Functional Programming - 3

- Detour: short explore of static and dynamic scoping
  - Locally declared variables versus heap stored variables
    - Some slides co-developed with Dr. Alex Borgida, Rutgers University
- Tail recursion
- Closures
- Streams

Lexical Scoping

- Block structured PLs
  - Allow for local variable declaration
  - Inherit global variables from enclosing blocks
  - Local declarations take precedence over inherited ones
    - Hole in scope
  - Lookup for non-local variables proceeds from inner to enclosing blocks in inner to outer order.
  - Used in Algol, Pascal, Scheme (with *let*), C++, C, Java
    - Some languages historically were "flat" with no nested procedure declarations (e.g., C)
    - Let's in Scheme allow this construct

Example

Visibility of procedures/functions is the same as visibility of variables.

```
program
   a, b, c: integer;
   procedure p
      c: real;
      procedure s
         c, d: integer;
         procedure r
         …
         end r;
         r;
      end s;
      r;
      s;
   end p;
   procedure r
      a: integer;
      = a, b, c;
   end r;
   …; p; r; …
end program
```

Q: Which procedure r is called here? (the call `r;` inside s)

Q: Which variables a, b, c are read here? (`= a, b, c;` in r)

Q: Which procedure r is called here? (`…; p; r; …`)

Example - Block Structured PL

```
program
   a, b, c: integer;
   procedure p
      c: integer;   main.a, main.b, p.c      main.p(), p.s(), main.r()
      procedure s
         c, d: integer;  main.a,main.b,s.c,s.d
         procedure r                          main.p(), s.r(), p.s()
         …  s.r(),p.s(),main.p()
         end r;
         r;
      end s;
      r;
      s;
   end p;
   procedure r
      a: integer;     r.a, main.b, main.c      main.r(), main.p()
      = a, b, c;
   end r;
   …; p; …
end program
```

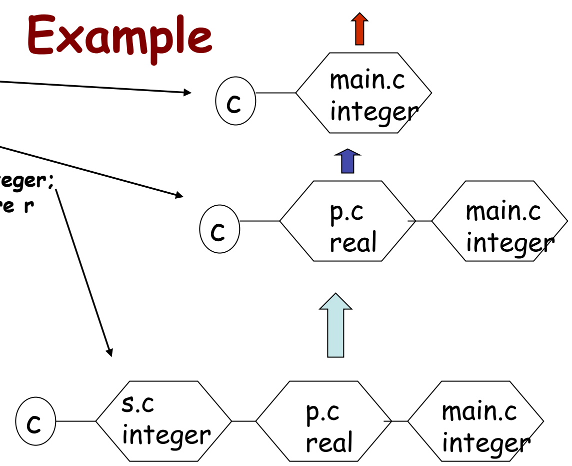
nested block structure allows locally defined variables and functions

Symbol Table

- Must support insertion, deletion, lookup of names
- For lexical scoping, need to use a stack for storing attributes of a variable (to handle hole-in-scope)
- Need to update as enter and leave a block at compile time (during translation)
- Used by compiler and debugger (at runtime)

Example

```
program
   a, b, c: integer;
   procedure p
        c: real;
        procedure s
                c, d: integer;
                procedure r
                …
                end r;
                r;
        end s;
        r;
        s;
   end p;
   procedure r
        a: integer;
        = a, b, c;
   end r;
   …; p; …
  end program
```



Q: How do these stacks become updated as execution proceeds in the debugger?

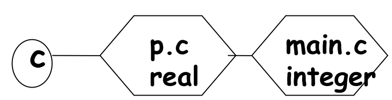
Dynamic Scoping

- What if declarations are entered into the symbol table as they are encountered at runtime?
  - Declarations are processed as they are encountered on an execution path
  - Lookup for non-local variables proceeds from closest dynamic predecessor to farthest
  - Or if variables are dynamically typed by their usage (as in Scheme and Prolog)
- Used mainly in interpreted PLs (e.g. Perl, APL)

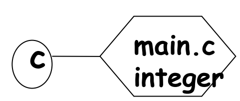
Example

```
program
   a, b, c: integer;
   procedure p
        c: real;
        procedure s
                c, d: integer;
                procedure r
                …
                end r;
                r;
        end s;
        r;
        s;
   end p;
   procedure r
        a: integer;
        = a, b, c;
   end r;
   …; p; …
  end program
```

