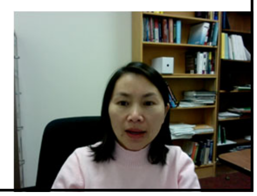
CSCI-C311 Programming Languages

Recursion

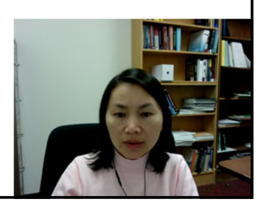
Dr. Hang Dinh



1

Outline and Reading

- After this lecture, you will learn
  - Recursion vs. Iteration
  - Tail recursion
- Reading
  - Scott 4e - Section 6.6
  - Scott 4e – Section 6.2.2
  - The Racket Guide – Section 10.3 Continuations



2

Recursion

- One of the major control-flow mechanisms in programming languages
- Requires no special syntax
- Only requires to permit functions to call themselves or to call other functions that then call them back in turn.
  - Fortran 77 and certain other languages do not permit recursion.
  - Fundamental to functional languages like Scheme.
- Equally powerful to (logically controlled) iteration
  - any iterative algorithm can be rewritten as a recursive algorithm,
  - and vice versa.

3

Iteration and Recursion

- In imperative languages: iteration is more natural
  - Because it's based on repeated modification of variables
- Example: It seems natural to use iteration to compute a sum $\sum_{i=1}^{10} f(i)$
  - In C one would say:

```
typedef int (*intFunc) (int);
int sum(intFunc f, int low, int high){
      int total = 0;
      for(int i=low; i<=high; i++)
            total += f(i);
      return total;
}
```

4

Iteration and Recursion

- In functional languages: recursion is more natural
  - Because it does not change variables.
- Example: It seems more natural to use recursion to compute a value defined by a recurrence:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b,b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b){
   /* assume a, b > 0 */
   if (a==b) return a;
   else if (a>b)
     return gcd(a-b, b);
   else return gcd(a, b-a);
}
```

5

Iteration and Recursion

- In both these cases, the choice could go the other way:

```
typedef int (*intFunc) (int);

int sum(intFunc f, int low, int high){
      //assume low <=high
     if (low==high)
          return f(low);
     return f(low)+sum(f, low+1, high);
}
```

```
int gcd(int a, int b){
   /* assume a, b > 0 */
   while (a!=b) {
     if (a > b)
         a = a-b;
     else
         b = b-a;
   }
   return a;
}
```

6

Iteration and Recursion

- It is sometimes argued that iteration is more efficient than recursion.
- It's more accurate to say that *naïve implementation* of iteration is usually more efficient than naïve implementation of recursion.
- Example:
  - the recursive implementations of the **sum** and the **gcd** will be less efficient if it makes real subroutine calls that allocate space on a run-time stack for local variables and bookkeeping information.
- An optimizing compiler can generate excellent code for recursive functions, especially one designed for a functional language.

7

Tail Recursion

- Tail-recursive function: No computation follows recursive call
  - The return value is simply whatever the recursive call returns.

```
int gcd(int a, int b){
   /* assume a, b > 0 */
   if (a==b) return a;
   else if (a>b)
     return gcd(a-b, b);
   else
     return gcd(a, b-a);
}
```

Tail-recursive function

```
typedef int (*intFunc) (int);

int sum(intFunc f, int low, int high){
       //assume low <=high
      if (low==high)
           return f(low);
      return f(low)+sum(f, low+1, high);
}
```

Not tail-recursive function

8

Why is Tail Recursion More Efficient?

- For tail-recursive functions, dynamically allocated stack unnecessary
  - Compiler can reuse the space belonging to the current iteration when it makes the recursive call.
- Iterative implementation of tail recursion:

```
int gcd(int a, int b){
   /* assume a, b > 0 */
   if (a==b) return a;
   else if (a>b)
     return gcd(a-b, b);
   else
     return gcd(a, b-a);
}
```

