Plotting and Scheming the Ubiquitous LISP.

André van Meulebrouck (vanMeule@earthlink.net)

"Try a lot of stuff and keep what works." (James Collins, author of "Built to Last")

Introduction

The purpose of this paper is to propose a game plan whereby LISP can become the ubiquitous lingua franca of computerdom; a role for which it has apt credentials.

The world wants not for better technology, but for knowledge of the better technologies that are already out there.

I will be drawing on existing ideas. Originality comes into play in determining which things should be knitted together in what ways to arrive at a winning formula.

The real world is the ultimate test bed; hence the ubiquitous tool is afforded the benefits to be learned from emerging through the best testing the world has to offer. That is why I think making LISP mainstream is such an important goal.

Ancient History

"Those who forget the past are doomed to repeat it." (George Santayana)

The current marginalized status of LISP and the vitriolic contempt held by many (especially by some in the LISP community!) towards the perceived hegemony of Microsoft reminds me of Ancient Israel under Roman subjugation.

The questions of the day involved whether to pay taxes to Rome or revolt. Controversy raged over the idealism of martyrdom versus the pragmatism of compromising to live for a better day: where is the boundary between compromise and selling out?

The best advice they got was to "render unto Caesar" and to get rid of an enemy by turning him into a friend. That must have been a tough pill to swallow! It must have sounded like pie-in-the-sky New Age folderol; and yet Josephus tells us the Jewish revolt went very badly, ending in suicide at Masada.

But that's not the end of the story: it gets stranger. The oppressor goes on to embrace the religion of the oppressed; and the language of the oppressor becomes a sacred language to the oppressed (even today, amongst Roman Catholics). How bizarre is that???

Nonetheless, if it could happen then, it can happen today.

Desideratum: a ubiquitous LISP.

I have always been puzzled as to why LISP isn't more widespread.

Like Martin Luther King, Jr.; I have a dream. My dream is that one day LISP will gain the acceptance I feel it should have had from the onset.

Hope springs eternal, but how do we know if our hope is in vain or attainable?

Certainly if someone is waiting for a bus that will never arrive, being apprised of that fact would be liberating. (I.e. "Sir, you're waiting for a bus that will never arrive: Your understanding of the schedules is incorrect.")

If asked whether I think my dream could ever come true, I would respond a la Pascal: It is easier to imagine that something that *has been* will *be again* than to imagine that something which has never been, will be.

LISP has had some successes! Bits and pieces of LISP thrive in the genes of many extant mainstream technologies; but how can we convince people to stop stealing LISP hubcaps and just steal the whole CAR?

Let's start with some simple observations and strategies.

Strategy: Identify those aspects of the goal which have, at least to some extent, already been achieved. Then, brainstorm a way to leverage *what is* into *what can be* in much the same way that a survivalist might try to start a fire by using a magnifying glass to focus the sun's rays on kindling.

Observation: If you are marginalized by overwhelming forces, it might be time to stop thinking in terms of usurpation, and "free your mind instead". In some cases, "if you can't beat 'em, join 'em" is better advised than dogged persistence.

A politically incorrect comic strip I once saw depicted Tonto and the Lone Ranger surrounded by hostile Indians. The Lone Ranger says: "It looks like we're surrounded, Tonto!", to which Tonto replies: "Whadaya mean 'we', paleface?". This is a chameleon tactic. (Example: giving LISP a C syntax so it looks like a conventional language.)

A related tactic is the Trojan Horse subterfuge: this could be used to sneak LISP into the world without anyone realizing it's LISP. (Example: introducing closures into conventional languages: many don't have any clue as to their presence!)

In case it sounds like I'm advocating betrayal and selling out; I assure you I'm merely being realistic and adopting a pragmatic approach.

Consider the case of a donut maker marginalized by today's health trends. Extending his product line to include items preferred by health conscious patrons makes good sense. Converting his product line to fat free donuts does not. People are either going to avoid

donuts or indulge. Those who want to indulge will insist on the real thing; hence tasteless, fat free donuts are unlikely to appeal to either camp.

