2: {sqrt 25}
3: 5
```

This value will be given to the browser's engine to display: **5**

#### *2.1.3.3. about evaluation speed*

Alphawiki allows a rather comfortable realtime edition of a standard page. Tested on a MacBook Air:

| | Pages's content in chars | Speed |
|---|---|---|
| 1 | A page containing about **5,000** chars | **1 to 2 ms** |
| 3 | Pages between **20,000** and **50,000** chars | **5 to 10 ms** |
| 2 | A **very heavy test page** "Jules Verne, Ile mystérieuse" built on a plain text of **1,228,778** chars with a TOC of 62 chapters (about 15 360 lines = 300 pages of 50 lines) | **about 70 ms** |

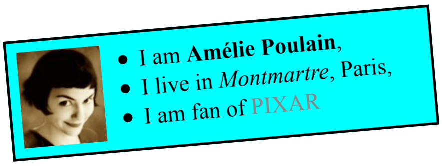
## 2.2. Lambdatalk code (level 1)

In this section we will focus on the **lambdatalk user side** and present some applications of the the built-in functionalities given by the **JS interpreter level 1**:

- to style and compose text,
- to compute mathematical expressions,
- to display images,
- to interact with dynamic elements,
- to build A4 formats, slides, posters,
- to create a forum, a spreadsheet,
- to call external JS code.

### *2.2.1. some text and image in a block*

```
{div
 {@ id="myId" style="
     position:relative;
     left:50px; top:10px;
     width:210px; height:50px;
     padding:5px; margin-bottom:-90px;
     background:cyan; border:1px solid;
     box-shadow:0 0 8px black;
     -moz-transform:rotate(-5deg);
     -webkit-transform:rotate(-5deg);"}
{img
 {@ src="data/amelie_sepia.jpg"
    height="50"
    title="Amélie Poulain"
    style="float:left; margin-right:20px;"}}
 {ul
  {li I am {b Amélie Poulain},}
  {li I live in {i Montmartre}, Paris,}
  {li I am fan of
   {a {@ href="http://www.pixar.com/"}PIXAR}}
 }
}
```




It must be noted that the function @ (pronounce "at") comes with HTML attributes and CSS rules written in their standard syntax, and **NOT** as symbolic-expressions. Using such a pure s-expression:

```
{@ {id myId} {style
     {text-align center} {border 1px solid}}}
```

instead of:

```
{@ id="myId" style="
       text-align:center; border:1px solid;"}
```

would respect more nicely the **claimed notation's coherence** but this would increase the dictionary with **innumerable CSS rules** and probably slow down the evaluation, would increase the difficulty to follow the **future evolutions** of HTML tags, attributes and CSS rules, and would be less convenient for beginners and for web-designers. This is a matter of debate and choice.

### *2.2.2. numbers & booleans*

**lambdatalk** offers the usual numeric computation capabilities that a pocket calculator would have. Following the same syntax **{first rest}** where **first** is a math function (+, -, *, /, %, sqrt, ...) and **rest** a sequence of numbers and/or valid s-expressions, any complex math expressions can be evaluated and inserted anywhere in the page. For instance writing in the editor frame:

```
1: {/ 1 2}
2: {* 1 2 3 4 5 6}
3: hypo(3,4) = {sqrt {+ {* 3 3} {* 4 4}}}
4: sin(90°) = {sin {/ {PI} 2}}
5: {/ {round {* {PI} {pow 10 4}}} {pow 10 4}}
6: {map sqrt {serie 2 4}}
7: {reduce * {serie 1 6}}
8: {< 1 2}
9: {= 1 1.000}
10: {or true false}
11: {and true false}
12: {b 1+2+3}, {+ 1 2 3} and {u {+ 1 2 3}}
```

displays:

```
1: 0.5
2: 720
3: hypo(3,4) = 5
4: sin(90°) = 1
5: 3.1416
6: 1.4142135623730951 1.7320508075688772 2
7: 720
8: true
9: true
10: true
11: false
12: 1+2+3, 6 and 6
```

**Note** the **similarity** between words-based and numbers-based symbolic expressions, allowing an easy mixture of words and numbers everywhere in a page.

### *2.2.3. input & script*

The **input** and **script** functions make it possible to call JS scripts to bring **interactivity** in the wiki pages.

#### *2.2.3.1. input*

