Lecture 3 — August 10, 2015

- Today:
  - The substitution model
  - Iteration and tail calls
  - Block structure
  - The let special forms
- Readings:
  - *SICP:* Finish Section 1.3, get started on Section 2.1

CSE130 Summer Session II

More Special Forms

- Both logical `and` and `or` ***short circuit:***

```
(and <e1> ... <en>)
```

```
(or <e1> ... <en>)
```

- Evaluate the expressions one at a time, left-to-right
- For `and`:
  - If any *expr* evaluates to false, the value of the `and` is false and the rest of the *exprs* are not evaluated
  - If true, its value is the value of the last *expr*
- For `or`:
  - If any *expr* evaluates to true, that value is returned and the rest of the *exprs* are not evaluated
- `not` does not short circuit: it is an ordinary procedure, not a special form

The way we've been evaluating expressions is through ***applicative order:***

- Evaluate the operator and operands first, and then apply the procedure to those arguments

An alternative is ***normal order:***

- Evaluate operands only when their values are needed.

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

Loops

- There are no special forms or procedures in order to loop.
- We already have what we need: Recursion!

```
(define (sum n)
  (if (<= n 0)
    0
    (+ n (sum (- n 1)))))
```

The Substitution Model

Rules for expression evaluation in the substitution model:

1. If ***self-evaluating***, (e.g., a number) just return that value.
2. If a ***name***, replace with values associated with that name
3. If expression is a `lambda`, create procedure and return
4. If expression is a special form, (e.g., `if`, `and`) follow specific rules for evaluating sub-expressions
5. If expression is a compound expression, then:
   - Evaluate all of the sub-expressions of combination (in any order)
   - If procedure is primitive, just do it
   - If procedure is compound procedure (created by `lambda`), substitute value of each sub-expression for corresponding procedure parameter in body of procedure, then repeat on body

Substitution Model in Action

```
(define (fact n)
  (if (= n 1) 1 (* n (fact (- n 1)))))
```

```
(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

Deferred Tasks

- The evaluator deferred multiplications while it worked on solving recursive sub-problems:

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24
```

- So, space required is O(?)

Iterative Algorithms

- An iterative algorithm uses constant space

```
(define (ifact n)
  (ifact-helper 1 1 n))

(define (ifact-helper product counter n)
  (if (> counter n)
      product
      (ifact-helper (* product counter)
                    (+ counter 1)
                    n)))
```

Iterative Algorithm: Evaluation

```
(ifact 3)
(ifact-helper 1 1 3)
(if (> 1 3) 1 (ifact-helper (* 1 1) (+ 1 1) 3))
(ifact-helper 1 2 3)
(if (> 2 3) 1 (ifact-helper (* 1 2) (+ 2 1) 3))
(ifact-helper 2 3 3)
(if (> 3 3) 2 (ifact-helper (* 2 3) (+ 3 1) 3))
(ifact-helper 6 4 3)
(if (> 4 3) 6 (ifact-helper (* 6 4) (+ 4 1) 3))
6
```

- No growing list of pending operations
- Partial answers are accumulated
- The "last thing" a procedure does is call itself

Tail Calls and Tail Position

- During evaluation, a procedure is replaced by the last thing it does. Here, **`ifact-helper`** returns the value, *not* **`ifact`**!

```
(define (ifact n)
  (ifact-helper 1 1 n))
```

```
(ifact 3)
(ifact-helper 1 1 3)
```

- A call is in ***tail position*** if it is: *[Dybvig, 3.2]*
  - The last expression in the body of a **`lambda`** expression
  - The consequent or alternative part of an **`if`** expression in tail position
  - The last sub-expression of an **`and`** or **`or`** expression in tail position
  - The last expression of a **`let`** in tail position

Tail Call Examples

- Each of the calls to `f` (in the expressions below) are tail calls
- But the calls to `g` are not.

```
(lambda () (f (g)))
(lambda () (if (g) (f) (f)))
(lambda () (or (g) (f)))
(lambda () (and (g) (f)))
```

- Remember: *IASEVIBTVOTEEISITP*
  - If a sub-expression's value immediately becomes the value of the entire expression (if the sub-expression is evaluated at all) it is in tail position.

Block Structure

- **defines** can be nested at the *top* of procedure bodies:

```
(define (ifact n)
  (define (ifact-helper product counter n)
    (if (> counter n)
        product
        (ifact-helper (* product counter)
                      (+ counter 1)
                      n)))
  (ifact-helper 1 1 n))
```

- Now `ifact-helper` is visible only to `ifact` and no one else.

