# On Self Interpretation in the Myself Programming Language

Josh Cough
Northwestern University

---

This document is a reasonably well organized set of notes on the implementation of a self interpreter in a programming language called Myself, which is an extension of the FAE programming language found in [Krish].

## On Purpose

The Myself programming language has one purpose - to explore self interpretation. At the start of the project there was one (rather vague) goal in mind. **To determine the smallest set of features required to enable practical self interpretation while minimizing change in representation.** What that statement really means is still up for debate, however I believe I've come understand it more now, having implemented a Myself interpreter in Myself. One purpose of this document is to explain the statement further in order to remove those ambiguities. There are several vague parts of the statement that need clarification:

1. What does the 'Smallest set of features' really mean?
2. What is really meant by 'practical'?
3. What does it mean to 'minimize' change in representation?

## On Myself

Myself is a language originally implemented in PLT Scheme. It's grammar is as follows:

```
id = ; any valid scheme symbol
num = ; a scheme number
exp = number
          | id
          | (sym id)
          | (+ exp exp)
          | (- exp exp)
          | (with (id exp) exp)
          | (if0 exp exp exp)
          | (my-list exp*)
          | (my-car exp)
          | (my-cdr exp)
          | (my-cons exp exp)
          | (same? exp)
          | (numb? exp)
          | (symb? exp)
          | (is-list? exp)
          | (is-empty? exp)
          | (my-print? exp)
          | (fun (id*) exp)
          | (exp*) ; function application
```

For the remainder of this document, I'll call the original implementation of Myself, "Myself-K1". The

use of k here comes from [Kogge]. The original parse and interp functions (written in Scheme) will be called myself-k1-parse and myself-k1-eval. The second version of Myself will be called Myself-K2, and it's parse and eval functions called myself-k2-parse and myself-2-eval, which are written in Myself-K1. Sometimes, due to laziness, I'll use simply K1 and K2 to mean Myself-K1 and Myself-K2.

## On Parsing

### On Code Representation in Myself-K2

The first important question is: What syntax will be used? Or, what representation am I going to parse? Answer: It can only be things that the language can understand. In particular, symV, numV, listV, closureV. But, we probably wont pass the parser closures. That doesn't really answer the question though, but it gets us closer. First lets explore what code looks like in Scheme:

```
(+ 6 7)
```

is a list containing the symbol '+ and the numbers 6 and 7.

Each of those things can be represented in Myself, however a little bit more verbose:

```
(my-list (sym +) 6 7)
```

This brings up a few questions:

1. Why cant Myself-K2 use exactly the same representation as Myself-K1?

   To answer that question is to really understand how Myself code is interpreted. Assuming that there exists parse and eval functions in the Myself-K2 library, then one should call them like so:

   ```
   (eval (parse (my-list (sym +) 6 7)))
   ```

   But lets first explore what it would mean if instead they were called like so:

   ```
   (eval (parse (+ 6 7)))
   ```

   The key here, is remember that this is Myself-K2 code that needs to be parsed and evaluated by the Myself-K1 parser and evaluator. That *is* the runtime for Myself-K2. Remembering this, we expand the code then to:

   ```
   (myself-k1-eval (myself-k1-parse (eval (parse (+ 6 7)))))
   ```

   And knowing what we already know about evaluation in myself-k1-eval, we can reduce this to:

   ```
   (eval (parse (numV 13)))
   ```

   This finally answers the question of why we can't use that original representation. (I'm not being 100% complete here, but that's because I believe the material is mostly understood.)

   (+ 6 7) is not a list in Myself-K2, it's a function call in Myself-K1. myself-k2-parse needs to work on lists. The only way to represent lists is to use my-list. So we go back to the original:

```
(my-list (sym +) 6 7)
```

The story is still somewhat incomplete however.

2. Why is the sym function needed?

   Why can't we say this:

   ```
   (my-list + 6 7)
   ```

   The answer lies also in the original parser (myself-k1-parse). When it encounters the +, it sees it as a bare symbol. It treats bare symbols as ids and passes (id +) to myself-k1-eval in the AST. When myself-k1-eval interprets the list, it interprets each of the items in the list. When it comes across (id +), it attempts to look it up in the environment. It's not there, and the evaluation fails.

   When (sym +) is used, the parser parses this as symbol creation, instead of an id. In scheme, it's the difference between x and 'x. Unfortunately Myself doesn't have such an easy syntax.

3. How do we represent ids in Myself-K2?

   The answer to this is quite simple.