Dynamic scoping
Main calls main.p() calls main.r():



Static scoping in main.r():



Example

```
main{
   procedure Z(){
        a: integer;
        a := 1;
        Y();
        output a;
   }//end Z;
   procedure W(){
        a: integer;
        a := 2;
        Y();
        output a;
   }//end W;
   procedure Y(){
        a := 0; /*1*/
   }//end Y;
   Z();
   W();
}//end main
```

Which a is modified by /*1*/ under dynamic scoping? Z.a or W.a or both?

Example

```
main{
   procedure Z(){
        a: integer;
        a := 1;
        Y();
        output a;
   }//end Z;
   procedure W(){
        a: integer;
        a := 2;
        Y();
        output a;
   }//end W;
   procedure Y(){
        a := 0; /*1*/
   }//end Y;
   Z();
   W();
}//end main
```

main calls Z,
Z calls Y,
Y sets Z.a to 0.

Example

```
main{
  procedure Z(){
      a: integer;
      a := 1;
      Y();
      output a;
  }//end Z;
  procedure W(){
      a: integer;
      a := 2;
      Y();
      output a;
  }//end W;
  procedure Y(){
      a := 0; /*1*/
  }//end Y;
  Z();
  W();
}//end main
```

main calls W,
W calls Y,
Y sets W.a to 0.

Is this program legal under static scoping? If so, which a is modified? If not, why not?

Example

```
main{
  procedure Z(){ /*4*/
      a: integer;
      a := 1;
      W();
      /*9*/ Y();
      output a;
  }// end Z/*10*/
  procedure W(){ /*5*/
      a: integer;
      a := 2;
      Y();
      output a;
  }//end W/*8*/
  procedure Y(){/*6*/
      a := 0;
  }//end Y/*7*/
  /*3*/ Z();
}//end main
```

table entry for a at:

/*3*/	empty top
/*4*/	&(Z.a) ←
/*5*/	&(W.a), &(Z.a)
/*6*/	$\boxed{\text{&(W.a)}}$, &(Z.a)
/*7*/	&(W.a), &(Z.a)
/*8*/	&(Z.a)
/*9*/	&(Z.a)
/*6*/	$\boxed{\text{&(Z.a)}}$
/*7*/	&(Z.a)
/*10*/	empty

Two Versions of Scope

```
(let ((x 10))
    (let   ((f (lambda (a) (+ a  x))))
        (let ((x  2))
            (*  x  (f  3) ) ) )
```

Will it evaluate to

- (* x (lambda (a)(+ a *x*) 3) -->
  (* 2 ((lambda (a)(+ a 10) 3)) --> 26

"lexical scoping"

or

- (* x (lambda (a)(+ a *x*) 3)) -->
  (* 2 ((lambda (a)(+ a 2) 3)) --> 10

"dynamic scoping"

Scheme chose lexical scoping model

Example - Scheme

```
((lambda (x)
    ((lambda (y)
        ((lambda (z)
    (+ x y))
          5)
       4)
   3)
```

evaluates to 12

```
(let ((x 3))
   (let ((y 4))
      (let ((z 5))
   (+ x y))))
```

also evaluates to 12

(let ((x 2)) (+ x ...)) is just an abbreviation for
((LAMBDA (x) (+ x ...)) 2)

Tail Recursive Functions

- If the result of a function is computed without a recursive call OR if it is the result of an immediate recursive call, then the function is *tail recursive*
  - E.g., foldl
- Tail recursive functions are efficient, because the result is *accumulated* in one of the arguments (saves space)
  - Don't need a stack to compute tail recursive functions!

