Discussion 12

Macros, Tail Calls

Aditya Balasubramanian
aditbala [at] berkeley [dot] edu

Slides available at `teaching.aditbala.com`

Announcements

- Online office hours before Thanksgiving:
  - Thursday 11/17 3pm-5pm on oh.cs61a.org.
  - Monday 11/21 2pm-5pm on oh.cs61a.org.
- Lab 12 due Wednesday 11/16.
  - A walkthrough playlist should help if you're stuck (@berkeley login required).
- Homework 8 due Thursday 11/17 (extended).
  - A guide video should help if you're stuck.
- Scheme due soonish...
- Go to cs61a.org for more announcements

Motivating Macros

- Consider this function which should execute `f` twice

```
scm> (define (twice f) (begin f f))
twice
scm> (twice (print 'woof))
woof
```

- Why is this only printing `woof` once?

Programs as Data Solution

```
scm> (define (twice f) `(begin ,f ,f))
twice
scm> (twice '(print 'woof))
(begin (print (quote woof)) (print (quote woof)))
scm> (eval (twice `(print 'woof)))
woof
woof
```

- Pass in `f` as unevaluated and substitute it into our expression

Macros Solution

```
scm> (define-macro (twice f) (list 'begin f f))
twice
scm> (twice (print 'woof))
woof
woof
```

What's new?

- operands remain unevaluated
- `eval` is called on resulting list

Macros summary

- Evaluate operator
- Apply operator to unevaluated operands
- Evaluate the expression returned by the macro in the frame it was called in.

Tail Recursion 

Tail Recursion

- What is Tail Recursion?
  - Recursion while keeping track of our result, instead of accumulating back up at the end
- How do we make our function Tail Recursive?
  - Usually have a helper function to keep track of information from previous frames

Tail Calls

- What are Tail Calls?
  - The recursive calls in Tail recursion
- When do they occur?
  - When a function calls another function as the final action of the current frame
  - This also means that the current frame can be discarded (different than normal recursion)
- Why Tail Recursion?
  - More efficient
  - Less Space

Consider this implementation of `factorial` that is NOT tail recursive:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

- Recursive call is on the last line, but is not the last expression to be evaluated.
- Must accumulate the result of `n * recursive calls` at the end

Visualization of NON tail-recursive `factorial`

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Tail recursive `factorial`:

```
(define (factorial n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

- `fact-tail` makes a single recursive call to `fact-tail`, and that recursive call is the last expression to be evaluated

Visualization of tail-recursive `factorial`

```
(factorial 6)
(fact-tail 6 1)
(fact-tail 5 6)
(fact-tail 4 30)
(fact-tail 3 120)
(fact-tail 2 360)
(fact-tail 1 720)
(fact-tail 0 720)
720
```

Tail Context

- To determine if a function call is a Tail Call, we have to look for Tail Contexts
- Following expressions are Tail Contexts
  1. the second or third operand in an `if` expression
  2. any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
  3. the last operand in an `and` or an `or` expression
  4. the last operand in a `begin` expression's body
  5. the last operand in a `let` expression's body
- What is the tail call in `(begin (+ 2 3) (- 2 3) (* 2 3))`
- `(* 2 3)` is a tail call because it is the last operand expression to be evaluated

Thank you!!!

Anon Feedback -> https://tinyurl.com/adit-anon