Metaprogramming — Nada Amin

Lecture 1 - *June 27, 2025*

1 Introduction

Metaprogramming is writing programs that manipulate programs. Some common examples are

- interpreters, where the input is a program;
- compilers, where both the input and output are programs;
- transformers;
- analyzers;
- discovery engines.

Metaprogramming is a foundational idea. It is deeply related to the concept of computation itself, since the universal computing machine first introduced the idea that a program is just data that can be manipulated – A turning machine that can interpret another turning machine –. Metaprogramming is useful because you can build domain-specific languages for tooling, but it is also principled; for example, you can mechanically turn an interpreter into a compiler.

2 Scheme interpreter

Here are a few components of a very simple interpreter. But first, let us define some Scheme conventions. We will use functions to represent our environments; thus, an environment is a function. Each function is responsible for the lookup of a single variable. Therefore, if the input variable name is equal to the variable defined in the current function, we return its value. Otherwise, we continue the lookup process by calling another environment. Another adopted convention is the naming of predicates. Predicates are functions that perform some check, in other words, they return a boolean. We use the suffix ? while naming predicates.

```
;; the empty environment
(define empty-env
  λ (y) (error 'empty-env (format "unbound variable ~s")))

;; a tagged expression
(define tagged?
  (λ (tag)
    (λ (e)
      (and (pair? e) (eq? (car e) tag)))))

;; evaluation function
(define evl
  (λ (exp env)
    (cond
      ((number? exp) exp)
                ;; case for number
      (((tagged? '*) exp)
        (* (evl (cadr exp) env) (evl (caddr exp) env)))
                ;; case for multiplication
      (((tagged? 'λ) exp)
        (let ((x (car (cadr exp)))
              (body (caddr exp))))
        (λ (a)
            (evl body (λ (y) (if (eq? y x) a (env y))))))
                ;; case for functions
      (else
          ((evl (car exp) env) (evl (cadr exp)))))))
                ;; case for application

(evl '(* 2 3) empty-env)
> 6
(evl '((λ (x) (* x 3)) 2) empty-env)
> 6
```

Notation note: If you look at the case for quotes (`(((tagged? 'quote) exp))`), this indicates a notation `'2` or `(quote a)`.

Important remarks

- If you have a list of things (1 (2 a b) 3 4),
  - `car` gives the first item
  - `cdr` gives the tail
  - `cadr` is `car` composed with `cdr`
  - `caadr` is `car` composed with `car` composed with `cdr`
- If we look up in the empty environment, we just emit an error.

Some elaboration on the interpreter above

- Closures are represented as a function. In the `((tagged? 'λ) exp)` case above, for example, we say the name of the variable is x, and define the function body. Only once we have a can we evaluate the body.
- To see that the code works, we need the application case, where we assume we have a pair `((evl (car exp) env) (evl (cadr exp)))`.

3 Metacircular interpreter

So now we have written a small interpreter for some Scheme programs. We can scale this up to an interpreter that is truly metacircular – a Scheme interpreter written in Scheme –.

- When we execute a lambda, we push a new frame onto the stack with the new bindings. We pop it off when we are done with the body;
- We need a table mapping symbols to underlying scheme language;
- We can run the metacircular interpreter, load its own source code, and create a stack of frames. The CPU time increases drastically for every frame.

4 Recursion

```
1 (define factorial
2   (λ (n)
3     (if (= n 0)
4         1
5         (* n (factorial (- n 1))))))
```

If we run this function, we see a list of nested stack frames.

```
(trace factorial)
(factorial 6)
> | (factorial 3)
  | | (factorial 2)
  | | | (factorial 1)
  | | | | (factorial 0)
  | | | | 1
  | | | 1
  | | 2
  | 6
  6
```

If instead, we defined factorial iteratively, using an accumulator, as in

```
(define fact-iter
  (λ (n acc)
    (if (= n 0)
        acc
        (fact-iter (- n 1)  n acc)))))
```

We see only one frame

```
> | (fact-iter 3 1)
  | (fact-iter 2 3)
  | (fact-iter 1 6)
  | (fact-iter 0 6)
  | 6
  6
```

