Posted on **April 11, 2015**

Before we go into our more in depth examples, I want to provide some basic building block macros.

Defering and evaluating macros are going to be used a lot and are really important to be discussed. The goal of defer is to delay the evaluation of a macro. Evaluation macros are going to evaluate macros. We will make use of combinations them heavily to achieve many things. Lets first look at what the use case of the macro will be

```
#define FOO(x, y) x y
#define BAR() x, y

/*
  The following expression results in an error
  This is because FOO is expanded first
  As a result, Since Foo only has 1 parameter Bar
  It complains we didnt provide the correct arguments
*/
FOO(BAR())

/*
  Using defer allows BAR() to be evaluated first.
  However, since we used the defer, FOO is not evaluated
*/
DEFER(FOO) (BAR()) // Expands to FOO(x, y)

/*
  To get around this, we will need an evaluation macro.
  Calling EVAL will allow us to evalated the DEFER expression
*/
EVAL(DEFER(FOO) (BAR())) // Expands to x y

/*
  We might also want to defer twice
  if we have two levels of evaluation needed
*/
#define QUX() BAR()

// Error: The following
// expands to FOO(BAR()) and FOO is evaluated
EVAL(DEFER(FOO) (QUX()))

EVAL(DEFER2(FOO) (QUX())) // Expands to x y
```

The definition of DEFER and EVAL is actually relatively simple, we can make use of our rules discussed previously to make it work. The idea of DEFER is to require n evaluations for the deferred expression/macro to actually get called. We can use our EMPTY macro that we first saw in the rules section. By adding EMPTY() calls in between a macro call and its parameters we can defer the execution of FOO by however many EMPTY's we have.

```
// EMPTY() expands to nothing
#define EMPTY()

FOO(x, y) // x y
// We require an extra evaluation to evaluate FOO
FOO EMPTY() (x, y) // FOO(x, y)
// We require two extra evaluations to evaluate FOO
FOO EMPTY EMPTY() () (x,y ) // FOO EMPTY() (x, y)
```

This pattern can be implemented in the DEFER macros.

```
// __VA_ARGS__ is the expression to defer
#define DEFER(...) __VA_ARGS__ EMPTY()
// To defer it twice we simply add in another defered EMPTY() call
#define DEFER2(...) __VA_ARGS__ DEFER(EMPTY) ()
#define DEFER3(...) __VA_ARGS__ DEFER2(EMPTY) ()
#define DEFER4(...) __VA_ARGS__ DEFER3(EMPTY) ()
#define DEFER5(...) __VA_ARGS__ DEFER4(EMPTY) ()
// .
// .
// .
// Can have this go to however many defer stages you need
// This can easily be autogen'd
```

Next, we want to implement an EVAL macro which evaluates the macro. This will come later when we explore recursion but the main goal is we want to have a sub-expression evaluated many times. The way this works is by relying on Rule 3. By having an empty macro that simply returns our arguments we can have an expression evaluated once. Stack the calls to these macros and we can evaluate many more times. If we have an EVAL call another EVAL twice and have that EVAL call another EVAL, we can have 2^n evaluations of an expression where n is how many times we stack. This is easier to show in code

```
#define EVAL_1(...) __VA_ARGS__
// Note how we call EVAL of the lower level twice
// This allows us to double the number of EVAL calls per level
#define EVAL_2(...) EVAL_1(EVAL_1(__VA_ARGS__))
#define EVAL_3(...) EVAL_2(EVAL_2(__VA_ARGS__))
#define EVAL_4(...) EVAL_3(EVAL_3(__VA_ARGS__))
#define EVAL_5(...) EVAL_4(EVAL_4(__VA_ARGS__))
#define EVAL_6(...) EVAL_5(EVAL_5(__VA_ARGS__))
#define EVAL_7(...) EVAL_6(EVAL_6(__VA_ARGS__))
#define EVAL_8(...) EVAL_7(EVAL_7(__VA_ARGS__))
// Finally our EVAL calls the top level EVAL call
#define EVAL(...) EVAL_8(__VA_ARGS__)
// ...
// This can easily  autogen'd
```

Note, if have a macro that is evaluated using EVAL which calls another macro that uses an EVAL then that EVAL wont be expanded (Rule 4). This can be seen in the following example.

```
#define BAR(x) EVAL(x)
EVAL( DEFER(BAR)(x) ) // Expands to EVAL(x)
```

We would need to create an appropriate EVAL chain for macro if they are going to be chained together

```
// Assuming EVALA and EVALB are defined appropriately
#define BAR(x) EVALA(x)
EVALB( DEFER(BAR)(x) ) // Expands to x
```