

ECS510

Algorithms and Data Structures in an
Object Oriented Framework

“ADSOOF”

Lisp Lists and Recursion

Recursion with Strings

How do we know if `str1` starts with `str2`?

- If first character of `str1` not equal to first character of `str2`, then no
- Otherwise depends on whether rest of `str1` starts with rest of `str2`
- If `str2` is empty, then yes
- If `str1` is empty and `str2` is not, then no

This is thinking about a problem recursively

Recursive method

```
public static boolean startsWith(String str1,String str2)
// Returns true if str1 starts with str2, false otherwise
{
    if(str2.length()==0)
        return true;
    else if(str1.length()==0)
        return false;
    else if(str1.charAt(0)!=str2.charAt(0))
        return false;
    else
        return startsWith(str1.substring(1),str2.substring(1));
}
```

Lisp Lists

- Not provided as standard part of Java
- Introduced in this module because useful for recursion examples
- Also introduced as an example of an immutable data structure
- A generic type `LispList<E>`
- A Lisp list of type `LispList<T>` is either empty or consists of two parts, its “head” (the first element, of type `T`), and its “tail” (the rest, of type `LispList<T>`)
- Similar to `str.charAt(0)` and `str.substring(1)`

Lisp List methods

For `LispList<E>`

- `E head()`
- `LispList<E> tail()`
- `LispList<E> cons(E item)`
- `boolean isEmpty()`
- `static LispList<E> empty()`

There are no other methods, and no constructors
(`tail`, `cons` and `empty` are factory methods)

Abstract Data Type

- Lisp lists are also introduced to illustrate the general idea of an abstract data type (ADT)
- An ADT is defined only by its methods in terms of their specified interaction with other code
- `LispList<E>` will be used to illustrate the general principle of an ADT implemented by an internal concrete data structure
- So do not confuse `LispList<E>` with the data structure used to implement it (linked list, but for illustration we will also see an array implementation)

Lisp List properties

- List of elements, each has a position
- But not indexable (no `ls.get(i)`)
- Immutable - no method changes the list it is called on, `cons`, `tail` both return new lists, so

`ls=ls.cons(w);` works

`ls.cons(w);` achieves nothing

Input/Output

- For practical purposes add method
`String toString()`
- So `ls.toString()` returns e.g. `"[5 , 8 , 2]"`
- `toString` is used automatically e.g in
`System.out.print(ls);`
- We also have
`static LispList<Integer>`
`parseIntLispList(String str)`
to do the reverse e.g
`ls=parseIntLispList(str);`

Recursive methods with Lisp lists

- Consider what you want
- Consider what happens when you apply the same operation to the tail
- See how you can put together the result of the recursive method on the tail with what you have to get what you want
- Example: sum of all integers in a Lisp List of integers is the head plus the sum of all integers in the tail (sum of empty list is 0)

- What is the sum of [5,3,8,4,2,7]?

- What is the sum of [5,3,8,4,2,7]? 29

- What is the sum of [5,3,8,4,2,7]? 29
- How can we work this out?

- What is the sum of $[5, \underline{3}, 8, 4, 2, 7]$? 29
- What is the sum of $[3, 8, 4, 2, 7]$?

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29=5+24$

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29 = 5 + 24$
- Is this general (doesn't matter what numbers)?

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29=5+24$
- Is this general (doesn't matter what numbers)? Yes

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29 = 5 + 24$
- Is this general (doesn't matter what numbers)? Yes
- So the sum of all the numbers in a list is its head plus the sum of all the numbers in its tail for any list of numbers

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29 = 5 + 24$
- Is this general (doesn't matter what numbers)? Yes
- So the sum of all the numbers in a list is its head plus the sum of all the numbers in its tail for any list of numbers
- So we have a complete algorithm

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29 = 5 + 24$
- Is this general (doesn't matter what numbers)? Yes
- So the sum of all the numbers in a list is its head plus the sum of all the numbers in its tail for any list of numbers
- So we have a complete algorithm
- Almost - we need to consider the case of a list which doesn't have a head and tail

