implicit heap-dynamic variables

- automatic heap-allocation
- JavaScript, [pg. 214]:
  
  ```javascript
  list = [10.2, 3.5];
  list = 47;
  ```

- Scheme
  - all allocation is done on heap
  - cons allocates a pair
  - environments are allocated on heap (so no runtime stack is needed for function calls).
;;; lambda-over-let

(define count1
  (lambda ()
    (let ((k 0))
      (set! k (+ k 1))
        k
    )))

(display (count1)) (newline)
(display (count1)) (newline)

/* OUTPUT IS:
1
1
1
*/
Environment before evaluation of `(count1)`: 

(let ((k 0))
 (set! k (+ k 1))
 k)

(count1)
Evaluation of the body of the function results in the binding of k changing from 0 to 1; this 1 is returned as the value of the function call (count1).
Environment after evaluation of (count1):

(let ((k 0))
  (set! k (+ k 1))
  k)

Note that the environment which contained the binding of k to a value does not persist.

This is because there was no closure referring to that environment.
;;; let-over-lambda

(define count2
  (let ((k 0))
    (lambda ()
      (set! k (+ k 1))
      k
    ))
)

(display (count2)) (newline)
(display (count2)) (newline)

/* OUTPUT IS:
1
2 */
Environment before evaluation of (count2):

count2’s closure persists the environment (invocation record) in which k is bound.
Environment during evaluation of (count2):

(set! k (+ k 1))

(k)

(lambda ()
  (set! k (+ k 1))
  k
)

k

(count2)

(set! k (+ k 1))

k

k

1
Environment after evaluation of (count2):

Note that the environment which contains the binding of k to a value persists.

This is because there is a closure referring to that environment (the one for count2).
Notes about example

- in count1 k is initialized each time function is called
- in count2 k is initialized when the function is defined
Exercise

• How do you define a Scheme function which can dynamically generate counter functions, each with its own independent counter?
counterMaker

(define counterMaker
  (lambda ()
    (let ((k 0))
      (lambda ()
        (set! k (+ k 1))
        k)
    )))

(define c1 (counterMaker))
(define c2 (counterMaker))
(c1) (c1) (c1) (c2) (c2) (c1) (c2)
> (define c1 (counterMaker))
> (define c2 (counterMaker))
> (c1)
> 1
> (c1)
> 2
> (c1)
> 3
> (c2)
> 1
> (c2)
> 2
> (c1)
> 4
> (c2)
> 3

Try it out!
6. Scope

• “The scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.”

• Two basic approaches:
  – static scoping
  – dynamic scoping
Static scope

Static scoping (also called lexical scoping): the scope of a variable is determined by the static (lexical) structure of the program.

– scope determined by lexical nesting
– some languages allow nested blocks
  • we’ve seen examples in C and Java
– some languages allow nested functions
  • C/C++/Java do not allow this
  • Scheme/ML do
Dynamic scope

- dynamic scope
  - scope determined by (dynamic) call history
  - we will see an example soon
Why do we care?

• Obviously important for our understanding of programs.
• More importantly, we use the scoping of variables to support encapsulation.
• Strong encapsulation lets us build robust components:
  – grants access to those components which need it
  – denies access to all other components
Collection classes

Each collection defines its own Iterator.

Each collection needs to keep its implementation details private.

Each iterator is defined as a separate class (so it can be instantiated independently of the class).

Each iterator encapsulates iteration logic for a specific collection. It must know the implementation details in order to serve as the bridge between clients of the collection and the internal workings of that collection.

How do we grant access to Iterator classes while denying access to others?
Inner classes

• Java’s Collection classes
  – concrete collection class must hide its implementation
  – must provide Iterator
  – Iterator must know implementation details
  – Inner class structure embeds Iterator definition within scope of collection class
  – Iterator can access internals of collection without collection having to break encapsulation
Nested functions

• Create nested scopes to hide bindings
• Allows us to hide helper functions (similar to what inner classes let us do):

```scheme
(define fact
  (letrec ((helper
            (lambda (n acc)
              (if (= n 0)
                acc
                (helper (- n 1) (* n acc))))))
    (lambda (n)
      (helper n 1))))
```
Examples comparing static and dynamic scope

- Example in C-like language
- Example in Scheme-like language
Dynamic vs. static scope: C-like language

```c
int a = 3;
void foo(){ printf("a has value %d \n",a); }
void bar(){ int a = 5; foo();}
foo();  // what does this print?
bar();  // what does this print?
```
With static scope

```c
int a = 3;
void foo(){ printf("a has value %d \n",a); }
void bar(){ int a = 5;  foo();}
void abc(){ int a = 7;  foo();}
foo();  // what does this print?
bar();  // what does this print?
abd();  // what does this print?
```

C is statically scoped, so the output will be:

```
a has value 3
a has value 3
a has value 3
```
With dynamic scope

```c
int a = 3;
void foo(){ printf("a has value %d \n",a); }
void bar(){ int a = 5; foo();}
void abc(){ int a = 7; foo();}
foo(); // what does this print?
bar(); // what does this print?
abd(); // what does this print?
```

If C were dynamically scoped, the output would be:

```
a has value 3
a has value 5
a has value 7
```
Languages with dynamic scope

- Common Lisp and Perl give the programmer the option of using either dynamic or static scope.
Dynamic vs. static scope: Scheme-like language

(define a 3)
(define f (lambda () a))
(define g (lambda (a) (f)))
(f)
(g 7)
Environment:

(define a 3)
Environment:

(define a 3)
(define f
  (lambda ()
    a)
)
(define a 3)

(define f
  (lambda ()
    a
  ))

(define g
  (lambda (a)
    (f)
  ))
Environment during evaluation of (f):

\[
\begin{align*}
&(\text{define a } 3) \\
&(\text{define f}
\begin{align*}
&\quad (\text{lambda}(())
\begin{align*}
&\quad \quad a
\end{align*}
\end{align*}\\
&)) \\
&(\text{define g}
\begin{align*}
&\quad (\text{lambda}(a)
\begin{align*}
&\quad \quad (f)
\end{align*}
\end{align*}\\
&)) \\
&(f)
\end{align*}
\]
Environment during evaluation of \((g \ 7)\):

\[
\begin{align*}
\text{(define } a & \ 3) \\
\text{(define } f & \ (\lambda () \ a) \\
\text{(define } g & \ (\lambda (a) \ (f)) \\
(f) & \ (g \ 7)
\end{align*}
\]
Questions?