```
{p A script interacting with the user:
{input
 {@ id="input"
    type="text"
    placeholder="Please, enter your name"
    onkeyup="
      getId('output').innerHTML=
'Hello '+getId('input').value+' !' "}}}

{h3 {@ id="output"}}
```

A script interacting with the user:

Amélie Poulain

**Hello Amélie Poulain!**

#### *2.2.3.2. script*

```
{div
 {@ id="output"
    style="font:bold 1.3em courier; color:red;"
 }time:}
{input
 {@ type="submit" value="start" onclick="start(
}}
{input
 {@ type="submit" value="stop" onclick="stop()"
}}
{script ••
function start() {
 document.chrono=window.setInterval(
  function() {
   getId('output').innerHTML='time: '
   + LAMBDATALK.eval_sexprs('{date}');
  }, 1000 );
}
function stop() {
 window.clearInterval(document.chrono);
}
••}
```

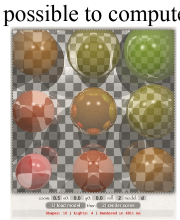
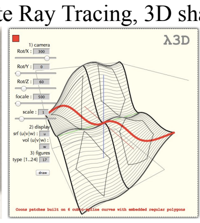
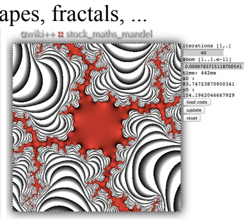
**time: 2014 08 14 12 30 30**

start stop

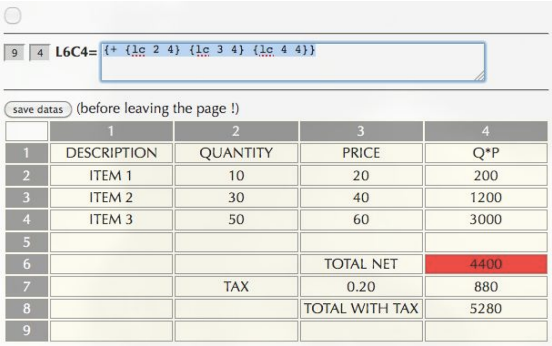
**Note:** for the sake of security, the **input and script** functions don't accept external links.

### *2.2.4. plugins*

Lambdatalk can call more complex scripts stored in the "**plugins**" folder and executed interactively in the wiki page. For instance, it's possible to compute Ray Tracing, 3D shapes, fractals, ...

**Spreadsheets** are known to be a good illustration of the functional approach. It's possible to insert a spreadsheet in a wiki page and to make some calculus in it. For instance, the symbolic-expression **{+ {lc 2 4} {lc 3 4} {lc 4 4}}** written in the cell L6C4 will display the sum of the contents of cells L2C4, L3C4 and L4C4, as it can be seen in the figure below:



9 4 L6C4= {+ {lc 2 4} {lc 3 4} {lc 4 4}}

save datas (before leaving the page !)

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | DESCRIPTION | QUANTITY | PRICE | Q*P |
| 2 | ITEM 1 | 10 | 20 | 200 |
| 3 | ITEM 2 | 30 | 40 | 1200 |
| 4 | ITEM 3 | 50 | 60 | 3000 |
| 5 | | | | |
| 6 | | | TOTAL NET | 4400 |
| 7 | | TAX | 0.20 | 880 |
| 8 | | | TOTAL WITH TAX | 5280 |
| 9 | | | | |

**alphawiki** can be considered as a stack of pages. In the same way, a spreadsheet embedded in a page can be viewed as a grid of micro-pages with all the lambdatalk's capabilities. We are going to see in the **level 2** that these capabilities can be extended grace to three powerful special forms.

# 3. CODING LAMBDATALK (level 2)

**lambdatalk** comes with the **smallest set of 3 special forms** allowing to code:

**if, lambda, def**

1) **{if boolean then TRUE_term else FALSE_term}**
   *allows **alternative** evaluation according to a given boolean values,*
2) **{lambda {:arguments} s-expression}**
   *allows **binding** in symbolic-expressions unknown terms to future values via a set of arguments; a kind of **delayed evaluation**,*
3) **{def name value}**
   *allows giving a **name** to constants, evaluable symbolic-expressions and functions.*

As we are going to see, **with these 3 special forms**, lambdatalk has **first class functions**, functions can be **recursive, called partially, nestable and used for local variables**.

## 3.1 JS INTERPRETER (level 2)

De facto, the symbolic-expressions built on the 3 special forms contain **unknown terms**. For instance in **{def myPI 3.1416}** the term **myPI** is unknown and the symbolic-expression can't be evaluated by the simplified **evaluate()** function shown previously. These special forms **must be handled in a preprocessing phase** before the evaluation loop.

### *3.1.1 evaluate() function (complete)*