Another analogy: imagine a duck wanting to run like a cheetah. He trains at the track everyday, until finally realizing that ducks will never be able to run like cheetahs. However at that point the duck is no longer a good duck either, because all that running ruined the webbing in his feet!

I'm not advocating that LISP give up its essence in a go-along-to-get-along fashion! Rather, I'm advocating working with what we've got, and trying to transform it into what we want. I'm advocating education: if you build an advantageous mouse trap it's incumbent on you to explain to the world why it's advantageous.

Notice that I said "advantageous" rather than "better". Better is not always synonymous with advantageous. Inventing a better mouse trap is merely a first step. Then other issues need to be grappled with: pricing, politics, psychology, marketing, and accessibility also play into the ultimate success and acceptance of a programming language

Another consideration, given a goal of widespread acceptance of LISP, is: How will we know when we've arrived?

I once saw a play in which two people were expecting a Messiah and debating what the Messiah would look like. Meanwhile, some dunderhead was sweeping the floor while the play was going on. This annoyed some of the audience: "What are you doing? Why can't you do that some other time? Can't you see we're trying to watch an important play here?". Finally, one of the actors opined "I think the Messiah will be a knight in shining armor, and he'll ride in on a white horse.". At that point the "janitor" stood up and proclaimed: "There is no Messiah, and I'm it!".

If there is a corollary to the adage: "Be careful what you wish for, you might just get it!" perhaps it should be: "If you wish for something, make sure you are clever enough to recognize it when you see it!".

The Goal

When seeking a goal, perhaps it is best to have two goals: an interim goal and a long term goal. We might not arrive all at once!

→ Phase One:

Like others, I want to have a practical skill set which can assure me of employment anywhere in the world in any major city; and to be as recession proof as possible.

Given those requirements, I want to get as close to programming in LISP as possible; while still being squarely in the mainstream.

Here are my requirements for a Phase One Ubiquitous LISP:

1) It must target the most ubiquitous platform first; other platforms will then follow the lead (hopefully).

   If you do a search on Google for "windows market share" you'll find many links showing the prevalence of Windows.

   Windows has 49% of the server market, with second place going to linux servers at 25.7%, according to the following URL:

   http://www.entmag.com/news/article.asp?EditorialsID=5526

   The client side is even more heavily tilted to Windows at 87% according to:

   http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2862549,00.html

2) It must target the most ubiquitous application as a starting point. I believe the winner in that category is the lowly browser.

3) It must be as inexpensive as Red Hat Linux.

4) It must feature low level hardware support, OS support, I/O support, and tight integration with user interface facilities.

5) It must posses a small footprint, providing few but powerful primitives emphasizing simplicity, thriftiness, elegance, and flexibility (a la Scheme).

6) It must be extensible (a la Emacs).

7) It must provide a listener loop.

8) It must provide a "universe rather than a polyverse" environment: Blurring the distinction between the OS and the work environment into a seamless integration; wherein all applications are programmed in LISP, LISP machine language, or at least s-expressions; so that the user need only speak LISP.

9) It must be targeted to the business world. LISP *must* "curry" favor with the business world to secure cash cow support.

10) It must provide seamless browser/web integration and allow using the browser as a simple development environment in which to rapidly prototype user interfaces and applications. User interfaces must be creatable via DHTML (Dynamic HTML) which is a widespread and well accepted standard.

11) It must be targeted to home users to generate a grass roots following. This necessarily requires open protocols such that users can build their own custom machines. (Apple egregiously violates this precept!)

12) It must provide automatic garbage collection.

13) All elements of the language must be first class. In particular, functions/closures must be first class and be promotable to proper objects.

14) It must provide a built in object system. I consider encapsulation to be the most powerful element of the object paradigm for controlling complexity. The ability to create instances provides a necessary form of abstraction. Built in inheritance is not necessary as it can be simulated by using objects, closures, composition, and extending objects dynamically.

