

# COMP 511 Class 4 Notes (Spring 2005)

Reproduced by James Sasitorn

January 19th, 2005

While it is possible to use a language like perl to implement a multistage language using string manipulation, it leads to a more complicated multistage programming model. Since, a string representation doesn't prevent a programmer from performing program transformations, accidental complications can happen (whether we do them intentionally or not).

Listed below is an example illustrating the problems that can arise in implementing a multistage using string manipulation. **Note: this program actually works. This shows that coming up with a good example is a bit tricky. (This is still a good staging example though).**

```
let rec f n x y =
  if n=0 then y
  else let z = x + y
       in f(n-1) x z

f 3 5 7
= f 2 5 (5 * 7= 35)
= f 1 5 (f*25 = 125)
= f 0 5 875
```

Assume n and x are early

```
let rec f n x y =
  if n=0 then y
  else <let z = x + ~y
       in ~(f(n-1) x <z>>>

<fun a -> ~(f 3 5 <a>>>
= <fun a -> ~<let z = 5 * a
              in ~(f 2 5 <z>>>>
= <fun a -> ~<let z = 5 * a
              in ~<let z=5* z
                  in ~(f 1 5 <z>>>
```

This example worked out because the scope of z is used immediately before the next binding occurrence of z. A working counter-example is in chapter 4 of MSP (assuming you read ahead 2 chapters). The problem occurs because of recursion. The example below avoids the use of letrec to emphasize that this recursive predicament does not relate to letrec recursion (assuming a first order letrec construct).

```
fun h n z = if n=0 then z else (fn x ? (h (n-1) (x+z))) n
           = ...                else let x = n in h (n-1) (x+z)

h 3 4
= h 2 (3+4=7) = h 1 (2+7=9) = h0 (1+9=10) = 10
```

In this example, z is early with type <int>.

```

fun h1 n z = if n=0 then z
             else <let x = n in ~(h (n-1) <(x~z)> ) >

h 3 <4>
= !<fun a -> ~(h 3 <a>>>
= !<fun a -> ~ <let x=3 in ~(h 2 <x + ~<a>>>
= !<fun a -> ~ <let x=3 in <let x=2 in ~(h 1 <x+a>>>>
= !<fun a -> ~ <let x=3 in ~<let x=2 in ~<let x=1 in ~(h 0 <x + (x + (x+a))>>>>>>
= !<fun a -> ~ <let x=3 in ~<let x=2 in ~<let x=1 in ~(<x + (x + (x+a))>>>>>>
= !<fun a-> 3 + a> 4 = 7

```

**The staged version gives u 10, so its clearly faster. You wont find this anywhere, thank god.**

This discrepancy is caused by our dependence on strings. In the example, brackets would be quotations and would be spliced using string concatenation. So before evaluation, our staged function is:

```
<fun a -> let x=3 in let x=2 in let x=1 in x+(x+(x+a))>
```

In a language without cross-stage scoping, the variable x is being captured by inner bindings. The correct solution is to generate a unique name every time we need to generate piece of code with a bound variable.

```
<fun a -> let x_1=3 in let x_2=2 in let x_3=1 in x_3+(x_2+(x_1+a))>
```

Weird things happen with side effects, but we aren't going to talk about them now. Specifically, every time you have a binding construct, you need to think of it as a template and in a sense deal with this x as twiddle hat, as a special construct, that splices in a new name in both the binding position and the usage position.

```
"let x= newSymbol in"
```

While it is possible to do this by hand, the analysis is non-trivial. You will always have to do explicit renaming and always keep thinking about explicit naming. Depending on your implementation, things may get further complicated if you have a name string and a code string.

With bracket and escape, you can simply think of them as delayed. This static typing tells you when it needs to be run. Consider the following:

```
<1 + 2> =? <3>
```

You can construct values in different ways. In MetaOCaml, are these two considered equal? While they have different runtime performance, they are equivalent. The reasoning follows that of mathematical functions. Are the following equal?

```
f_1(x) = x + x and f_2(x) = 2*x
f_1 =? f_2
```

In mathematics, checking for equality on functions is undecidable. Similarly, in MetaOCaml you can't test equality on pieces of code- equality can be used only on ground types. This notion of equivalence is absent in a string representation:

```
"1 + 2" != "3"
```

An aside remark: Macros are really two stage computations. (1) Macro expansion and (2) Normal computation.