```
function evaluate(str) {
 str = preprocessing( str );
 str = eval_special_forms('if', str);
 str = eval_special_forms('lambda', str);
 str = eval_special_forms('def', str, true);
 str = eval_sexprs( str );
 str = postprocessing( str );
 return str;
}
```

Remember that the greyed lines belong to the level 1.

### *3.1.2 evaluating sequence of S-expressions built with if, lambda and def*

The **eval_special_forms()** function catches symbolic-expressions built on special forms **if, lambda, def** and replaces them in the code string by their value or by symbolic-expressions evaluated when all the missing values are given.

```
function eval_special_forms(form, str, flag){
 while (true) {
  var s = catch_sexpression(form, str);
  if (s === 'none') break;
  switch (form) {
   case 'if':
    str = str.replace('{if '+s+'}',
                      eval_if(s.trim()));
   break;
   case 'lambda':
    str = str.replace('{lambda '+s+'}',
                      eval_lambda(s.trim()));
   break;
   case 'def':
    str = str.replace('{def '+s+'}',
                      eval_def(s.trim(),flag));
   break;
  }
 }
 return str;
}
```

### *3.1.3 catching an S-epxression in the wiki-code string*

The **catch_sexpression()** function catches the symbolic-expressions according to the given symbol **if, lambda, def**.

```
function catch_sexpression(symbol, str) {
 symbol = '{' + symbol + ' ';
 var start = str.indexOf( symbol );
 if (start == -1) return 'none';
 var long = symbol.length, nb=1, index=start;
 while(nb  > 0) {
  index++;
  if ( str.charAt(index) == '{' )      nb++;
  else if ( str.charAt(index) == '}' ) nb--;
 }
 return str.substring( start+long, index );
}
```

### *3.1.4. eval_lambda()*

We remember that the **evaluate()** function used a Regular Expression to replace the symbolic-expressions by their values. The **eval_lambda()** function does the same:

- The **eval_lambda()** function builds a function with a list of **arguments** and a **body** and stores it in the dictionary under a randomized name, i.e. **lambda_1234**.
- When called with some values, this function will use the arguments' names as Regular Expression patterns to replace in the body the arguments' occurences by the corresponding values.
- It will return a symbolic-expression, a value, a word or a number.

Two cases are to be considered depending on number of values:

- **if number of values < number of arguments**: it memorizes the given values and returns the name of a function waiting for the missing values,
- **else**: the symbolic-expression contains evaluable terms and will be returned to the main loop for evaluation, extra values are just ignored.

```
var eval_lambda = function (s) {
 s = eval_special_forms( 'lambda', s );
 var name = 'lambda_' +
            Math.floor(Math.random()*10000),
     args = s.substring(1, s.indexOf('}'))
            .trim().split(' '),
     body = s.substring(s.indexOf('}')+1)
            .trim();
 dico[name] = function () {
  var vals = arguments[0].split(' ');
  return function (bod) {
```

```
   if (vals.length < args.length) {
    for (var i=0; i < vals.length; i++)
     bod = bod.replace(RegExp( args[i],
                       'g'), vals[i] );
     var _args = args.slice(vals.length)
                .join(' ');
     bod = eval_special_forms('lambda',
         '{lambda {'+_args+'}'+bod+'}');
   } else {
    for (var i=0; i <  args.length; i++)
     bod = bod.replace(RegExp( args[i], 'g'),
                       vals[i] );
   }
   return bod;
  }(body);
 };
 return name;
};
```

### *3.1.5. eval_def()*

The **def** special form extends the dictionary with user functions; it **gives names** to constants, to evaluable symbolic-expressions and to lambdas. User function's names are given before the main evaluation loop and so can be called by any symbolic-expressions anywhere in the page.

```
var eval_def = function (s, flag) {
 s = eval_special_forms( 'def', s, false );
 var name = s.substring(0, s.indexOf(' ')).trim(),
     body = s.substring(s.indexOf(' ')).trim();
 dico[name] = (dico.hasOwnProperty(body))?
  dico[body] :
  function () { return body };
 delete dico[body];
 return (flag)? name : '';
};
```

### *3.1.6. eval_if()*

The **if** special form is twinned with an **_if_** function belonging to the dictionary, in a **deactivation/reactivation process**.

#### *3.1.6.1 deactivation*

The **if** special form returns a modified symbolic-expression where **if** is replaced by **_if_** and where the **then_term** **<u>AND</u>** the **else_term** are **deactivated**.

