Slides by Andrew DeOrio
and James Juett

EECS 280
Programming and Introductory Data Structures

Recursion and Iteration

Review: recursion

- A recursive problem is one that is defined in terms of itself.
- A recursive problem has two important features:
  - A base case(s) that stops the recursion
  - A recursive step that solves a smaller version of the same problem
- Unfortunately, general recursion requires many stack frames, which can use a lot of memory

```
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```

Tail call review

- A function call is a **tail call** if it is the very last thing in its containing function

```
int factorial(int n){
  //...
  return factorial(n-1);
}
```

- The calling function has **no pending work** to do after a tail call (in the passive flow)
- Avoids saving the stack frame!
- Tail call optimization can take a recursive algorithm from linear to constant space complexity as long as it only makes tail calls

Review: Tail Recursion

- The many stacks frames problem was solved with tail recursion
- Tail recursion **reuses the stack frame**

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

  return fact_helper(num, 1);
}
```

Tail Recursion and Iteration

- Both of these implementation require only 1 stack frame

```
int factorial(int n){
  int result = 1;
  while (n != 0) {
    result *= n;
    n -= 1 ;
  }
  return result;
}
```

```
int factorial(int n, int result) {
  if (n == 0) return result;
  return factorial(n-1, result*n);
}
```

Stack frame example

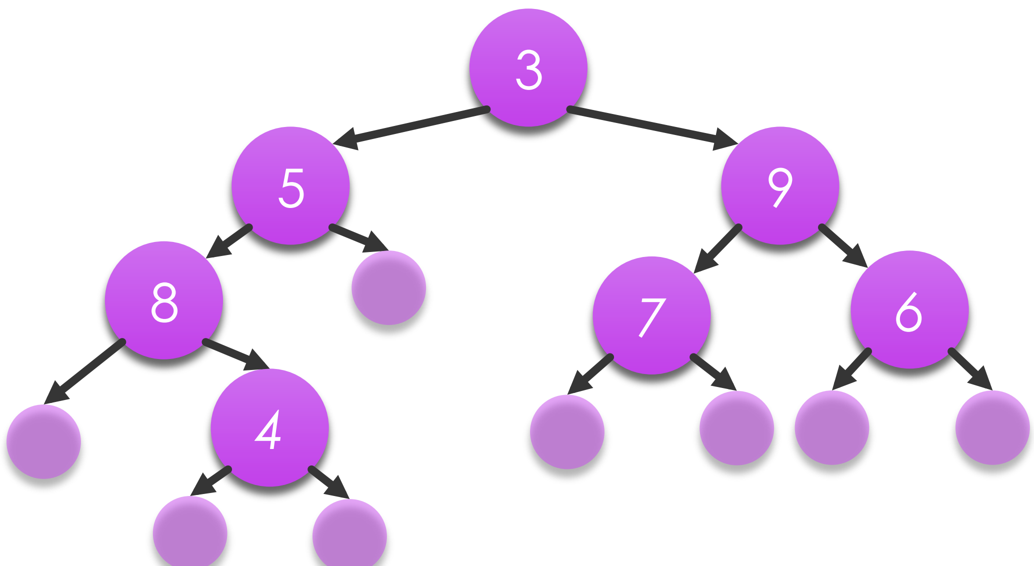
- Stack frames example

```
int factorial_helper(int n, int result){
  if (n == 0) return result;
  return factorial_helper(n-1, n * result);
}

int factorial(int n) {
  return factorial_helper(n ,1);
}

int main() {
  factorial(3);
  return 0;
}
```

Recursive data structures

- List

- Tree



List definition

- List

- A list is:
  - The empty list, or
  - An integer followed by a list

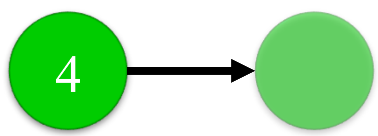
List functions provided in project 2

```
list_t list_make();
// returns an empty list.

list_t list_make(int elt, list_t list);
// given the list (list) make a new list consisting
// of the new element followed by the elements of
// the original list.

void list_print(list_t list);
    // MODIFIES: cout
    // EFFECTS: prints list to cout.
```

