Iteration and Recursion

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 8 is due July 19
  - **Final exam, 1:00pm, Friday, August 2 in CHEB 170**
  - **Student Rating Instruction is open**

    Time will be provided in class next week, July 23, to complete them
- Readings: Read Chapter 6.5 - 6.6
- Lecture Contents
  - Motivation
  - Logical vs Enumeration Loops
  - Generators and Iterators
  - Recursion
  - Tail Recursion

How are the Student Ratings of Instruction (SRI) used?

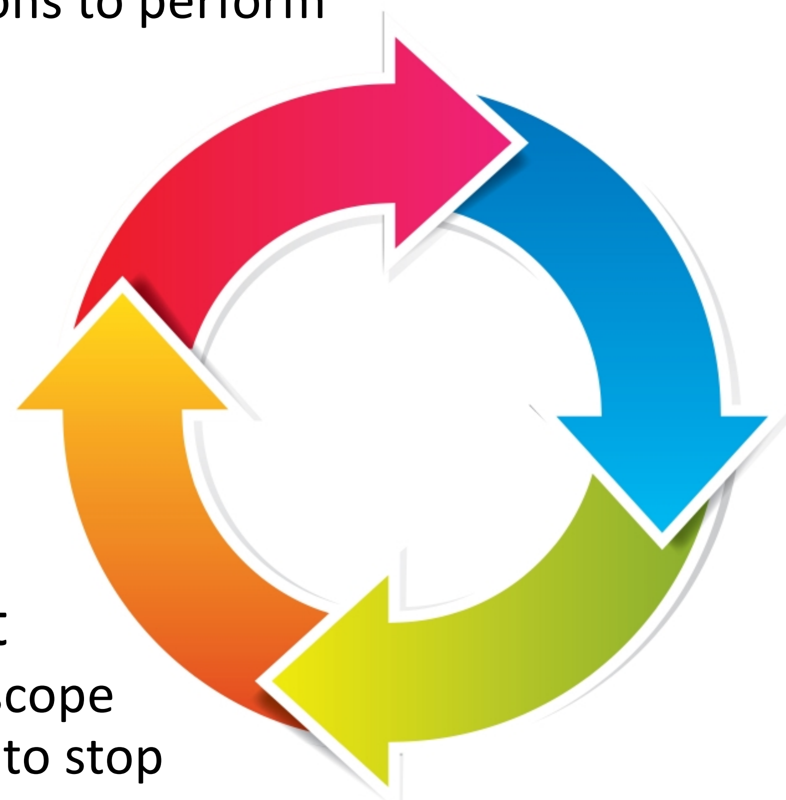
- ✓ Course and program ***(re) design.***
- ✓ ***Evaluation*** of teaching effectiveness.
- ✓ ***Promotion and tenure applications*** for instructors, and other personnel decisions.
- ✓ Preparation of supporting evidence for ***teaching awards and grants***.
- ✓ ***Quality assurance*** processes in the review and restructure of institutional, faculty, department and program goals.

How to complete the SRI

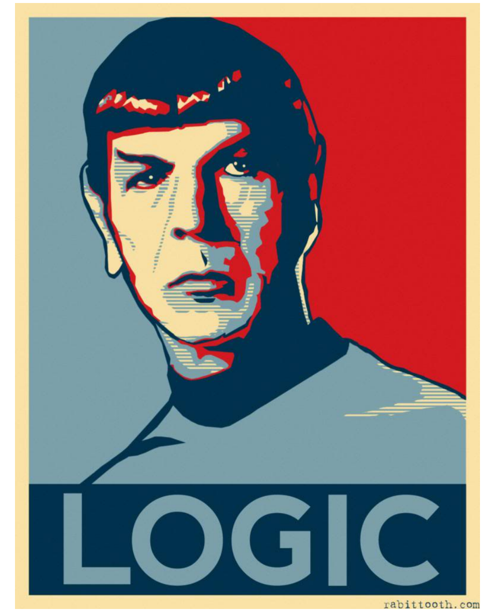
- Find the email in your Dal email account
  - Subject heading (depending on the system) is:
    - *Student Ratings of Instruction; or*
    - *Course Name and Number*
- Open the email and click on the link
  - Your course list should be visible
- Select the course for which you want to complete the evaluation
- Be sure to hit the SUBMIT button when you FINISH completing the form
- You may also SAVE and return to your work later

Motivation

- To be useful, programs need to
  - Repeat sequences of instructions
  - Decide at run-time how many iterations to perform
- E.g.,
  - Process arrays
  - Walk lists
  - Read arbitrary sized input
  - etc
- There are two approaches
  - Iteration
  - Recursion
- Iteration uses a loop construct that
  - Performs all repetitions in the same scope
  - Uses side-effects to determine when to stop



Iteration (Looping)



- Two types of loops
  - Logically controlled loops
    - Example: while-loop
    - Executed until a Boolean condition changes
    - The number of iterations is not known in advance
  - Enumeration-controlled loops
    - Example: for-loop
    - One iteration per element in a finite set
    - The number of iterations is known in advance