```
int gcd(int a, int b){
Start:
   if (a==b) return a;
   else if (a>b) {
     a =  a-b; goto Start;
   }
   else {b= b-a; goto Start;}
}
```


Transforming Naïve Recursion to Tail Recursion

- General transformation is based on *continuation-passing style* (CPS)
  - A recursive function can avoid doing work after returning from a recursive call by passing that work into the recursive call, in the form of a *continuation*.
- Continuations: generalizations of the notion of nonlocal **`goto`**s
- In low-level terms: a *continuation* consists of
  - a code address,
  - a referencing environment that should be established (or restored) when jumping to that address
  - A reference to another continuation that represents what to do in the event of a subsequent subroutine return.

Continuations

- In high-level terms, a *continuation* is an abstraction that captures a *context* in which execution might continue.
- Continuation support in Scheme/Racket takes the form of a function named **`call-with-current-continuation`** (or **`call/cc`**).
- The **`call/cc`** function
  - takes a single argument `f`, which is itself a function of one argument
  - calls `f`, passing as argument a continuation `c` that captures the current program counter, referencing environment, and stack backtrace.
  - At any point in the future, `f` can call `c`, passing a value `v`. The call will return `v` into `c`'s captured context, as if it had been returned by the original call to `call/cc`

11

Function **`call/cc`** in Racket

Argument function f

c is the captured continuation

```
> [call/cc (lambda (c) 0)]
0
> [call/cc (lambda (c) 4)]
4
> [call/cc (lambda (c) (c 7))]
7
> (+ 1 (+ 2 [call/cc (lambda (c) (c 7))]))
10
```

A continuation behaves like a one-argument function. This applies c to argument 7 and returns 7.

7

12

Function call/cc in Racket

```
> (+ 1 (+ 2 [call/cc (lambda (c) 4)]))
7
> (define saved-c #f)
> (+ 1 (+ 2 [call/cc (lambda (c)
                        (set! saved-c c)
                        4)]))
7
> (saved-c 0)
3
> (saved-c 4)
7
> (+ 3 (saved-c 4))
7
```

Continuation c encapsulates program context
`(+ 1 (+ 2  ?))`
so that it behaves like the function
`(lambda (v)`
`   (+ 1 (+ 2 v)))`

Store `c` to global variable `saved-c` so that we can use it else where later.

`saved-c` will abandon its own continuation when plugged in other expressions

13

Function call/cc in Racket

- The continuation captured by **call/cc** is determined at run time, not compile time.

```
(define saved-c #f)
(define (factorial n)
  (if (= n 1)
      [call/cc (lambda (c) (set! saved-c c) 1)]
      (* n (factorial (- n 1)))))
```

Definition of factorial:
$n! = 1 \times 2 \times 3 \times \cdots \times n$
$= n \times (n-1)!$

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (saved-c 2)
 application: not a procedure;
 expected a procedure that can be applied to arguments
  given: #f
> (factorial 3)
6
> (saved-c 2)
12
```

After this call, `saved-c` becomes
`(lambda (v)`
`      (*3 (*2 v)))`

14

Transforming to Tail Recursion using CPS

- We can **automatically** transform any recursive function to a tail-recursive function using continuation-passing style (CPS).
  - By converting every function in direct style to CPS style.
- A function written in CPS takes an extra argument: an explicit "continuation"; i.e., a function of one argument.
  - When the CPS function has computed its result value v, it "returns" it by calling the continuation function with this value v as the argument.
  - CPS versions of primitive functions **=**, **–**, and *****:

```
(define (=& x y c) (c (= x y)))
(define (-& x y c) (c (- x y)))
(define (*& x y c) (c (* x y)))
```

15

Transforming to Tail Recursion using CPS

```
(define (factorial n)
 (if (= n 0)
     1     ; NOT tail-recursive
     (* n (factorial (- n 1)))))
```

```
(define (=& x y c)
        (c (= x y)))
(define (-& x y c)
        (c (- x y)))
(define (*& x y c)
        (c (* x y)))
```

```
(define (factorial& n c)           ;c is a continuation
 (=& n 0 (lambda (b)
          (if b                    ; growing continuation
              (c 1)                ; in the recursive call
              (-& n 1 (lambda (nm1)
                       (factorial& nm1 (lambda (f)
                                        (*& n f c)))))))))
```

16

Transforming to Tail Recursion using CPS

```
(define (factorial& n c)            ;c is a continuation
 (=& n 0 (lambda (b)
         (if b                      ; growing continuation
             (c 1)                  ; in the recursive call
             (-& n 1 (lambda (nm1)
                      (factorial& nm1 (lambda (f)
                                       (*& n f c)))))))))
```