15) It must provide graphical HTML elements to allow plotting mathematical functions and graphical programming without the need of plug-ins.

16) No non-polymorphic syntaxes should be allowed. Syntax must be first class. No operator syntax!

17) All invokable routines must implicitly return a value.

18) Facilities for exception handling must be built in.

19) It must provide a tightly integrated, built in debugger.

Dynamics lists are not a requirement; but they are definitely a plus. The world is infatuated with gratuitously complicated, cryptic C syntaxes; hence LISP syntax would be a tough sell.

→ Phase Two:

I want to be able to build a custom computer from off the shelf parts; then buy a LISP OS for it for about $40.

Professionally, I want to do all programming in LISP.

Javascript and Dynamic HTML

One day, a funny thing happened on the way to the forum: I was asked (professionally) to write and maintain applications in Javascript, a new language for me. At that time, I had no idea how close to LISP Javascript is!

Javascript is an interpreted, garbage collected language natively supported by most web browsers. It is one of the pillars comprising DHTML (Dynamic HTML):

1) Javascript
2) CSS (Cascading Style Sheets)
3) DOM (Document Object Model)
4) HTML (HyperText Markup Language)

The DOM gives programmers access to the browser's document model. Previously, the elements of the DOM were specified statically via HTML: these elements are now programmatically accessible and creatable at runtime, including the event model.

CSS attempts to decouple form and substance; i.e. the specification of a document versus the appearance or presentation of it.

Browsers: the ubiquitous application and development environment.

Arguably, browsers are the world's most widespread computer application: nearly everyone has one and almost all of them support Javascript.

Javascript is tightly integrated with browser functionality, and is designed to facilitate gluing components together and linking them to the browser's user interface facilities.

Since browsers are so highly optimized for rendering user interfaces; languages like Visual Basic are becoming much less attractive for writing user interfaces, especially in light of the level of sophistication DHTML has attained. Browsers provide an inexpensive, ubiquitous development environment (including a debugger) for writing stand alone applications or client/server applications; and with great ease and small footprints!

Hence, if LISP were the development engine of browsers; that would be huge win for LISP.

There is no ubiquitous LISP, and Javascript is it!

As a language, Javascript is so expressively powerful; it can be used for writing entire applications with or without components. Indeed, because of the integrated script debugger; it is often tempting to write code in Javascript rather than in component modules (for ease of debugging).

In addition, there is no compilation, nor package and deployment process, nor linking, nor installation process; simply use any text editor to create DHTML, save it with a .htm extension, then double click on the file to launch it! Presto! No muss, no fuss. Sending code in e-mail is a simple matter of a small textual attachment.

Javascript, along with C# and VB; are the officially supported compiled languages in Microsoft's .Net environment (an environment which incidentally features a generation scavenging garbage collector inspired by LISP).

Javascript's expressive power comes from what is tantamount to a LISP engine with C syntax sans the dynamic lists which LISP is so famous for.

Javascript features a prototype-based object system after Scheme's own heart: elegantly simple, flexible and powerful. When leveraged with Javascript's first class higher order functions; the user has more than enough power to easily simulate virtually any preference in object programming style.

It would appear that in Javascript and DHTML, we've either achieved the Phase One goal, or at least we're in shooting range!