- Some languages do not have loop constructs

  E.g., Scheme, which uses tail recursion instead

Logically Controlled Loops

- Pre-loop test

```
while ( condition ) do
   ...
end
```

  - The loop may be executed 0 or more times
  - Test occurs before the iteration
- Post-loop test :

```
do
   ...
while ( condition )
```

  - Loop is executed at least once
  - Test is performed at end of iteration
- Mid-loop test or one-and-a-half loop

```
loop
   ...
   if ( condition ) break
   ...
   if ( condition ) break
   ...
end
```

  - Conditions are tested inside the loop and may break out
- Modern languages provide a `break` statement to use first two loop constructs in this way

Do-While Loop Implementation

DO

statements

WHILE cond

```
L1:
  statements
  r1 := cond
  if r1 goto L1
...
```

While Loop Implementation

WHILE cond do

  statements

END

```
    goto L2
L1:
    statements
L2: r1 := cond
        if r1 goto L1
...
```

For Loop Implementation

for(**init**; **cond**; **step**) {
  statements
}

// A “for” loop is a
// “while” loop with
// extra stuff

```
   [init]
   goto L2
L1:
   statements
   [step]
L2: r1 := [cond]
    if r1 goto L1
...
```

Enumeration Controlled Loops

- Use an index variable to count up or down the number of iterations
- The index variable can be incremented / decremented by a step other than 1

```
FOR i = start TO end BY step DO:
      ...
END
```

  - This loop
    - Initializes *i* to *start*
    - Adds *step* to i at the end of each iteration
    - Tests if *i* is less than *end*
    - If so, performs another iterations

For (Enumeration) Implementation

FOR i = start TO end BY step DO

  statements

END

```
    r1 := start
    r2 := end
    r3 := step
L1: if r1 > r2 goto L2
    statements
    r1 := r1 + r3
    goto L1
L2: ...
```

For (Enumeration) Implementation If the index is not used in the loop

```
FOR i = start TO end BY step DO
  statements
END
```

```
    r1 := end - start
    r1 := r1 / step
    inc r1
L1: if r1 == 0 goto L2
    statements
    dec r1
    goto L1
L2: ...
```

Trade-Offs

- Logically controlled loops are very flexible but expensive.
  - May have arbitrarily expensive condition
  - Cannot be unrolled
  - May have arbitrarily expensive step
- Enumeration-Controlled loops
  - Have a single int or float comparison
  - Can be unrolled to improve pipelining
  - Have a simple step,
    e.g., increment / decrement
- for-loop in C/C++/Java is syntactic sugar for init-test-step idiom of logic-controlled while loops.

Iterators and Generators

- Observation: Often, loops are used to iterate over sequences of elements that are
  - Stored in a data structure
  - Generated by a procedure
  - Read from a file (or input)
- ***Iterators*** are a clean way for iterating over a stored sequence
- ***Generator*** are a clean way for generating a sequence as needed

Generators

- Idea:
  - A generator is a self contained function that stores its local state

    Sound familiar?
  - Instead of *returning* a value, it *yields* a value
  - Next time it is called, it continues from the yield statement
- Generators provide an easy way for the programmer to generate a sequence of values
  - In languages that do not have generators, a loop would need to be used to generate all values at once
  - Generators generate the next value and “yield” it when they are called
  - The values are generated only when they are need it

Generators in Python

- In Python, a generator function returns a generator object
- The generator object has a `next()` method, which executes the code in the generator function "yields" the next value
- Example

```
def make_counter():
  for i in range(0, 1000000):
    yield i

cnta = make_counter()
print cnta.next()
print cnta.next()
print cnta.next()
…
```

  - *Prints out, 0 1 2 3 …*

C Implementation of a Counter

```
int counter() {
  static int i = -1;
  i = (i + 1) % 1000000;
  return i
}

// Idea: state of generator is
// stored in a static variable
// Can only be used once
```

Another Generator in Python

```
def make_iterator(lst):
  for e in lst:
    yield e

for item in make_iterator( lst ):
  print item
```

- Calling this function on a list will return an iterator that iterates over the list

Iterators

- An iterator is an object that has a `next()` method, which returns the next item in a sequence
- All generators are iterators, but not all iterators are generators
- Generators are typically a single function that stores all the necessary state to generate the next item
- **Iterators do not necessarily keep local state** (most usually do)
- Iterators are typically applied to collections, such as lists, array, sets, maps, etc
- The iterators are used to sequentially access all items in a collection

Iterator Interface

- Idea: Many languages define standard interfaces for iterating over a data structure
  - `begin()` : returns the first element of the iteration
  - `next()` : returns the next element of the iteration
  - `end()` : returns the last element of the iteration
- Example: C++

```
for( cont::iterator i = cont.begin(); // get iter
                    i != cont.end();  // done?
                    i++ ) {           // goto next
  cout << *i;                         // Use i
}
```

- C++ overloads two operators on the iterators
  - ++ : which is the same as calling `next()`
  - * : dereference used to access the value at current locations

Iterator Interface (in Java)