   ```
   (sym the-id-you-want)
   ```

   As long as it appears in the right location in your code, it will properly be parsed as an id. For example:

   ```
   (my-list (sym +) (sym x) (sym y))
   ```

   adds the ids x and y.

4. But then how do you represent symbols themselves?

   To answer that, we need to know about function application. In Myself-K2, function application looks like so:

   ```
   (my-list (sym +) (sym x) (sym y))
   ```

   But we already know that, we've just seen it. That's ok, it is the key to creating symbols. (sym +) is very much like a function application. sym is not technically a function, its a special form, but its the same syntax so you can consider it to be. Therefore, we use the following syntax to create the symbol 'x:

   ```
   (my-list (sym sym) (sym x))
   ```

   This is brutal, I know. In Scheme it's 'x. In K1 it's (sym x) and in K2 it's all that nonsense above.

5. Could quasiquote help here?

   Almost definitely, and it's certainly worth exploring. It would be the first TODO on a long list of things that could improve the language.

### Last notes on representation

I stated that instead of (+ 5 6), we use (my-list (sym +) 5 6). While this is true, it deserves a little more treatment. What the myself-k2-parser is really accepting is the output of myself-k1-eval. (my-list (sym +) 5 6) is evaluated to (listV (symV '+) (numV 5) (numV 6). However, it's not possible to call the myself-k2-parser with those values. The only way to create those values is through Myself-K1 code. Therefore, the initial syntax, as explained, is the real syntax for K2. It seems a bit strange to me, still, that the syntax of the language depends somehow on both myself-k1-parse and myself-k2-parse. I think it's going to take a bit of time before I completely understand that. If this isn't explained well its for that reason.

Let's take a second to restate the goal: **To determine the smallest set of features required to enable practical self interpretation while minimizing change in representation.** Or, let's not change the representation too dramatically. Does this qualify as dramatic? Yes and No. It's ugly, but it should be clear from the explanation above that these both have the same meaning (in K1 vs K2), just that the original implementation can rely on Schemes reader, and Myself can't. This means that, if we did have a more sophisticated parser, its possible that we could get close to the same syntax. However, in order to do that, we'd probably have to add more power to K1. But the next goal of this project was to determine the smallest set of features needed to enable self-interpretation. At this point we realize that the goals are in fact, in direct opposition to one another. To use the same representation you must add features to the language. To get the smallest set of features you must remove features from the language. I think I've found an okay balance between the two opposing forces. The code at times, is quite ugly. But, I've already added more features than I originally anticipated. I think this fits the definition of good compromise - both sides are unhappy. :)

### Myself-K2 Grammar

With the notes above, we have enough information to lay out the grammar for Myself-K2.

```
[id] = (sym s) ; where s is any valid scheme symbol
[num] = n ; where n is a scheme number
[exp] = [num]
          | [id]
          | (my-list (sym sym) [id])
          | (my-list (sym +) [exp] [exp])
          | (my-list (sym -) [exp] [exp])
          | (my-list (sym if0)[exp] [exp] [exp])
          | (my-list (sym my-list) [exp]*)
          | (my-list (sym my-car) [exp])
          | (my-list (sym my-cdr) [exp])
          | (my-list (sym my-cons) [exp] [exp])
          | (my-list (sym same?) [exp])
          | (my-list (sym numb?) [exp])
          | (my-list (sym symb?) [exp])
          | (my-list (sym is-list?) [exp])
          | (my-list (sym is-empty?) [exp])
          | (my-list (sym my-print?) [exp])
          | (my-list (sym fun) [id] [exp])
          | (my-list [exp] [exp]) ; function application
```

## On AST Representation

The next important question is: what is the AST going to look like? Fortunately, this is a much easier question to answer than that of syntax representation. The AST must be a legal Myself-Val because it will be the output of the parser.

AST nodes will be tagged lists where the first item in the list indicates the type of the expression. This tagged list style comes from [Abelson et al]. For simplicity, I omit the outer list for the listings below. (contents) really means (listV (list contents)).

```
((symV 'num) (numV 7))
((symV 'sym) (symV 'something))
((symV '+) lhs rhs)
((symV '-) lhs rhs)
((symV 'id) (symV 'something))
((symV 'if) test then else)
((symV 'fun) id body)
((symV 'app) f a)
((symV 'list) x1 ... xn)
((symV 'my-car) lst)
((symV 'my-cdr) lst)
((symV 'numb?) x)
((symV 'symb?) x)
((symV 'is-list?) x)
```

## On Evaluation

Writing the eval function in Myself was actually quite reasonable, and was done in around 50 lines, including comments. The process was quite simple - check the data type (the first element of the AST node) and then take action on it.