Code Sample

It's time now for a Javascript code sample. The following code sample is designed to run in Internet Explorer 6: the only browser I code in. It would need to be modified for other browsers.

```
<html>
   <head>
      <title>Javascript Test Form</title>
      <script>

      var windowOnLoad, clsRadioButtons, gobjRadioButtons, name;

      name = function(obj, strName) { obj.name = strName; };

      /*
      MS Bug warning:  Do not attach name attributes to radio buttons or they'll
      be unselectable!
      */
      clsRadioButtons =
         function() {
            var cls;
            cls =
               function() {
                  this.allowNoSelections = true;
                  this.collection = new Array(); };
            name(cls, "clsRadioButtons");
            cls.prototype.push =
               function(btn) {
                  this.collection.push(btn);
                  // Embed this keyword into a closure.
                  btn.attachEvent(
                     "onclick",
                     function(slf) {
                        return function() {
                           slf.setButton(btn); }; }(this)); };
            name(cls.prototype.push, "clsRadioButtons.prototype.push");
            cls.prototype.setButton =
               function(btn) {
                  var intIndex;
                  if(this.allowNoSelections || !btn.checked) {
                     btn.checked = !btn.checked; }
                  if(btn.checked) {
                     for(
                        intIndex = 0;
```

```
                    intIndex < this.collection.length;
                    intIndex++) {
                    if(this.collection[intIndex] == btn) { continue; }
                    this.collection[intIndex].checked = false; }}};
          name(cls.prototype.setButton, "clsRadioButtons.prototype.setButton");
          return cls; }();

    windowOnLoad =
       function() {
          gobjRadioButtons = new clsRadioButtons();
          gobjRadioButtons.push(document.all("radOption", 0));
          gobjRadioButtons.push(document.all("radOption", 1));
          gobjRadioButtons.push(document.all("radOption", 2)); };

    </script></head>

 <body
    onload="windowOnLoad()">
    <table>
       <tr>
          <td>Option 1</td>
          <td>Option 2</td>
          <td>Option 3</td></tr>
       <tr>
          <td><input
                id="radOption"
                type="radio"></td>
          <td><input
                id="radOption"
                type="radio"></td>
          <td><input
                id="radOption"
                type="radio"></td></tr></table></body></html>
```

The first observation you hopefully made from the above code sample is that it is written entirely in an object-centric style, rather than a procedural-centric style.

I like Javascript's object system, though I might propose some changes to it; but that is beyond the scope of this paper. (Hollywood tactic: leave room for a sequel!)

Javascript's object system is an object system after Scheme's own heart. It's elegantly simple, yet powerful enough to allow simulating virtually any extant object paradigm.

To make multiple instances of an object, define a constructor function in terms of the `this` keyword. When the `new` keyword is used, `this` will be bound to the current instance.

Anything defined on an object's `prototype` is visible to all instances created from the object; but can be shadowed by defining a local attribute of the same name. (This is useful for customizing different instances made from the same object.)

Notice the power being leveraged here by combining the object oriented paradigm with the best of the procedural paradigm: closures, which are essentially lightweight objects.

Objects are crafted to hide complexity: you merely tell the object what the HTML element is and it does everything for you. Methods are defined and attached to HTML elements dynamically at runtime. There is then a communication problem to be solved: the object is essentially a wrapper built on top of an HTML element, but the HTML element has no knowledge of the wrapper above it. It is crucial that it be in communication with it.

Closures are used to bridge this gap by embedding the knowledge of the wrapper into the event methods being attached to the HTML elements. Even if I am restricted to defining methods of 0-arity; that's no problem! Since the HTML element will typically be calling methods with the same arguments each time; those arguments can be embedded in a parental closure. That's also more efficient: we are passing the arguments only once!

I define, in both Scheme and Javascript, all functions via binding variables with closures. Javascript, like Scheme, offers two syntaxes for defining functions: `function fname(args)` vs. `fname = function(args)`. In Scheme this would be `(define (fname args)` vs. `(define fname (lambda (args)`. I consider the former to be an impure syntax, and I'm puzzled why [Abelson] would use it.

I agree with the immortal words of Alan Perlis: "Syntactic sugar causes cancer of the semicolon.". I am not opposed to macros to hide details, but I'm not in favor of creating multiple syntaxes for the same semantic structures. All types of functions can be defined by binding variables to closures, but not all types of functions can be defined via the impure syntax; hence it lacks what I would call "syntactic polymorphism". For instance if you define a function as being a function returned from a parental closure, the impure syntax cannot be used to define it. Worse yet, I believe the impure syntax causes tunnel vision: people who use it are often unaware that richer function definitions are possible.