- Assume that variable *coll* refers to a *Collection* of *MyObjs*
- Before Java 5, iterators were not built-in

```
Enumeration e = coll.elements();      // get iter
while (e.hasMoreElements()) {         // done?
  MyObj o = (MyObj) e.nextElement();  // Use o
  // Use o
}
```

- Most modern languages have iterators built-in
- In Java 5 and later:

```
for (MyObj o : coll) {
  // Use o
}
```

Iterator Interface (in Python)

- In Python, all loops use iterators
- To implement a simple counting loop you need to create an iterator

```
for i in range(0,10):
  print i
```

  - The range() function returns an integer iterator from 0 to 9
  - All collections in Python have iterators
  - It’s actually hard to use a collection in Python without iterators
- Even input and output is done using iterators:
  - Example: the first line of a scheme parser creates a token iterator

```
tokens = iter(sys.stdin.read().split())
```

Iteration in Functional Languages

- Use closures to create generators and iterators
- Use **recursion** to iterate over any collection or sequence

Recursion: It Throws Us for a Loop

- Every iterative procedure can *easily* be turned into a recursive one:

Iterative	Recursive

`while( condition ) {`
`  S1;`
`  S2;`
`  ...`
`}`
	`procedure P() {`
	`  if( condition ) {`
	`    S1;`
	`    S2;`
	`    ...;`
	`    P();`
	`  }`
	`}`

- The converse is not true

  e.g., quicksort, merge sort, fast matrix multiplication, etc.

- Q: Why don't functional languages support iteration?
- A: Iteration relies on updating the iterator variable (side effect)

Tail Recursion

- Naive recursion is less efficient than tail iteration
- An optimizing compiler often converts recursion into iteration when tail recursion occurs
- ***Tail recursion*** occurs when there is no work done after the recursive call
- This is a standard approach for implementing iteration in functional languages

Aside: `car` and `cdr` in Scheme

- Scheme has two important list functions
- The car function takes a list as a parameter and returns the head of the list:
  - E.g., (**car** ‘(1 2 3 4)) yields **1**
- The cdr function takes a list as a parameter and returns, removes the head and returns the rest of the list:
  - E.g., (**cdr** ‘(1 2 3 4)) yields **(2 3 4)**

Example 1:

- What does this do?

```
(define sum
  (lambda (int_list)
    (if (null? int_list)
      0
      (+ (car int_list )
         (sum (cdr int_list))))))
```

Is list empty?

Yes, return 0

Recursive case

Addition after the recursive call

- Is this tail recursive?
  - Why not?
- Non-tail-recursive implementations can be made tail recursive
- Idea: Work to be done after the recursive call is passed to the recursive call.
  - A Helper function is typically used

Example 1: Tail Recursive Version

- Here is a tail recursive version

```
(define sum_helper
  (lambda (total int_list)
    (if (null? int_list)
      total
      (sum_helper
        (+ total (car int_list))
        (cdr int_list)))))
```

```
(define sum
  (lambda (int_list)
    (sum_helper 0 int_list)))
```

Pass the running total and the rest of the list

If list is empty, return total

Recursive case

Increment total first

Remainder of the list

Call sum_helper, initial total is 0

Example 2: Sum of *f(i)*

- This is not a tail recursive version, why?

```
(define sum_f
  (lambda (f low high)
    (if (= low high)
      (f low)
      (+ (f low)
         (sum f (+ low 1) high)
      )
    )
  )
)
```

- Can we do better?

Aside: `letrec` in Scheme

The **`letrec`** function in Scheme is like **`let`** except that the binding is visible immediately instead of only in the body.

Example with `let`

```
(let ((a 4))
  (let (a (f a))
    a
  )
)
```

Example with `letrec`

```
(let ((a 4))
  (letrec (a (f a))
    a
  )
)
```

Example 2: Tail Recursive Version

- What is happening here?

```
(define sum_f (lambda (f low high)
  (letrec ((sum (lambda (i total)
    (if (> i high)
      total
      (sum (+ i 1) (+ total (f i)))
    ) )
    (sum low 0)
  )
)
```

Need letrec ☺

Increment total

Increment i

Call helper

- Where is our sum_helper?

Example 3: Fibonacci

- What's happening here?

```
(define fib (lambda (n)
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2))
    )
  ) )
)
```

- Why is this not tail recursive?
- Can we do better?

Example 3: Tail Recursive

- What's happening here?

```
(define fib (lambda (n)
  (letrec ((fib (lambda (f1 f2 i)
    (if (= i n)
      f2
      (fib f2 (+ f1 f2) (+ i 1))
    ))))
    (fib 0 1 0)
  )
)
```

- What's going on here?

Is Tail Recursion Always Possible?

- It depends…
- Answer 1:
  - Every recursive algorithm can be implemented iteratively by using a stack
  - Every iterative algorithm can be implemented using tail recursion
  - So, technically, every recursive algorithm has a tail-recursive variant.
- Answer 2:
  - In practice, multi-way recursive algorithms cannot be implemented in a tail-recursive manner that is also intuitive.