```
var eval_if = function( s ) {
 s = eval_special_forms( 'if', s );
 var index1 = s.indexOf( 'then' ),
     index2 = s.indexOf( 'else' ),
     bool = s.substring(0,index1).trim(),
     then_term = s.substring(index1+5,index2).trim(),
     else_term = s.substring(index2+5).trim();
 then_term = then_term.replace(/{/g, '&123,')
                      .replace(/}/g, '&125,');
 else_term = else_term.replace(/{/g, '&123,')
                      .replace(/}/g, '&125,');
 return '{_if_ ' + bool + ' then ' +
      then_term + ' else ' + else_term + '}';
};
```

#### *3.1.6.2 reactivation*

The **_if_** function returns a modified symbolic-expression where the **then_term** **<u>OR</u>** the **else_term** is **reactivated** according to the **bool_term** value.

```
dico['_if_'] = function () {
 var s = arguments[0],
     index1 = s.indexOf( 'then' ),
     index2 = s.indexOf( 'else' ),
     bool_term = s.substring(0,index1).trim(),
     then_term = s.substring(index1+5,index2).trim(),
     else_term = s.substring(index2+5).trim(),
     r = (bool_term === "true")?
                      then_term : else_term;
 return r.trim().replace( /&123,/g, '{' )
                .replace( /&125,/g, '}' );
};
```

**Note:** in the HTMLentities "**&123,**" and "**&125,**" the "," character is *of course* to be replaced by ";".

---

## 3.2. Coding lambdatalk

Grace to a reduced set of 3 special forms [**if, lambda, def**], lambdatalk becomes a **programmable programming language**.

- 1) **selections** can be done according to **boolean values**,
- 2) **user functions** can be built to **bind** in S-expressions **future values to arguments**,
- 3) and the **dictionary can be extended** by **user functions**.

This section presents some examples illustrating these capabilities.

### *3.2.1 constants*

#### *3.2.1.1 scalars*

```
{def myPI 3.1416}
{myPI} is the value pointed by PI
{def my2PI {* 2 {myPI}}}
{my2PI}
```

```
myPI
3.1416 is the value pointed by PI
my2PI
6.2832
```

Note that, contrary to the classics of LISP dialects, writing **myPI** displays **myPI** and NOT **3.1416**. The name of a constant must be considered as a **pointer** to a content. It's happy in a wiki context and leads to some interesting properties, for instance **arrays**.

#### *3.2.1.2 arrays*

```
{def V 0.123 0.456}
V's content = [{V}]
V's length = {length {V}}
V[0] = {nth 0 {V}}
V[1] = {nth 1 {V}}
V[2] = {nth 2 {V}}
norm(V) = {sqrt {+
             {* {nth 0 {V}} {nth 0 {V}}}
             {* {nth 1 {V}} {nth 1 {V}}}}}
```

```
V
V's content = [0.123 0.456]
V's length = 2
V[0] = 0.123
V[1] = 0.456
V[2] = undefined // V's length is 2!
norm(V) = 0.4722975756871932
```

In the previous example **V** is defined as a list of two numerical values. Grace to the primitive **nth** V <u>can be considered as an array with length = 2</u> and on which some **vector algebra** can be done, for instance computing the norm of a vector. In the same way, **polynoms, complex numbers, and some others array structures** can be defined. But **<u>lambdatalk knows nothing about arrays and other structures</u>**. The **type** of the value depends on the context and on the functions designed upon them by the coder. It's up to the coder to do that!

- more to see on arrays, vectors, complex numbers in [ARRAY]
- some reflexions about Object Oriented Programing in [OOP].

### *3.2.2 lambdas*

We have seen that lambdatalk uses the arguments' names as a **Regular Expression** pattern to replace the arguments occurences by the given values in the body's string. To avoid undesirable replacements, arguments' names **must be prefixed** by some distinctive char, i.e. a colon ":".

#### *3.2.2.1 anonymous lambdas*