In Javascript, there are primitives that allow access to stack information so that you can create stack traces to show what functions and arguments are on the stack: this is very useful information in exception handlers.

Functions not defined via the impure syntax will not show up on the stack because they are anonymous. To remedy this problem, I define a function called `name` which names functions even though they were created via closures.

This can be done in Javascript because functions are proper objects. This then prevents stack traces from degenerating into a list of anonymous functions calling anonymous functions.

Note also that functions in Javascript are (in a sense) more first class than in Scheme, because functions support the polymorphic method `toString` which allows access to a function's source code (as opposed to a listener merely printing out `#<procedure>`).

Javascript objects are essentially first class p-lists (that is to say, they can be created locally, globally, and anonymously). They are basically associative arrays. (In reality, Javascript arrays are simply specialized objects.)

Because of closures, currying (a la λ-calculus) is possible [vanMeule]. This can be a wonderful tactic to maintain signature polymorphism via reducing n-arity functions to match the arity of a method that wants a lesser arity.

Sometimes objects are a more heavy weight tool than is necessary. Javascript provides automatic conversion from primitive data types into objects whenever a primitive type is used as an object [Flanagan].

(Trivial nit: I never combine definitions of variables with initializations: I believe Javascript programs are clearer when these are separated out.)

Proposal: SchemeScript

I believe Scheme needs to spawn off a new language effort: SchemeScript. Given my proposal of the browser as the ultimate application I propose a browser written in LISP with a LISP engine at its core that supports all DHTML standards.

In addition, it should allow script blocks written in SchemeScript; and allow them to coexist and interact with Javascript code blocks on the same page. (Internet Explorer allows this for VBScript and Javascript.)

Scheme's current philosophy is to not add any new features until there is consensus as to what the right paradigm should be. This is noble, but denies Scheme the benefit of real world testing. "Repetition is the mother of learning." (Russian Proverb.) To get the repetition necessary for learning advancements, you need to accumulate a lot of test data from real world testing: the kind of real world testing that is only afforded to the most widespread tools.

While it sounds like a good plan to tell everyone to continue experimenting in those areas where it's unclear what the "right thing" is; if everyone creates different versions of Scheme, and they are all relatively obscure; not much is going to be learned from the rigors of the real world as a test bed from as many different domains as possible.

Therefore, the goal of SchemeScript would be to be of service to the real world rather than being an experiment in elegance; hence it would be much freer to experiment and make mistakes than Scheme allows itself.

What Javascript could learn from LISP

→ Some Javascript improvements would not break existing code.

The Javascript supported by the proposed SchemeScript browser could make several improvements to Javascript that would not break any existing code and would make Javascript much more powerful and semantically closer to SchemeScript:

1) All Javascript functions could be made to implicitly return a value.
2) `if` should be a first class function.

→ Javascript needs a macro facility.

It would make sense for Javascript to adopt Scheme's hygenic macro system just as Javascript adopted Perl's regular expressions. (If you see a good feature in another language, grab it!)

With the introduction of macros, closures could be used for block structure in the style of the LISP `let` macro.

→ Non-polymorphic syntaxes need to go.

I consider C style operator syntax as non-polymorphic syntax. Javascript could deprecate such syntaxes and support functional equivalents.

The built in sort method for arrays is a higher order function, which takes a comparison function that returns a number: negative indicates the first argument compares less, positive indicates it compares greater, and zero indicates equality.

This means that the subtraction function is the comparison function for ascending sort order on numbers and strings. However operators are not higher order and cannot be passed as arguments to higher order functions.

→ Dynamic lists.

It would be nice if Javascript supported dynamic lists, even though they can be created using the object system.

It would also be nice if Javascript adopted LISP's syntax because of the benefits to be had by a simple syntax.

