My first document

John Doe

2013-09-01

1 All about Macros

1.1 the BAD-OR macro

Let us define a two argument BAD-OR macro in terms of LET and IF

```
(defmacro bad-or (a b) `(let ((tmp ,a)) (if tmp tmp ,b)))
```

Now let us put the macro to work in an environment where TMP is bound

```
(let ((tmp 5))
  (bad-or #f tmp))

=> (let ((tmp 5))
     (let ((tmp #f))
      (if tmp tmp tmp)))
```

We can ignore outer let as the binding it set up cannot be seen by anything

```
=>  (let ((tmp #f))
       (if tmp tmp tmp))
```

by substitution this is

```
=>  (if #f #f #f)
```

resulting in the conclusion

```
=>  #f
```

Ideally we would want the following to be true

```
(let ((tmp 5))  =>   5
  (or #f tmp))
```

1.2 syntactic closures

Let us return to the BAD-OR macro

```
(defmacro bad-or (a b) `(let ((tmp ,a)) (if tmp tmp ,b)))
```

Let us look at introduced variables

```
tmp
```

Now let us suppose everything in the language is in a state of flux and can be redefined at any time.

```
(defmacro bad-or (a b) `(let ((tmp ,a)) (if tmp tmp ,b)))
```

Parameters of BAD-OR are A and B , what if they were named LET and IF ?

```
(defmacro silly-bad-or (let if) `(let ((tmp ,let)) (if tmp tmp ,if)))

(defmacro bad-or (a b) `(let ((tmp ,a)) (if tmp tmp ,b)))

(let ((tmp 5))
  (bad-or #f tmp))
Renamed =>
 (let:0 ((tmp:1 5))
   (bad-or:0 #f tmp:1))
```

2 defmacro

```
(defmacro my-unless (condition &body body)
  `(if (not ,condition)
     (progn
       ,@body)))
```

2.1 free variable capture

The problem is when inadvertant variable capture occurs there is no telling when or how the bug will manifest itself. An almost impossible task to debug this sort of problem.

If one bad macro is allowed then all other macros that depend on that will become susceptible to

3 kohlbecker

4 macro stepper and debugger

5 continuation passing macros

6 scrap

```
[1]> (defmacro bad-or (a b) `(let ((tmp ,a)) (if tmp tmp ,b)))
BAD-OR
```

So this says whenever i say

```
(bad-or ?a ?b)
```

replace that with

```
(let ((tmp ?a)) (if tmp tmp ?b))
```

Lets try it in a few cases

```
[5]> (let ((a 1)(b 5)) (bad-or a b))
1
[6]> (let ((a nil)(b 5)) (bad-or a b))
5
```

So everything appears to be working normally.

```
:: (let ((a 1)(b 5)) (bad-or a b))
=> (let ((a 1)(b 5)) (let ((tmp a)) (if tmp tmp b)))
1
[6]> (let ((a nil)(b 5)) (bad-or a b))
5
```

```
[2]> (macroexpand '(bad-or nil tmp))
(LET ((TMP NIL)) (IF TMP TMP TMP)) ;

T
[3]> (macroexpand `(let ((tmp 5)) ,(macroexpand '(bad-or nil tmp))))
(LET ((TMP 5))
 (LET ((TMP NIL))
  (IF TMP TMP TMP))) ;
NIL

[3] (let ((tmp 5)) (bad-or nil tmp))
=> NIL

[5]> (if nil nil nil)
NIL
```