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.

Example - Block Structured PL

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:

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

```
main.c
  integer
p.c
  real
s.c
  integer
main.c
  integer
```

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

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

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

```
C
p.c
real
main.c
integer
```

**Static scoping in main.r():**

```
C
main.c
integer
```
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 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(){
    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

Table entry for $a$ at:
/*3*/ empty top
/*4*/ &($Z.a$)
/*5*/ &($W.a$), &($Z.a$)
/*6*/ &($W.a$), &($Z.a$)
/*7*/ &($W.a$), &($Z.a$)
/*8*/ &($Z.a$)
/*9*/ &($Z.a$)
/*10*/ empty
Two Versions of Scope

(\( \text{let } ((x 10)) \)
  (\( \text{let } ((f \text{ lambda } (a) (+ a x))) \)
    (\( \text{let } ((x 2)) \)
      (* x (f 3)))))

Will it evaluate to

- \( (* x (\text{lambda } (a)(+ a x)) 3) \rightarrow \)
  \( (* 2 ((\text{lambda } (a)(+ a 10)) 3) \rightarrow 26 \)

or

- \( (* x (\text{lambda } (a)(+ a x)) 3) \rightarrow \)
  \( (* 2 ((\text{lambda } (a)(+ a 2)) 3) \rightarrow 10 \)

Scheme chose lexical scoping model

"lexical scoping"

Example - Scheme

(((\text{lambda } (x))
  (((\text{lambda } (y))
    (((\text{lambda } (z))
      (+ x y))
    5)
  4)
3)
(\( \text{let } ((x 3)) \)
  (\( \text{let } ((y 4)) \)
    (\( \text{let } ((z 5)) \)
      (+ x y))))

also evaluates to 12

\( (\text{let } ((x 2)) (+ x \ldots)) \) is just an abbreviation for
( (\text{LAMBDA } (x) (+ x \ldots)) 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

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

Len not tail recursive
```

```scheme
(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

\[
\text{(define (fact n)} \\
\text{ (cond} \\
\text{ ((zero? n) 1))} \\
\text{ ((eq? n 1) 1))} \\
\text{ (else (* n) }} \\
\text{ (fact (- n 1))))))
\]

\text{fact is original version}

\[
\text{(define (factor n acc)} \\
\text{ (cond ((zero? n) 1))} \\
\text{ ((eq? n 1) acc))} \\
\text{ (else (factor (- n 1)) (* n acc))))}
\]

\text{(define (factorial n)} \\
\text{ (factor n 1))}

\text{factor is tail recursive version}

Closures

- A \textit{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 \textit{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

\[
\text{let } ((x \ 5)) \\
(\text{let } ((f \ (\text{let } ((x \ 10)) \ (\text{lambda } (y) \ x)) \)) \\
(\text{list } x \ (f \ 1) \ x \ (f \ 1))) \quad \text{yields } (5 \ 10 \ 5 \ 10)
\]

Evaluation of Closures

\[
(\text{define } (gg \ z) \\
(\text{let* } ((x \ 2) \ (f \ (\text{lambda } (y) \ (+ \ x y)))) \ (\text{map } f \ z)))
\]

\(gg\) is actually a closure which is \((\text{lambda } (z) \ (\text{map } f \ z))\) where the defining environment is \{ \(x \rightarrow 2\); \(f \rightarrow (\text{lambda } (y) \ (+ \ x y))\}\)

we need this environment to evaluate \(gg\).

\(\text{> (square 2)}\)

4 \(\); is assumed to be evaluated in the context of the empty environment \{}\n
\(\text{> (gg '(1 2 3))}\)

1. value of \(gg\) is its closure
2. closure environment is expanded by argument association with parameter \(\{ \(x \rightarrow 2\); \(f \rightarrow (\text{lambda } (y) \ (+ \ x y)); \(z \rightarrow ' (1 \ 2 \ 3)\}\}\)
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 × R → R, curried+ : R → (R → R)
(\define (curried+ x) (\lambda (y) (+ x y)))
((curried+ 2) 3) yields 5
(let ((f (curried+ 1))) (f 10)) yields 11
Currying (and Closures)

\( \text{(define (mm x y) \((* x y)\))} \)
\( \text{mm} \); returns a closure

\( \text{(mm 2)} \); returns error because \text{mm} expects 2 arguments, not 1!

\( \text{(mm 2 3)} \); returns 6

\( \text{(define hh (lambda (x) (lambda (y) (* x y))) )} \)
\( \text{hh} \); returns \((\lambda (a1) \ldots)\)

\( \text{(hh 2 3)} \); not called in a curried manner, one argument at a

time, returns

\( \text{hh: expects only 1 argument, but found 2} \)

\( \text{((hh 2) 20)} \); proper way to call a curried \text{fcn} of 2 args
40

\( \text{(hh 2)} \); hh plus 1 argument returns a closure

\( (\lambda (a1) \ldots) \)

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

\( \text{(define (stream f n) (cons (f n) (list f)))} \); encodes

value of \text{f} applied to \text{n} as first element of the list

and \text{f} as the rest of list

\( \text{(define (head str) (car str))} \); head retrieves the next

value that is stored as the first element of list

\( \text{(define (tail str) (cons (apply (car (cdr str))}

\( (\text{list (car str)) (cdr str))} \);.tail constructs a new list with the next value as its

car and the generating function as it \text{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