Two Defns of Length function

```
(define (len ys)
  (if (null? ys)
      0
      (+ 1 (len (cdr ys)))))
(len '(3 4 5))
```

Len *not tail recursive*

```
(define (lentr ys tot)
  (if (null? ys)
      tot
      (lentr (cdr ys)
             (+ 1 tot))))
(define (len2 ys)
  (lentr ys 0))
(len2 '(3 4 5))
```

Lentr is *Tail recursive*

Tot is used to accumulate the length calculation

Tail Recursive Factorial

```
(define (fact n)
  (cond
   ((zero? n) 1)
   ((eq? n 1) 1)
   (else (* n
      (fact (- n  1))))))
```

fact is original version

```
(define (factor n acc)
 (cond ((zero? n) 1)
       ((eq? n 1) acc)
       (else (factor (- n 1)
        (* n acc)))))

(define (factorial n)
   (factor n 1))
```

factor is tail recursive version

Closures

- A *closure* is a function value plus the environment in which it is to be evaluated
  - Sometimes need to include variables not local to the function so closure can eventually be evaluated
- A **closure** can be used as a function
  - Applied to arguments
  - Passed as an argument
  - Returned as a value

Closure Bindings are 'Immortal'

- Normally, when execution exits a let or a function, its bindings disappear.
- If those bindings are part of a closure
  - When the let exits they become inactive but are not destroyed
  - They become active whenever (and wherever) the closure is called

f is constant 10 function

```
(let ((x 5))
   (let  (( f  (let ((x 10))  (lambda (y)  x )  ))
     (list x (f 1) x (f 1)) )  )      yields (5 10 5 10)
```


Evaluation of Closures

```
(define (gg   z)
  (let* ((x   2) (f  (lambda(y) (+ x y)))) (map  f  z)))
```

gg is actually a closure which is (lambda (z) (map f z)) where the defining environment is { x → 2; f → (lambda (y) (+ x y))}
we need this environment to evaluate gg.

\>(square 2)

4 ; is assumed to be evaluated in the context of the empty environment {}

\>(gg '(1 2 3))

1. value of gg is its closure
2. closure environment is expanded by argument association with parameter { x → 2; f → (lambda (y) (+ x y)); z → '(1 2 3) }
3. evaluation occurs and (3 4 5) is returned

More on Evaluation of Closures

(define ff (lambda (x) (* 2 x))) ; binds ff to a closure

If evaluate ff, the system will print something like this:

(lambda (a1) ...) showing its value is a closure

If evaluate (ff), the system will complain about a missing argument

If evaluate (ff 3) the system will return 6.

Currying, revisited

- What's going on?
  - *We are reducing n-ary functions to n applications of unary functions*
  - Can always do this, so n-ary functions don't add more power to your language

    $+ : R \times R \rightarrow R$,   curried+ : $R \rightarrow (R \rightarrow R)$

    (define (curried+ x) (lambda (y) (+ x y)))

    ((curried+ 2) 3) yields 5

    (let ((f (curried+ 1))) (f 10)) yields 11

Currying (and Closures)

```
>(define (mm x  y) (* x  y))
>mm ; returns a closure
>(mm 2)  ; returns error because mm expects 2 arguments, not 1!
>(mm 2 3) ; returns 6
```

```
>(define hh (lambda (x) (lambda (y) (* x y))  ))
>hh ; returns (lambda (a1) ...)
>(hh 2 3) ; not called in a curried manner, one argument at a
time, returns
    hh: expects only 1 argument, but found 2
>((hh 2) 20) ; proper way to call a curried fcn of 2 args
40
>(hh 2) ; hh plus 1 argument returns a closure
    (lambda (a1) ...)
```


Streams

- A mechanism to generate an unbounded number of elements in a sequence
- Involves putting a function value as an element in a list and then executing that function to produce a sequence of values

(define (stream f n) (cons (f n) (list f))) ; encodes value of f applied to n as first element of the list and f as the rest of list

(define (head str) (car str)) ; head retrieves the next value that is stored as the first element of list

(define (tail str) (cons (apply (car (cdr str))
                                                    (list (car str))) (cdr str))) ;
tail constructs a new list with the next value as its car and the generating function as it cdr.

Example

```
(define (stream f n) (cons (f n) (list f)))
(define (head str) (car str))
(define (tail str)
    (cons  (apply (car (cdr str))
             (list (car str))) (cdr str)))
(define (square x) (* x x))
(define ss (stream square 2))
(head ss)
>4
(head (stream (lambda(x)(* 2 x)) 5))
>10
(tail ss)
> (list 16 (lambda (a1) ...))
(head (tail ss))
>16
(head (tail (tail ss)))
>256
```