We can use **continuation-passing style (CPS)** to mechanically transform a recursive process into an iterative one. In CPS, we reify the rest of the computation as a continuation, thus we can perform a single step and yield the result to the continuation which knows what to do next.

```
(define factorial-cps
  (λ (n k)
    (if (= n 0)
        (k 1)
        (factorial-cps (- n 1) (λ (v) (k (* n v)))))))

(factorial-cps 6 (λ (x) x))
```

Now we get a series of calls, like the iterative version, but not a stack. The idea of CPS is that we send the value of our current execution to a "continuation," and never return. In this code, we make a new recursive call and give it the continuation `λ (v) (k (* n v))`, which sends $n *$ `factorial-cps (- n 1)` to the current continuation. We don't need to keep the current stack frame because we never return.

5 call/cc

`call/cc` does a function call with the current continuation.

```
> (* 2 (call/cc (λ (k) (+ (k 3) (k 6)))))
6
> (* 2 (call/cc (λ (k) (+ (k 3) (k 7)))))
6
> (λ (hole) (* 2 hole))
<procedure>
```

Scheme has a call-by-value evaluation order; thus, when evaluating a function application, we must first evaluate its arguments. In this example, 2 is already evaluated, so next the thing to evaluate is: `call/cc (λ (k) (+ (k 3) (k 7)))`. `call/cc` reifies the current evaluation context as continuation – $*$ 2 $\square$ –, and exposes it as k. You may think that a continuation is a special function that does not return when invoked. So, during the evaluation of this term, when we do the first k call – `(k 3)`–, we immediately jump to our previous evaluation context ($*$ 2 $\square$) and discard the remainder computation, and immediately plug in 3 and evaluate $2 * 3$. This allows us to escape the current expression and provide a value to plug into the current continuation; thus, the k call k 6 never gets evaluated.

6 Trampolining

```
(define factorial-tra
  (λ (n k)
    (λ ()
        (if (= n 0)
            (k 1)
            (factorial-tra (- n 1) (λ (v) (k (* n v))))))))

(define driver
    (λ (p)
        (while (procedure? p)
            (set! p (p)) p )))
```

Trampolining inserts thunks on each call. So `factorial-tra` only resumes execution on each invocation, not forcing the full evaluation. One advantage of trampolining in this case is that you get the benefit of no stack overflows. We can use a driver whose purpose is to repeatedly force the evaluation of `factorial-tra` until termination.

7 CPS in our interpreter

We only need to perform minor changes to make your interpreter follow a continuation-passing style

- Add a parameter `k` to the `evl` input.
- Instead of returning, invoke `k`.

When you have two recursive calls, you have to make a choice about the order of evaluation and set up the correct continuation to follow the evaluation. So,

```
(define evl
  (λ (exp env k)
    (cond
      ((number? exp) (k exp))
      (((tagged? '*) (k exp))
        (* (evl (cadr exp) env)
            (λ (v1) (evl (caddr exp) env
            (λ (v2) (k (* v1 v2)))))))
      (((tagged? 'λ) exp)
        (let ((x (car (cadr exp)))
              (body (caddr exp)))
          (k (λ (a cont)
            (evl body (λ (y) (if (eq? y x) a (env y))) cont)))))
      (else
          ((evl (car exp) env) λ (f)
            (evl (cadr exp) env) (λ (v) (f v k)))))))
```

Now we can even add `call/cc` to our interpreter.

```
(((tagged? 'call/cc) exp)
 (evl (cadr exp) env
      (λ (f)
        f (λ (v k0) (k v)) k))))

(* 2 (call/cc (λ (k) (+ (k 3) (k 7)))))
> 6
```

Note that the argument of `call/cc` is anything that evaluates to a procedure.

8 Reflection

From the program, we might want to be able to reason and operate with the program itself. We can adopt a methodology of using two operations: *reify*, to transform our current code into data – go one level up –, and *reflect* to transform data back into code – go down one level –. By using this methodology, we can create a distinction between the base language and the languages that can operate on the programs of the base language. Therefore, with those operations, we can make a reflective tower of interpreters, in which we can freely move between its levels of meta languages.

- reify: program → data
- reflect: data → program