Lexical Scope

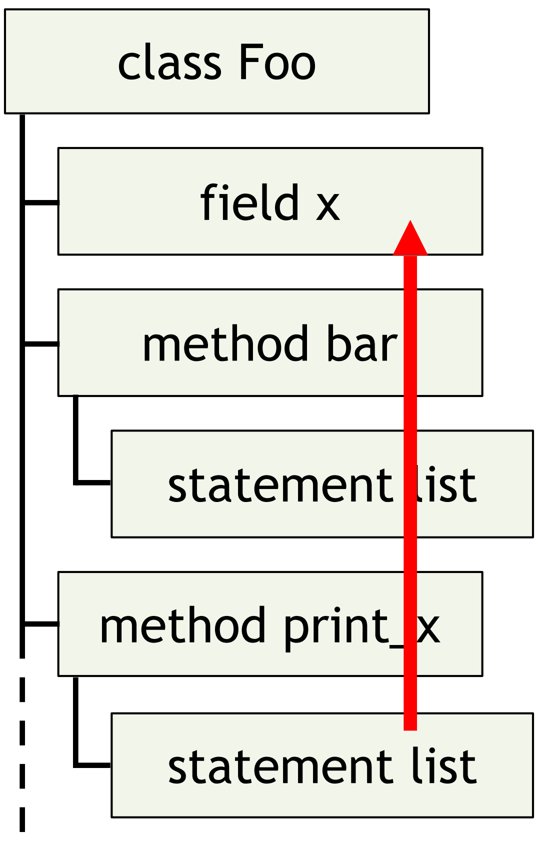
- This nesting follows ***lexical scoping*** rules.
- So, we don't need to pass **`n`** to **`ifact-helper`**:

```
(define (ifact n)
  (define (ifact-helper product counter)
    (if (> counter n)
        product
        (ifact-helper (* product counter)
                      (+ counter 1))))
  (ifact-helper 1 1))
```

Semantics of Scoping

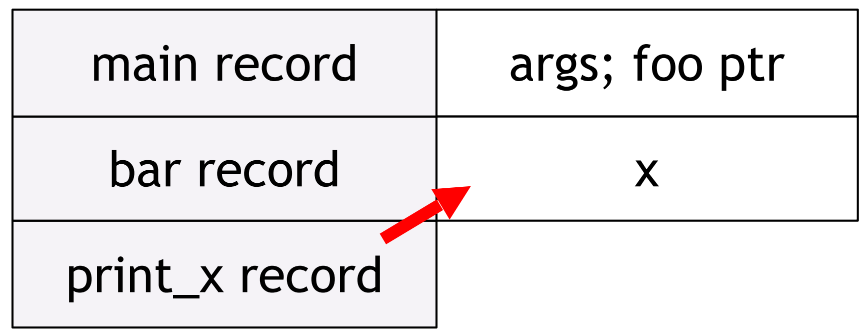
- Static scope:

looks up the syntax tree at compile/parse time



- Dynamic scope:

looks up the dynamic call stack at runtime



Static vs. Dynamic Scope

```
class Foo {
  static int x = 20;

  void bar() {
    int x = 10;
    print_x();
  }

  void print_x() {
    System.out.print(x);
  }
}
```

If we call **bar**, what value will **print_x** print using:

- Static scope?
- Dynamic scope?

The benefits of naming

- Values that are used multiple times in a procedure can benefit from being named (why?)
- *E.g.*, $f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x a a)
     (* y b)
     (* a b)))
```

The `let` Special Form

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x a a)
     (* y b)
     (* a b)))
```

Can be expressed as:

```
(define (f x y)
  (let ([a (+ 1 (* x y))]
        [b (- 1 y)])
    (+ (* x a a)
       (* y b)
       (* a b))))
```

Note: Using `[]`s instead of `()`s is just a Racket feature to help with visual grouping. They are interchangeable.

The `let` Special Form

Just syntactic sugar:

```
(define (f x y)
  (let ([a (+ 1 (* x y))]
        [b (- 1 y)])
    (+ (* x a a)
       (* y b)
       (* a b))))
```

for:

```
(define (f x y)
  ((lambda (a b)
     (+ (* x a a)
        (* y b)
        (* a b)))
   (+ 1 (* x y)) (- 1 y)))
```

Scoping of initializers in `let`

- The syntactic sugar definition actually explains something about the scope of new variables introduced by `let`:

```
(define (f n)
  (let ((n (+ n 1)))
    n))
```

```
(f 3)    → 4
(f 0)    → 1
```

`let`'s let functions

- `let` can be used to define any kind of data, including functions:

```
(define (odd? n)
  (let ((even (lambda (n)
                (zero? (remainder n 2)))))
    (even (- n 1))))
```

```
(odd? 3)
(odd? 0)
```

→ **#t**

→ **#f**

No circular references in `let`

- **But*!*** the functions we define with `let` cannot be recursive:

```
(define (ifact n)
  (let ((ifact-iter
          (lambda (product counter)
            (if (> counter n)
                product
                (ifact-iter (* product counter)
                            (+ counter 1))))))
    (ifact-iter 1 1)))
```

`ifact-iter: unbound identifier in module in: ifact-iter`

The `letrec` Special Form

- You can use another form, `letrec`, for such circular dependencies:

```
(define (ifact n)
  (letrec ((ifact-iter
            (lambda (product counter)
              (if (> counter n)
                  product
                  (ifact-iter (* product counter)
                              (+ counter 1))))))
    (ifact-iter 1 1)))
```

The named-**let** Special Form

- This use of **letrec** is common enough to warrant its own special form, the named-let syntax:

```
(define (ifact n)
  (let ifact-iter ((product 1) (counter 1))
    (if (> counter n)
        product
        (ifact-iter (* product counter)
                    (+ counter 1)))))
```

- This use of **letrec** is common enough to warrant its own special form, the named-let syntax:

The `let*` Special Form

`let` cannot handle linear dependencies:

```
(define (g n)
  (let ([a (* 2 n)]
        [b (* a a)])
    (+ a b (* a b))))
```

(But Racket allows this for `letrec`, in addition to the newer Scheme standard)

but `let*` can…

```
(define (g n)
  (let* ([a (* 2 n)]
         [b (* a a)])
    (+ a b (* a b))))
```

…which is equivalent to nesting

```
(define (g n)
  (let ([a (* 2 n)])
    (let ([b (* a a)])
      (+ a b (* a b)))))
```