- What is the sum of [5,3,8,4,2,7]? 29
- What is the sum of [3,8,4,2,7]? 24
- $29 = 5 + 24$
- Is this general (doesn't matter what numbers)? Yes
- So the sum of all the numbers in a list is its head plus the sum of all the numbers in its tail for any list of numbers
- So we have a complete algorithm
- Almost - we need to consider the case of a list which doesn't have a head and tail
- Sum of all the numbers in [] is 0

Sum of Lisp List of integers using recursion

```
static int sum(LispList<Integer> ls)
{
    if(ls.isEmpty())
        return 0;
    else
        return ls.head()+sum(ls.tail());
}
```

Base Case

Empty list will be one case, there may be others when we have a solution in one step

```
static <T> boolean isIn(LispList<T> ls, T item)
{
    if(ls.isEmpty())
        return false;
    else if(ls.head().equals(item))
        return true;
    else
        return isIn(ls.tail(), item);
}
```

Building Lisp Lists recursively

```
public static <T> LispList<T> change(LispList<T> ls, T p, T q)
{
    if(ls.isEmpty())
        return LispList.<T>empty();
    else
    {
        LispList<T> t = change(ls.tail(), p, q);
        T h = ls.head();
        if(h.equals(p))
            return t.cons(q);
        else
            return t.cons(h);
    }
}
```


- To change all 6s to 7s in [3,6,5,7,6,4,3]

- To change all 6s to 7s in [3,6,5,7,6,4,3]
- Answer is [3,7,5,7,7,4,3]

- To change all 6s to 7s in $[3, \underline{6, 5, 7, 6, 4}, 3]$
- Answer is $[3, \underline{7, 5, 7, 7, 4}, 3]$
- Change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$

- To change all 6s to 7s in $[3, \underline{6, 5, 7, 6, 4}, 3]$
- Answer is $[3, \underline{7, 5, 7, 7, 4}, 3]$
- Change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$
- Answer is to change all 6s to 7s in tail, then cons head to front?

- To change all 6s to 7s in $[3, \underline{6, 5, 7, 6, 4, 3}]$
- Answer is $[3, \underline{7, 5, 7, 7, 4, 3}]$
- Change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$
- Answer is to change all ps to qs in tail, then cons head to front?

- To change all 6s to 7s in $[3, \underline{6, 5, 7, 6, 4}, 3]$
- Answer is $[3, \underline{7, 5, 7, 7, 4}, 3]$
- Change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$
- Answer is to change all ps to qs in tail, then cons head to front?
- Not general for all numbers, change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$ not $[6, 5, 7, 7, 4, 3]$

- To change all 6s to 7s in $[3, \underline{6, 5, 7, 6, 4}, 3]$
- Answer is $[3, \underline{7, 5, 7, 7, 4}, 3]$
- Change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$
- Answer is to change all ps to qs in tail, then cons head to front?
- Not general for all numbers, change all 6s to 7s in $[6, 5, 7, 6, 4, 3]$ is $[7, 5, 7, 7, 4, 3]$ not $[6, 5, 7, 7, 4, 3]$
- Special case - if head is p, cons q to front of result of recursive call

- To change all 6s to 7s in [3,6,5,7,6,4,3]
- Answer is [3,7,5,7,7,4,3]
- Change all 6s to 7s in [6,5,7,6,4,3] is [7,5,7,7,4,3]
- Answer is to change all **p**s to **q**s in tail, then cons head to front?
- Not general for all numbers, change all 6s to 7s in [6,5,7,6,4,3] is [7,5,7,7,4,3] not [6,5,7,7,4,3]
- Special case - if head is **p**, cons **q** to front of result of recursive call
- Plus base case - change all **p**s to **q**s in [] is []

Sum of Lisp List of integers using iteration

```
static int sum(LispList<Integer> ls)
{
    int sumSoFar=0;
    for(; !ls.isEmpty(); ls=ls.tail())
    {
        int n = ls.head();
        sumSoFar = sumSoFar+n;
    }
    return sumSoFar;
}
```

Stack and reverse

```
public static <T> LispList<T> change(LispList<T> ls1, T p, T q)
{
    LispList<T> ls2 = LispList.empty();
    for(; !ls1.isEmpty(); ls1=ls1.tail())
    {
        T h = ls1.head();
        if(h.equals(p))
            ls2 = ls2.cons(q);
        else
            ls2 = ls2.cons(h);
    }
    LispList<T> ls3 = LispList.empty();
    for(; !ls2.isEmpty(); ls2=ls2.tail())
        ls3 = ls3.cons(ls2.head());
    return ls3;
}
```