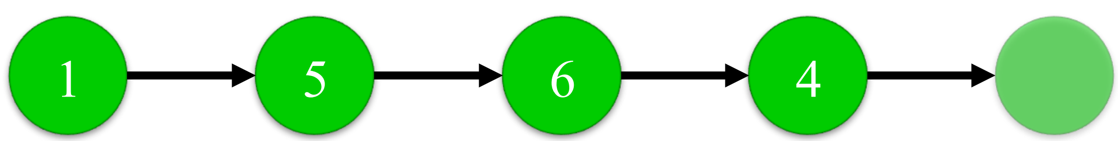
list_make()

- `list_t empty = list_make();`

```
list_t list1 = list_make(4, empty);
```



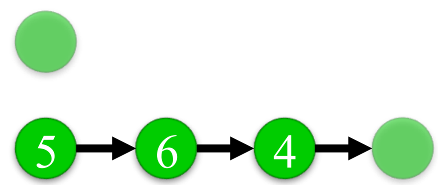

```
list_t list2 = list_make(1,
               list_make(5,
               list_make(6, list1)));
```



list_first() and list_rest()

```
empty        list1         list2
```


```
list_first(list2); // 1
list_first(empty); // BAD!

list_rest(list1);
list_rest(list2);
list_rest(list_rest(list1)); // BAD!

list_first(list_rest(list2)); // 5
```



list_isEmpty()

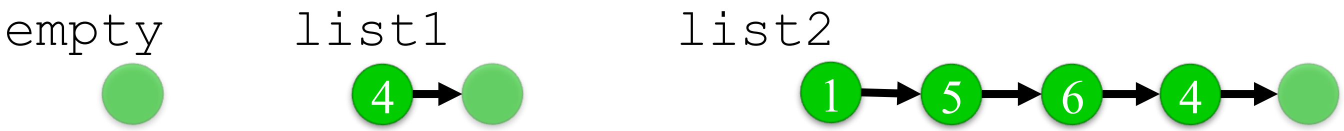

```
list_isEmpty(list2); // false
list_isEmpty(empty); // true
list_isEmpty(list_rest(list1)); // true
```

List properties

- Lists are *immutable*
  - There is no `list_setElement`
- Think of lists as values
  - To “modify” a list, you just have to “compute” a new one with desired changes
- What does this line do?

```
list2 = list_make(0, list_rest(list_2));
```

Exercise: list_max()




```
// REQUIRES: list must not be empty
// EFFECTS:  Returns the largest element in list
int list_max (list_t list){
```

Use “general” recursion

```
}
```

```
max(empty); // BAD!
max(list1); // 4
max(list2); // 6
```

Solution: list_max()

```
static int max(int x, int y){
  return x > y ? x : y;
}

// REQUIRES: list must not be empty
// EFFECTS:  Returns the largest element in list
int list_max (list_t list){
  if  (list_isEmpty(list)) {
    return -9999; //HACK!
  }
  return max(
    list_first(list),
    list_max(list_rest(list))
    );
}
```

Solution: list_max()

```
static int max(int x, int y){
  return x > y ? x : y;
}

// REQUIRES: list must not be empty
// EFFECTS:  Returns the largest element in list
int list_max (list_t list){
  assert(!list_isEmpty(list);
  if (list_isEmpty(list_rest(list)) { //hack fixed :)
    return list_first(list); //single element
  }
  return max(
    list_first(list),
    list_max(list_rest(list))
    );
}
```

Exercise: list_max() tail

```
empty      list1        list2
```


```
// REQUIRES: list must not be empty
// EFFECTS:  Returns the largest element in list
int list_max (list_t list){
```

Use tail recursion recursion
Hint: make a helper function with
an extra parameter if needed

```
}
```

```
max(empty); // BAD!
max(list1); // 4
max(list2); // 6
```

Solution: list_max() tail

```
static int max(int x, int y){ return x > y ? x : y; }

int list_max_h(list_t list, int soFar){
  if (list_isEmpty(list)) return soFar;
  return list_max_h(
    list_rest(list),
    max(list_first(list), list_max(list_rest(list))
    );
}

int list_max (list_t list){
  assert(!list_isEmpty(list);
  return list_max_h(list, list_first(list));
}
```

Converting list_max to Iteration

Tail Recursive

```
int list_max_h(list_t list, int soFar){
  ...
  ...
}

int list_max (list_t list){
  return list_max_h(list, list_first(list));
}
```