A typical example follows:

```
{with {type {1st expr}} {with {body {my-cdr expr}}
              ...
              ; numb? ((symV 'numb?) x) -> numV (0 or 1)
              {if0 {same? type {sym numb?}} {if0 {numb? {EVAL {1st body} env}} 0 1}
              ...
```

Here, type is the first element of the AST node, and were checking to see if it is numb?. If so, eval the body itself, and simply check to see if it is a number.

I should add some important notes, however. In order to enable such an easy eval function, features had to be added to the Myself language. That was a huge part of this project - figuring out what features needed to be added (above and beyond the features in the original FAE language). There were several:

1. a numb? predicate
2. symbols, and a symb? predicate
3. equality check for symbols and numbers
4. lists, and is-list? is-empty? predicates
5. list helper functions - car, cdr and cons

In FAE, lists were represented as functions. Why couldn't they be in Myself? Because you need a list? predicate (is-list?). If lists are functions, then there is no way to distinguish between lists and functions.

Looking back that myself-k2 parse and eval, I actually didn't really need to distinguish between them. So, it might actually be time to revisit this a little.

However, using pair from FAE to write code was not sufficient. The variadic nature of the list function was an absolute requirement for writing code. The first (now abandoned) Myself implementation used pair instead of list, and the code was a disaster. All code became long chains of pairs that completely cluttered the actual meaning of the code, and took it from impractical to impossible.

I'll almost certainly have more to say here in the future.

## Revisiting Goals

Let us revisit some of the questions posed earlier in relation to the goals of the project. Hopefully now we have enough information to attempt to remove some of the ambiguities.

1. What does the 'Smallest set of features' really mean?
2. What is really meant by 'practical'?
3. What does it mean to 'minimize' change in representation?

### On Practicality

We'll handle them a bit out of order. Let's start with the definition of practical. Here is how we apply the identity function to the value 7, in Myself-K1.

```
{{fun {x} x} 7}
```

and in K2...

```
(my-list (my-list (sym fun) (sym x) (sym x)) 7)
```

Even though we can don't yet have a definition of practical, its probably see that this is bordering on impractical. No one would really want to write any code in this language. Does that mean 'practical' is a 'gut feeling' sort of thing? Yes and no. We could say that the original K1 is practical (that is certainly subjective), and then say that anything longer than that is impractical. I suppose that building a definition off of a subjective premise isn't the best approach, but I think it works. The first version of the identity function seems practical, and the second version is much longer, and so we can judge it as impractical.

Does that mean this part of the project is a failure? No, because the goal was to aim for practical self-interpretation, and the K2 parse and eval functions are about as long as the K1 versions.

### On Minimizing Change in Representation

To remove ambiguities from this statement, we first need to consider what we mean by representation. We could mean syntax or semantics. While the identity example above has a significant difference in syntax, it's semantics are identical in every way. So which do I mean? To be honest, I originally meant syntax. I did want the code to come out looking nearly the same. Unfortunately, it isn't. But that's the definition I'm sticking with, so I will reword it - Minimizing Change in Syntax.

I think quasiquote can help achieve this goal, and it needs to be implemented.

The next version of Myself can include quasiquote, and then this paper can be redone with that ambiguity removed. I'm comfortable with this, as this paper itself reflects the learning process.

**On the Smallest Set of Features**

The next question concerns what it means to have the smallest set of features in the original language, in order to implement the self interpreter. I've already explained how minimizing change and the smallest set of features are in direct opposition, but its worth exploring a little bit further. The question to ask is, can any of the features in K1 be removed? And the wonderful answer is - probably. However, would it change the representation? Probably. Dramatically? Possibly. Would it make the syntax of K2 less practical? Almost definitely. Would it make the implementation of myself-k2-parse and myself-k2-eval less practical? Very likely.

Once again, I think I've managed to strike a pretty nice balance on all the goals. Features we're added to Myself only when it appeared absolutely necessary in implementing myself-k2-parse and myself-k2-eval. So when we ask what it means to have the smallest set of features, we really need to ask what it means to have the smallest set of features without compromising practicality.

I think quasiquote could help here immensely. It's one feature that we could add to the original language that has the potential to improve practicality, and bring us much closer to the original implementation in terms of syntax.

## References

Alebeson, H. and Sussman, J. 1996. Structure and Interpretation of Computer Programs, 2nd Edition. The MIT Press.

Kogge, P. The Architecture of Symbolic Computers. 1991. McGraw-Hill, Inc.

Krishnamurthi, S. 2007. Programming Languages, Application and Interpretation. Krishnamurthi, S.