```
1: defining a lambda with two arguments:
{lambda {:a :b}
  {sqrt {+ {* :a :a} {* :b :b}}}}
2: calling it with one value:
{{lambda {:a :b}
 {sqrt {+ {* :a :a} {* :b :b}}}} 3}
3: calling it with one value then with another:
{{{lambda {:a :b}
 {sqrt {+ {* :a :a} {* :b :b}}}} 3} 4}
4: calling it with two values:
{{lambda {:a :b}
 {sqrt {+ {* :a :a} {* :b :b}}} 3 4}
```

```
1: lambda_7908
2: lambda_8897
3: 5
4: 5
```

#### *3.2.2.2 giving a name to a lambda*

```
{def hypo
 {lambda {:a :b}
  {sqrt {+ {* :a :a} {* :b :b}}}
}}
hypo(3,4) is equal to {hypo 3 4} - > 5
```

#### *3.2.2.3 a local var created with an inside lambda call*

```
{def roundto
 {lambda {:x :d}
  {{lambda {:x :p}
    {/ {round {* :x :p}} :p}
   } :x {pow 10 :d}}
}}
{def print
 {lambda {:i}
  {br}roundto(PI,:i) = {roundto {PI} :i}
}}

{map print {serie 0 5}
```

```
roundto(PI,0) = 3
roundto(PI,1) = 3.1
roundto(PI,2) = 3.14
roundto(PI,3) = 3.142
roundto(PI,4) = 3.1416
roundto(PI,5) = 3.14159
roundto(PI,6) = 3.141593
roundto(PI,7) = 3.1415927
roundto(PI,8) = 3.14159265
roundto(PI,9) = 3.141592654
```

#### *3.2.2.4 the quadratic equation*

Words and numbers can easily be mixed, without any string quotation or any special printing format. This is an example giving the roots of a quadratic equation $\mathbf{ax^2 + bx + c = 0}$:

```
{def equation
 {lambda {:a :b :c}
  {{lambda {:a :b :c :d}
discriminant = :d
   {if {> :d 0} then
2 real roots :
   x1 = {/ {- {- :b} {sqrt :d}} {* 2 :a}}
   x2 = {/ {+ {- :b} {sqrt :d}} {* 2 :a}}
   else {if {= :d 0} then
1 double real root :
   x = {/ {- :b} {* 2 :a}}
   else
2 complex roots :
   x1 = [{/ {- :b} {* 2 :a}} ,
             -{/ {sqrt {- :d}} {* 2 :a}}]
   x2 = [{/ {- :b} {* 2 :a}} ,
             +{/ {sqrt {- :d}} {* 2 :a}}]
   }}
  } :a :b :c {+ {* :b :b} {* 4 :a :c}}}
}}
```

```
{equation 1 -1 1} ->
discriminant = 5
   2 real roots :
   x1 = -0.6180339887498949
   x2 = 1.618033988749895

{equation 1 -2 1} ->
discriminant = 0
   1 double real root :
   x = 1

{equation 1 1 -1} ->
discriminant = -3
   2 complex roots :
   x1 = [-0.5 , -0.8660254037844386]
   x2 = [-0.5 , +0.8660254037844386]
```

### *3.2.3 recursive*

Two approaches, the naïve and the fast/tail recursion.

#### *3.2.3.1 basic recursion*

```
{def fac
 {lambda {:n}
  {if {< :n 1}
   then 1
   else {* :n {fac {- :n 1}}}}}}

1*...*6 = {fac 6} = 720
```

#### *3.2.3.2 tail recursion*

```
{def ifac
 {lambda {:result :n}
  {if {< :n 1}
   then :result
   else {ifac {* :result :n} {- :n 1}}
}}}

1*...*21 = {ifac 1 21} = 51090942171709440000
```

#### *3.2.3.3 pascal coefficients C(n,p) = C(n-1,p-1)*n/p*

```
{def pascal
 {lambda {:n :p}
  {if {or {< :n 2} {< :p 1}}
   then 1
   else {/ {* :n {pascal {- :n 1} {- :p 1}}} :p
}}}
{def pascal_line
 {lambda {:n :p}
  {if {< :p 0}
   then .
   else {pascal :n :p} {pascal_line :n {- :p 1}
}}}