A Sort Algorithm

- To sort a Lisp list
 - If list is empty, return empty
 - Otherwise, sort its tail
 - and insert its head into correct position in sorted tail
- To insert item into a sorted Lisp list
 - If list is empty, return list of one item
 - If item is less than head, cons to front
 - Otherwise, insert item into tail, and cons head to front
- Remember, all operations are constructive

Insertion Sort and Quicksort

- The previous algorithm is insertion sort
- You may have covered insertion sort before, but you may not have seen the link between how it was described previously and how it was described just now
- The reason is that insertion sort is often introduced in introductory programming in a form that involves swapping the position of elements in an array
- The underlying algorithm is as just described, swapping elements in an array is one way to implement it, so long as it is with an array and done destructively
- Similar applies to quicksort

Recursion and Algorithm Description

- One of the reason for considering recursion is that description of algorithms in recursive terms tends to closely match the code for the algorithms implemented recursively
- It is often easier to reason logically that an algorithm will work correctly when it is described recursively rather than iteratively
- Separating out the core aspects from secondary implementation details should make it easier to understand an algorithm from its description
- The above assumes that you have reached the point where you can think recursively: some seem to find that difficult

Recursion and Algorithm Development

- Once you are used to thinking recursively, it can become easier to develop new algorithms and write code by starting off thinking of solutions recursively
- You can think of the solution in terms of the algorithm, and then add the code details to implement it, rather than thinking of it purely in terms of code details
- If the recursive algorithm you develop is tail recursive, it can be modified to give iterative code
- In some programming languages, the compiler will automatically convert tail recursive code to iterative code at the machine or virtual machine level
- Java does not yet do this, although it is being considered

Insertion sort code (recursive)

```
public static LispList<Integer>
    sort(LispList<Integer> ls)
{
    if(ls.isEmpty())
        return LispList.<Integer>empty();
    else
        return insert(ls.head(), sort(ls.tail()));
}
```

Insert code (recursive)

```
private static LispList<Integer>
insert(Integer n, LispList<Integer> ls1)
{
    if(ls1.isEmpty())
        return LispList.empty().cons(n);
    Integer h = ls1.head();
    if(h.compareTo(n)>0)
        return ls1.cons(n);
    else
    {
        LispList<Integer> ls2 = insert(n,ls1.tail());
        return ls2.cons(h);
    }
}
```


Quicksort algorithm

- Split tail of list into two:
 - All items less than head
 - All items greater than or equal to head
- Sort the two lists
- Join sorted lists together, with old head in middle
- Base case: sort of empty list is empty list

```
public static LispList<Integer> sort(LispList<Integer> ls)
{
    if(ls.isEmpty() || ls.tail().isEmpty()) return ls;
    Integer pivot = ls.head();
    ls = ls.tail();
    LispList<Integer> smaller = LispList.empty();
    LispList<Integer> greater = LispList.empty();
    for(; !ls.isEmpty(); ls=ls.tail())
    {
        Integer h = ls.head();
        if(h.compareTo(pivot)<0)
            smaller = smaller.cons(h);
        else
            greater = greater.cons(h);
    }
    smaller = sort(smaller);
    greater = sort(greater);
    greater = greater.cons(pivot);
    return append(smaller,greater);
}
```

Merge sort algorithm

- Split list into two lists of equal size, doesn't matter how
- Sort the two lists
- Merge sorted lists together

```
public static LispList<Integer> sort(LispList<Integer> ls)
{
    if(ls.isEmpty() || ls.tail().isEmpty())
        return ls;
    LispList<Integer> ls1 = LispList.empty();
    LispList<Integer> ls2 = LispList.empty();
    for(; !ls.isEmpty(); ls=ls.tail())
    {
        ls1 = ls1.cons(ls.head());
        ls = ls.tail();
        if(!ls.isEmpty())
            ls2 = ls2.cons(ls.head());
        else
            break;
    }
    ls1 = sort(ls1);
    ls2 = sort(ls2);
    return merge(ls1, ls2);
}
```

```
public static LispList<Integer>
merge(LispList<Integer> ls1,LispList<Integer> ls2)
{
    if(ls1.isEmpty()) return ls2;
    else if(ls2.isEmpty()) return ls1;
    else
    {
        Integer h1 = ls1.head();
        Integer h2 = ls2.head();
        if(h1.compareTo(h2)<0)
        {
            LispList<Integer> ls3 = merge(ls1.tail(),ls2);
            return ls3.cons(h1);
        }
        else
        {
            LispList<Integer> ls3 = merge(ls1,ls2.tail());
            return ls3.cons(h2);
        }
    }
}
```