Accumulator

Initial

Iterative

```
int list_max(list_t list){


  int soFar = list_first(list);
```

Converting list_max to Iteration

Tail Recursive

```
int list_max_h(list_t list, int soFar){
  if  (list_isEmpty(list)){ // BASE CASE
    return soFar; // final result

  . // RECURSIVE CASE


  list_max (list_t list){ ... }
```

Repetition
(Stop at base case)

Iterative

```
  list_max(list_t list){
    soFar = list_first(list);

  while (!list_isEmpty(list)){

  }
  return soFar;
}
```

Converting list_max to Iteration

Tail Recursive

```
int list_max_h(list_t list, int soFar){
  ... // BASE CASE

  // RECURSIVE CASE
 return list_max_h(list_rest(list),
                   max(list_first(list), soFar));
}

int list_max (list_t list){ ... }
```

Compiler handles these!

Computation (Update)

Iterative

```
int list_max(list_t list){
  int soFar = list_first(list);

  while (!list_isEmpty(list)){
    list = list_rest(list);
    soFar = max(list_first(list), soFar);
  }
  return soFar;
}
```

Update ordering matters!

Converting list_max to Iteration

```
while (!list_isEmpty(list)){
  list = list_rest(list);
  soFar = max(list_first(list), soFar);
}
```

Ordering 1

```
while (!list_isEmpty(list)){
  soFar = max(list_first(list), soFar);
  list = list_rest(list);
}
```

Ordering 2

- The new value of `soFar` depends on `list`
- The new value of `list` does NOT depend on `soFar`
- Ordering 2 is safe, but ordering 1 is not

Dependency Graphs

- Allow us to determine an ordering to safely update variables.
- An arrow from one variable to another indicates a dependency.
  - e.g. new soFar depends on current list

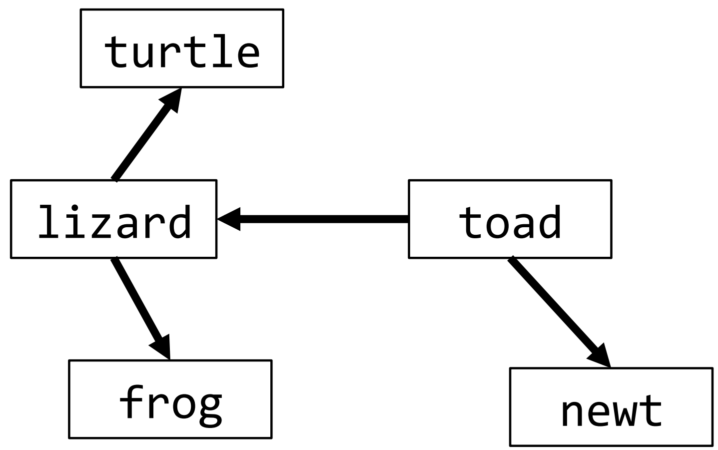
  So we must update soFar before list.

  `soFar` → `list`

- In general, we "topologically sort" the dependency graph to determine update order

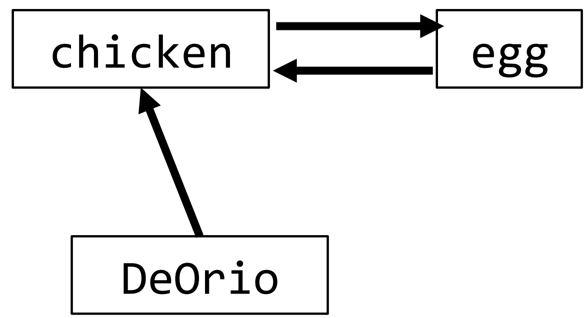
Exercise: Dependency Graphs

- Find a suitable ordering for updating the variables with this dependency graph



Dependency Graphs: Cycles

- What if you have a cycle in the dependency graph?



- Shadow variables

```
int cTemp = chicken;
int eTemp = egg;
chicken = // use eTemp;
egg = // use cTemp;
```

So…where are we now?

- Tail recursion and iteration are the same
  - We just walked through a constructive proof
  - Should you ever use tail recursion?
    Pro: Compiler handles update dependencies
    Pro: Sometimes base/recursive cases easy
    Con: Sometimes iteration is more intuitive
    Con: Can be tricky to ensure TCO

- "General" recursion is expensive
  - Stack frames sit and store pending work
  - Why would you ever use this?
  - Sometimes you need it (example: project 2 tree functions)