{map {lambda {:n} {br}{pascal_line :n :n}}
                                   {serie 0 12}}
```

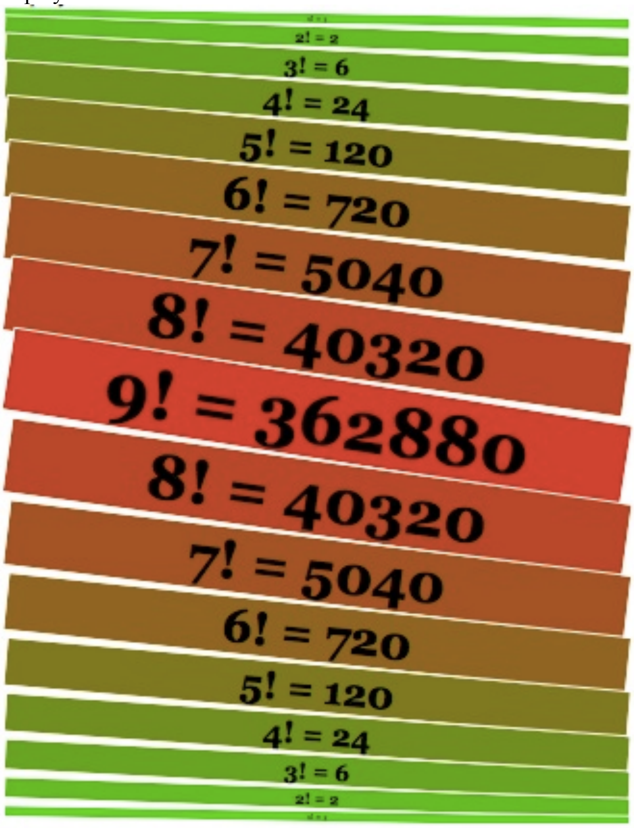
```
1 .
1 1 .
1 2 1 .
1 3 3 1 .
1 4 6 4 1 .
1 5 10 10 5 1 .
1 6 15 20 15 6 1 .
```

```
1 7 21 35 35 21 7 1 .
1 8 28 56 70 56 28 8 1 .
1 9 36 84 126 126 84 36 9 1 .
1 10 45 120 210 252 210 120 45 10 1 .
1 11 55 165 330 462 462 330 165 55 11 1 .
1 12 66 220 495 792 924 792 495 220 66 12 1 .
```

#### 3.2.3.4 some web design

```
{def fac
 {lambda {:n}
  {if {< :n 1} then 1 else {* :n {fac {- :n 1}}}
}}}
{def postits
 {lambda {:n}
  {div {@ style="
  font:bold {/ :n 3}em georgia;
  background:rgb({* :n 25},{- 250 {* :n 25}},0);
  border:1px solid white;
  text-align:center;
  -webkit-transform:rotate(:ndeg);
     -moz-transform:rotate(:ndeg);
               transform:rotate(:ndeg);
     "}:n! = {fac :n}
}}}
{map postits 0 1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1 0}
```

displays:




### 3.2.4 formulas

As long as the **mathML** tags won't be recognized by Chrome, lambdatalk can be used to display formulas.

#### 3.2.4.1 defining some lambdatalk functions

```
{def numero
 {lambda {:n} {div {@ style=
"float:right; font-size:12px;"}:n}}}

{def radicand {@ style=
 "text-decoration:overline;"}}

{def quotient {lambda {:h}
 {@ style="display:inline-block;
   text-align:center;
   font-size::hem;
   vertical-align:-0.8em;"}}}

{def quotient_line
 {lambda {:w} {div {@ style="
 border-top:1px solid;
 height:0px; width::wpx;"}}}}
```

```
numero
radicand
quotient
quotient_line
```

#### 3.2.4.2 using these functions to display formulas

```
x = {div {quotient 1.0}
{div -b ±
√{span {radicand} b{sup 2} - 4ac}}
{quotient_line 100}
{div 2a}}
{numero 1.1}

Δf(x,y,z) = {div {quotient 1.0}
{div ∂{sup 2}f(x,y,z)}
{quotient_line 60} {div ∂x{sup 2}}}
+ {div {quotient 1.0}
{div ∂{sup 2}f(x,y,z)}
{quotient_line 60} {div ∂y{sup 2}}}
+ {div {quotient 1.0}
{div ∂{sup 2}f(x,y,z)}
{quotient_line 60} {div ∂z{sup 2}}
}
{numero 1.2}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad 1.1$$

$$\Delta f(x,y,z) = \frac{\partial^2 f(x,y,z)}{\partial x^2} + \frac{\partial^2 f(x,y,z)}{\partial y^2} + \frac{\partial^2 f(x,y,z)}{\partial z^2} \qquad 1.2$$

### 3.2.5 recursion vs composition

In order to compare the recursion and the compose methods, we Compute the first derivees of the cubic function via recursion and via composition.