How method calls work

- Remember, a method call executes in its own environment
- The variables in its environment are not linked to variables of the same name in other environments
- But the variables in a method call environment may alias objects from the environment where the method call was made
- When a method call finishes, computation returns to the environment where the method call was made

Recursion example

```
public static <T> LispList<T>
append(LispList<T> ls1, LispList<T> ls2)
{
    if(ls1.isEmpty()) return ls2;
    T h = ls1.head();
    LispList<T> ls3 = append(ls1.tail(), ls2);
    return ls3.cons(h);
}
```

- Consider a call to `append` where `ls1` is `[1, 2, 3]` and `ls2` is `[4, 5, 6]`
- In this environment, `h` will be set to 1
- The recursive call will have its own environment with its own `ls1` set to `[2, 3]`, its own `ls2` also set to `[4, 5, 6]` and its own `h` set to 2.
- When the recursive call finished, it returns `[2, 3, 4, 5, 6]`, and computation goes back to the old environment where `h` is 1
- So `[1, 2, 3, 4, 5, 6]` is returned

ls1=[1,2,3] ls2=[4,5,6] h=1

ls1=[1, 2, 3] ls2=[4, 5, 6] h=1 *calls*

ls1=[2, 3] ls2=[4, 5, 6] h=2

ls1=[1, 2, 3] ls2=[4, 5, 6] h=1 *calls*

ls1=[2, 3] ls2=[4, 5, 6] h=2 *calls*

ls1=[3] ls2=[4, 5, 6] h=3

ls1=[1, 2, 3] ls2=[4, 5, 6] h=1 *calls*

ls1=[2, 3] ls2=[4, 5, 6] h=2 *calls*

ls1=[3] ls2=[4, 5, 6] h=3 *calls*

ls1=[] ls2=[4, 5, 6]

ls1=[1, 2, 3] ls2=[4, 5, 6] h=1 *calls*

ls1=[2, 3] ls2=[4, 5, 6] h=2 *calls*

ls1=[3] ls2=[4, 5, 6] h=3 *calls*

returns [4, 5, 6]

ls1=[1, 2, 3] ls2=[4, 5, 6] h=1 *calls*

ls1=[2, 3] ls2=[4, 5, 6] h=2 *calls*

ls1=[3] ls2=[4, 5, 6] h=3 ls3=[4, 5, 6]

ls1=[1,2,3] ls2=[4,5,6] h=1 *calls*

ls1=[2,3] ls2=[4,5,6] h=2 *calls*

returns [3,4,5,6]

ls1=[1,2,3] ls2=[4,5,6] h=1 *calls*

ls1=[2,3] ls2=[4,5,6] h=2 ls3=[3,4,5,6]

ls1=[1,2,3] ls2=[4,5,6] h=1 *calls*
returns [2,3,4,5,6]

ls1=[1,2,3] ls2=[4,5,6] h=1 ls3=[2,3,4,5,6]

returns [1, 2, 3, 4, 5, 6]

Thinking recursively (induction)

- Just think of the recursive call as a separate program which goes off, calculates what is wanted, and returns it
- Check the base case works
- Show that if the recursive call works, the whole method works

So ...