Perhaps this could happen in stages: Javascript++ would get rid of non-polymorphic syntaxes and make other improvements that wouldn't break existing code. Javascript# could then adopt LISP syntax (hence Javascript# would essentially converge on SchemeScript).

What Scheme and SchemeScript could learn from Javascript.

SchemeScript must have exception handling: this is imperative for real world programming. Continuations may be elegant, but a simple try/catch/finally model might

be better advised in the short run until continuations are better understood. (In addition, many Scheme implementations punt on implementing continuations.)

SchemeScript must have low level stack primitives. This is a very nice facility in Javascript because access to stack arguments allows creating stack traces when exceptions occur.

Scheme has no built in object system. You can craft one but it won't be as efficient as a built in system. Also, different people will craft object systems differently, and human laziness will tempt some to program procedurally due to the "30-minutes-to-the-gym" syndrome (i.e. the more time it takes you to get to the gym, the less likely you are to go to the gym). The object system must be as simple and pervasive as possible; otherwise people won't use it in all situations where objects should be used.

S-Expressions: 60s retro or simple syntax?

The recent XML craze seems unaware that LISP has had XML since the 60s!

S-expressions are essentially the most minimalist form of XML possible. If you were to strip everything unessential from XML to reduce it to the smallest set of primitives necessary to bootstrap the rest; the resulting XML would be s-expressions.

[vanMeule] explored the use of closures as Turing complete: everything computable can be expressed in terms of closures; closures can also be used to create s-expressions and numbers (a la λ-calculus), and s-expressions can be used as a primitive form of data structuring.

I believe s-expressions should be a low level lingua franca and that many languages would benefit by being based on s-expressions for the sake of uniformity.

For instance, SQL needlessly diverges from other languages in both form and substance. This adds complexity to programming, rather than helping to control it.

However, like all good things, it is still possible to misuse and abuse s-expressions!

In early LISPs, before the advent of the object oriented paradigm, s-expressions were essentially used as a form of minimal XML and as data structures. In a pre-object oriented age, the procedural paradigm enhanced with s-expressions is very elegant.

Now that we have an object paradigm, procedural-centric code seems spaghetti coded by comparison.

When dynamic lists (i.e. s-expressions) are present in a language, there is a temptation to use them in contexts for which object oriented programming is the right tool.

For instance, here is a URL for a Scheme implementation in Javascript: http://www.bluetail.com/~luke/jscm/scm.js.txt . It is written in a procedural style even though Javascript supports a very adequate object model.

Given Javascript's object model; are s-expressions still useful? Absolutely!

There are some cases where s-expressions are the right weight for the job, and proper objects would be too heavy.

Javascript, like LISP, features `eval` and `apply`. `eval` can be used to promote data space into code space. There are times when it would seem easier to simply trust intelligent users and give them a large text area in an application where they can express what they want in terms of Javascript code.

However, Javascript isn't clean enough to serve as a good lingua franca for such expression. In addition, you might want to do some parsing on code entered textually before promoting it. (Perhaps you might want to restrict the user to a subset language.)

Parsing is much easier when the language has a simple syntax; therefore LISP's s-expression based syntax is the ultimate for applications wherein it may be desirable to promote data space to code space; and analysis and parsing of code space must be done.

If Javascript was a close enough cousin to LISP, it would be possible to make syntax negotiable and allow everyone to program in the syntax they prefer, without any coercion or usurpation of anyone's favorite mode of expression.

S-expressions make syntactic transformation as easy as possible by making syntax as simple as possible.

Therefore, s-expression based syntax should be the default and foundational syntax.

LISP initiatives, and how they could win better.

1) LISP machines.

   Downside: too proprietary and expensive. Hardware is more than adequate these days to support LISP on stock hardware.

   Upside: powerful, industrial strength. Features a uniform environment ("universe" rather than "polyverse").