#### 3.2.5.1 recursion

The recurrent relation giving the pth finite difference is used to write an approximation of the derivee:

```
ΔPf(x) = ΔP-1f(x+dx) - ΔP-1f(x)
```

```
{def rD
 {lambda {:n :func :x :dx}
  {if {<  :n 1}
   then
    {:func :x}
   else
 {/ {- {rD {- :n 1} :func {+ :x :dx} :dx}
       {rD {- :n 1} :func :x :dx}} :dx}
}}}
{def cubic {lambda {:x} {* :x :x :x}}}

{round {rD 0 cubic 2 0.001} - > 8
{round {rD 1 cubic 2 0.001} -> 12
{round {rD 2 cubic 2 0.001} -> 12
{round {rD 3 cubic 2 0.001} -> 6
{round {rD 4 cubic 2 0.001} -> 0
```

#### 3.2.5.1 composition

**lambdatalk** functions can be partially called. Writing derivees of any order is straightforward.

```
{def cD
```

```
 {lambda {:f :x}
  {/ {- {:f {+ :x 0.01}}
        {:f {- :x 0.01}} } 0.02}
}}
{def cubic {lambda {:x} {* :x :x :x}}}

{round {cubic 2}}                     -> 8
{round {{cD cubic} 2}}                -> 12
{round {{cD {cD cubic}} 2}}           -> 12
{round {{cD {cD {cD cubic}}} 2}}      -> 6
{round {{cD {cD {cD {cD cubic}}}} 2}} -> 0
```

### 3.2.6 1stClassFunc

According to rosettacode.org, lambdatalk has First Class Functions:

> *A language has first-class functions if it can do each of the following without recursively invoking a compiler or interpreter or otherwise metaprogramming:*
> - *Create new functions from preexisting functions at run-time*
> - *Store functions in collections*
> - *Use functions as arguments to other functions*
> - *Use functions as return values of other functions*

Example:

```
1) add cube and cuberoot user functions,
2) store sin, cos, cube in the "array" fun
3) store asin, acos, cuberoot in "array" inv
4) define compose(f,g,x) as f(g(x))
5) display compose(fun[i],inv[i],0.5)
   for i in [0,2]
6) The result must be always 0.5,
within the limits of computational accuracy.
```

```
{def cube
 {lambda {:x} {pow :x 3}}}
{def cuberoot
 {lambda {:x} {pow :x {/ 1 3}}}}
{def compose
 {lambda {:f :g :x} {:f {:g :x}}}}
{def fun sin cos cube}
{def inv asin acos cuberoot}
{def display
 {lambda {:i}
  {br}{compose {nth :i {fun}}
               {nth :i {inv}} 0.5}
}}
{map display {serie 0 2}}
->
0.49999999999999994
0.5000000000000001
0.5000000000000001
```

### 3.2.7 let & set!

It was a choice to limit the set of special forms to 3. Sometimes we need more functionalities!

#### 3.2.7.1 let

There is **NO let special form**. This Lisp's standard *sugar* form can be easily replaced by lambdas. For instance, the area of a triangle [a,b,c] is given by this formula:

```
Let a,b,c be the sides of a triangle,
then area = sqrt[s*(s-a)*(s-b)*(s-c)]
```

It's a good thing to compute once the value s used 4 times :

```
{def triangle
 {lambda {:a :b :c}
  {{lambda {:a :b :c :s}
   {sqrt {* :s {- :s :a}
               {- :s :b}
               {- :s :c}}}
  } :a :b :c {/ {+ :a :b :c} 2}}
}}
```

```
the area of triangle [3,4 5] is
                     {triangle 3 4 5} -> 6
```

#### 3.2.7.2 set!

There is **NO set! special form**. Lambdatalk chose to follow the pure functional programming paradigm. But, in case of necessity, the **lisp** lambdatalk function can embed calls to a **tiny but true** LISP interpreter, **lambdalisp**, included in the "plugins" folder which was written following Peter Norvig Python Lisp's implementation. With lambdalisp, we can do that inside the page:

```
{lisp
(define make-account
  (lambda (balance)
    (lambda (amt)
      (begin
        (set! balance (+ balance amt))
        balance
) ) ) )
}

> {lisp (define a1 (make-account 100))}
> {lisp (a1 -20)} -> 80
> {lisp (a1 -20)} -> 60
> {lisp (a1 -20)} -> 40
> {lisp (a1 -20)} -> 20
> {lisp (a1 -20)} -> 0
> {lisp (a1 -20)} -> {u oops}
```

Note the standard parentheses **()** instead of the curly braces.