- If the base case works, the call whose recursive call is the base case works
- So the call whose recursive call is the call whose recursive call is the base case works
- So the call whose recursive call is the call whose recursive call is the call whose recursive call is the base case works
- And so on for as long as you like

Tail Recursion

- Tail recursion is when the result of a recursive call is returned directly as the return value of the call that the recursive call is in
- In all the examples we have seen, something is done with the result of the recursive call to give the value that is returned, so they are not tail recursive
- Tail recursion is almost identical to iteration, because recursive calls are repeated, but when the base case is reached, the calculation is done
- With recursion that is not tail recursion, work that is to be done after the recursive call is in effect stacked up in the environments that are returned to
- With tail recursion there is no return to access the environment of previous recursive calls

Tail Recursion Example

```
static int sum(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty())  
        return acc;  
    else  
        return sum(ls.tail(), acc+ls.head());  
}
```

Tail Recursion Example

```
static int sum(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty())  
        return acc;  
    else  
        return sum(ls.tail(), acc+ls.head());  
}
```

- Note this is NOT an acceptable solution

Tail Recursion Example

```
static int sum(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty())  
        return acc;  
    else  
        return sum(ls.tail(), acc+ls.head());  
}
```

- Note also that writing this as:

```
static int sum(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty()) return acc;  
    else {  
        acc=acc+ls.head();  
        ls=ls.tail();  
        return sum(ls, acc);  
    }  
}
```

suggests a way of thinking that is purely iterative

Java Method Calls (recap)

- Java method calls pass values not variables
- So every call to a method has its own set of variables, initially those named by the method parameters
- The parameter variables in a method call are assigned the values passed as the method arguments
- Java variables of an object type hold references to objects
- Assignment of reference to an object results in an alias to that object
- Changing the value of a variable inside a method call does not cause the value of a variable of the same name in another method call to get changed
- Changing the value of a variable inside an object does change it for aliases of that object

Mutability and Immutability (recap)

- An immutable object is one that cannot have its state changed by calling a method on it
- Java `Strings` are immutable, but Java `ArrayLists` are mutable
- `LispList<E>` is an example of an immutable collection type
- For example, `arr.add(val)` changes the actual object to which `arr` refers
- But `ls.cons(val)` does not change the actual object to which `ls` refers, instead it returns a new object representing the change
- So `ls.cons(val)` only makes sense if a variable is assigned to refer to what it returns, the same applies to `ls.tail()`
- If `ls.cons(val)` or `ls.tail()` is used as an argument to a method call, it is actually an assignment to a parameter variable of the method call

Tail Recursion Example

```
private static int sum1(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty())  
        return acc;  
    else  
        return sum(ls.tail(), acc+ls.head());  
}
```

```
static int sum(LispList<Integer> ls) {  
    return sum1(ls, 0);  
}
```

- If you are asked to write a method which takes some arguments and returns a value, it is not acceptable to supply code that takes an extra argument which the calling code has to set to a particular value
- You can get round this by putting the code into a helper method, as is done here

Tail Recursion and Iteration

```
private static int sum1(LispList<Integer> ls, int acc) {  
    if(ls.isEmpty())  
        return acc;  
    else  
        return sum(ls.tail(), acc+ls.head());  
}
```

- Almost the same as:

```
static int sum2(LispList<Integer> ls) {  
    int acc=0;  
    while(!ls.isEmpty()) {  
        acc=acc+ls.head();  
        ls=ls.tail();  
    }  
    return acc;  
}
```

- But the recursion means separate variables for each iteration, rather than reassignment of one set of variables

Lisp Lists

- Introduced as particularly suitable for demonstrating and practicing recursion
- You can use them with iteration as well
- Immutable, so all operations must be constructive
- Quicksort and merge sort algorithms easy to show on them, no additional complication of array indexes

Recursion

- If the solution to a problem can be given as putting together solutions to smaller versions of the same problem, it is a complete solution
- So long as all base cases are covered
- Should be intuitively obvious after practice, but the untrained human mind seems happier with iteration
- So train yourself to think recursively, and practice
 - Lisp list programming will help
- Don't try to convert it mentally to iteration

Recursion and Iteration

- Recursion can be a good way of thinking about a solution to a problem, experienced programmers often find recursive code easier to read and write
- Recursive code which is tail recursive can be easily converted to iteration
- However, if you are asked to solve a problem using recursion and come up with a tail recursive solution, it may mean you have not really thought about it in recursive terms
- Recursive code which is not tail recursive can be turned into iterative code through the use of an explicit stack to represent the environments returned to
- The stack and reverse technique in effect does this