2) Emacs

   Downside: too procedural. Must be object oriented. Must have a greater emphasis on graphical user interfaces. Must be lexically scoped.

Upside: inexpensive /free, runs on lots of platforms, powerful, extensible.

3) Scheme.

Downside: doesn't provide enough low level and graphical user interface support for crafting real world applications. Too perfectionistic: doesn't give itself enough freedom to add new features and modify or retract them as real world experimentation dictates. Lacks a built in object system.

Upside: inexpensive/free. Elegant.

Conclusion

"If he is a good wizard, he will serve." (Dorothy, from the Wizard of Oz.)

Most of the ideas I've presented have been very simple, common sense ideas. Nothing I've put forth is very lofty. Yet, it's the common sense things that will get you every time! That is why I mention them.

Consider how simple the wheel is, yet the Native Americans never invented it. Why? How would history have been different had they invented it? What "wheels" are we failing to invent?

I hope the LISP community will help cultivate Javascript (Near LISP!) and guide its development into a more mature language benefiting by what we've learned from LISP.

This can be accomplished by raising our voices, as a community, to the appropriate standards bodies; and by educating the public and contributing towards a common goal.

If Javascript can be cleaned up semantically, it would then be within "shooting range" of LISP, thereby becoming merely a syntactic alternative to LISP.

I would also like to see the LISP community focus on making LISP mainstream rather than improving it. If LISP was in the mainstream, it would have more test data to draw on in deciding improvements.

I believe we should try to dovetail with current technologies, build on them, build with them, and peacefully coexist with them as much as is possible.

The age of competition is over. This is an age of synergy and leveraging. This is an age of building economies of scale. That requires cooperation and specialization as preferable to head-to-head competition in already over crowded arenas.

Part of the success of Microsoft is due to their simple willingness to cater to the business world. Perhaps it's time to learn something from their marketing strategy.

I believe LISP can still become the lingua franca of computerdom and would benefit the world greatly by helping to control complexity in a world which is drowning in it.

Bibliography and References

Please note that [vanMeule] on-line articles at MacTech have some textual errors in them that were not in the originals.

[Abelson et al, 1996] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*, second edition. MIT Press, Cambridge, Massachusetts, USA, 1996.

[Flanagan, 2002] David Flanagan. *Javascript The Definitive Guide,* fourth edition. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[vanMeule Sep 93] André van Meulebrouck. *The Lambda Lambada: Y Dance?* (Mutual recursion.) MacTech Vol. 9, #9. LA, CA 90025-250055. ISSN 1067-8360. http://www.mactech.com/articles/mactech/Vol.09/09.09/Lambda/index.html

[vanMeule Jul 92] André van Meulebrouck. *Arbitrarily Large Bignums: Three Easy Substrates.* (Bignums.) MacTutor Vol. 8, #3. LA, CA 90025-250055. ISSN 8756-8810. http://www.mactech.com/articles/mactech/Vol.08/08.03/BigNums/index.html

[vanMeule Apr/May 92] André van Meulebrouck. *Deriving Miss Daze Y.* (Deriving the Y combinator.) MacTutor Vol. 8, #1. LA, CA 90025-250055. ISSN 8756-8810. http://www.mactech.com/articles/mactech/Vol.08/08.01/DazeY/index.html

[vanMeule Jun 91] André van Meulebrouck. *Going Back to Church.* (Church numerals.) MacTutor Vol. 7, #6. Anaheim, CA 92807. ISSN 8756-8810. http://www.mactech.com/articles/mactech/Vol.07/07.06/ChurchNumerals/index.html

[vanMeule May 91] André van Meulebrouck. *"A Calculus for the Algebraic-like Manipulation of Computer Code", or "Why Oh Why Oh Y?".* (λ-calculus.) MacTutor Vol. 7, #5. Anaheim, CA 92807. ISBN 8756-8810. http://www.mactech.com/articles/mactech/Vol.07/07.05/LambdaCalculus/index.html