- To call a procedure written in CPS, need to provide a continuation that will receive the result computed by the CPS procedure.

```
> (factorial& 3 [lambda (x)(display x)])
6
```

17

Specific Transformations to Tail Recursion

- Not based on continuation passing; Use an "accumulating" parameter

```
(define (sum f low high) ;NOT tail recursive
  (if (= low high)
      (f low)                              ;then part
      (+ (f low) (sum f (+ low 1) high)))); else part
```

```
(define (sum f low high subtotal) ;tail recursive
  (if (= low high)
      (+ subtotal (f low))
      (sum f (+ low 1) high (+ subtotal (f low)))))
```

```
> (sum (lambda (x) (* 2 x)) 1 5 0)
30
```

18

Specific Transformations to Tail Recursion

- Hide subtotal parameter in an auxiliary, “helper” function

```
(define (sum f low high)
  (letrec ([sum-helper
            (lambda (low subtotal)
              (let ([new_subtotal (+ subtotal (f low))])
                   (if (= low high)
                       new_subtotal
                       (sum-helper (+ low 1) new_subtotal))))])
          (sum-helper low 0)))
```

```
> (sum (lambda (x) (* 2 x)) 1 5)
30
```

19

Think Recursively

- Example: Consider the calculation of Fibonacci numbers:

$$\mathrm{F}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

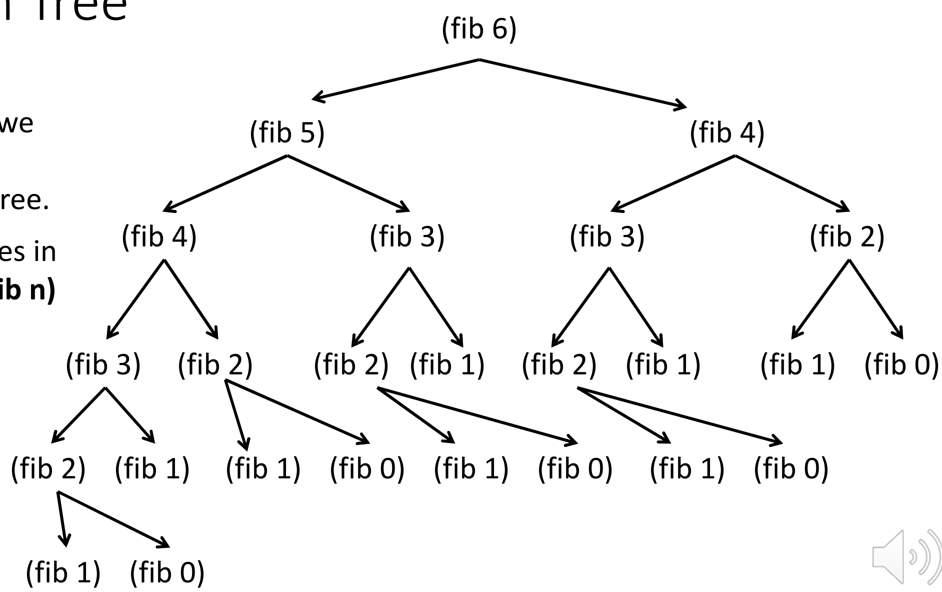
```
//linear time iteration in C
int fib(int n){
   int f1=0, f2=1;
   for(int i=2; i<=n; i++){
     int temp=f1+f2;
     f1=f2; f2=temp;
   }
   return f2;
}
```

```
;exponential time naïve recursion in Racket
(define (fib n)
  (if (< n 2)
      1
      [+ (fib (- n 1)) (fib (- n 2))]))
```

20

Recursion Tree

- For any call to a recursive function, we can organize the recursive calls in a tree.
- The number of nodes in recursive tree for **(fib n)** is exponential in **n**.



21

Thinking Recursively

- Cast the linear-time iterative algorithm for Fibonacci numbers in a tail-recursive form:

```
;linear time tail recursion in Racket
(define (fib n)
  (letrec ([fib-helper
            (lambda (f1 f2 i)
                     (if (= i n)
                         f2
                         (fib-helper f2 (+ f1 f2) (+ i 1))))])
          (fib-helper 0 1 0)))
```

  - This tail recursion has no side effect; the iteration version has side effects.

22