### 3.2.8 more

We don't forget that **we are in a wiki context**, where text/code is entered and evaluated in real time, and error messages are not welcome!

#### 3.2.8.1 lambdatalk accepts a number of values ≠ number of arguments

We have seen that functions can be called with any number of arguments (curry, partial application). This makes things easy, for instance :

```
{def boo {lambda {:a :b} {+ :a :b}}} -> boo
   // OK, it's a function waiting for 2 values
{boo}         -> lambda_6195
   // OK, it's a function waiting for 2 values
{boo 1}       -> lambda_3262
   // it's a function waiting for 1 value
{boo 1 2}     -> 3
   // OK, it's called with two values
{{boo 1} 2}   -> 3
   // OK, it's called in two steps
{boo 1 2 3}   -> 3
   // OK, no matter with extra values
```

#### 3.2.8.2 syntax errors are ignored

lambdatalk is **permissive** and it's very useful in a wiki context. For instance :

```
{oops yep hip} -> {oops yep hip}
```

No matter the fact that **oops** is not a known function, or **yep** or **hip** are unknown values, lambdatalk returns the symbolic-expression, **as it is**, unevaluated, just with blue-colored curly braces!

#### 3.2.8.3 some alternate simplified notations (a kind of level 0 for beginners)

Beginners don't like symbolic-expressions! Because **titles, paragraphs and ordered/unordered lists** are blocks **between two carriage returns**, they can be written without curly braces via easy alternative forms: **_h1, _p, _ul, _ol**, for instance:

- **{h1 TITLE}** can be replaced by **_ h1 TITLE CR**

- **{p some text}** can be replaced by **_ p Some text CR**
- **{ul {li unordered list item}}** can be replaced by **_ ul unordered list item CR**
- **{ol {li ordered list item}}** can be replaced by **_ ol ordered list item CR**

**Links** are not forgotten and can be written using a standard **Markdown syntax**:

- **{a {@ href="?view=Introduction"}Introduction}** can be replaced by **[ [Introduction] ]**,
- **{a {@ href="http://www.pixar.com/"}PIXAR}** can be replaced by **[ [PIXAR|http://www.pixar.com/] ]**,

#### *3.2.8.4 quoting, comments, locking*

- Lambdatalk doesn't need any **quote** special form. To display a **symbolic-expression unevaluated** as it is, write this:
  **°°{first rest}°°**
- To **hide blocks of any text** write this :
  **°°°THIS IS A COMMENT°°°**
- To **temporarily lock** the page code evaluation, for instance in a page with long time evaluation, just **unbalance** curly braces.

## 4. CONCLUSION

We have seen the both sides of the interpreter, the underlying engine and the syntax. We have highlighted two steps, one for the user, the other for the coder.

- **1)** The lambdatalk syntax is **small, simple and easy to be used** by any beginner and any web-designer.
- **2)** The underlying JS code is **small, simple and easy** to be mastered by any JS developer.
- **3)** The underlying JS code appears to be **fast enough** to be usable in the context of webdesign.
- **4)** The lambdatalk syntax appears to be **powerful enough** to follow some more complex developer's experimentations.

With α-wiki and λ-talk, **the beginner, the web-designer and the developer** benefit from a **simple text editor and a common syntax** allowing them, in a **gentle learning slope** and a **collaborative work**, to build sets of complex and dynamic pages. **alphawiki is free**, under the Copyleft Licence. This presentation has been made with **alphawiki** at **http://epsilonwiki.free.fr/alphawiki_2**.

## 5. REFERENCES


- [1] : John McCarthy, for LISP
- [2] : Ward Cunningham, for WIKI
- [3] : Brendan Eich, for Javascript
- [4] : Steven Levithan, for Regular Expressions
- [5] : Manuel Serrano, for SKRIBE, http://www-sop.inria.fr/,
- [6] : Manuel Serrano, for HOP, http://en.wikipedia.org/wiki/Hop,
- [7] : Bruce R.Lewis, for BRL, http://brl.sourceforge.net/,