CSE 341
Lecture 5

efficiency issues; tail recursion; `print`

Ullman 3.3 - 3.4; 4.1

slides created by Marty Stepp

http://www.cs.washington.edu/341/

Efficiency exercise

- Write a function called reverse that accepts a list and produces the same elements in the opposite order.
  - `reverse([6, 2, 9, 7])` produces `[7,9,2,6]`

- Write a function called `range` that accepts a maximum integer value *n* and produces the list [1, 2, 3, ..., *n*-1, *n*]. Produce an empty list for all numbers less than 1.
  - Example: `range(5)` produces `[1,2,3,4,5]`

Flawed solutions

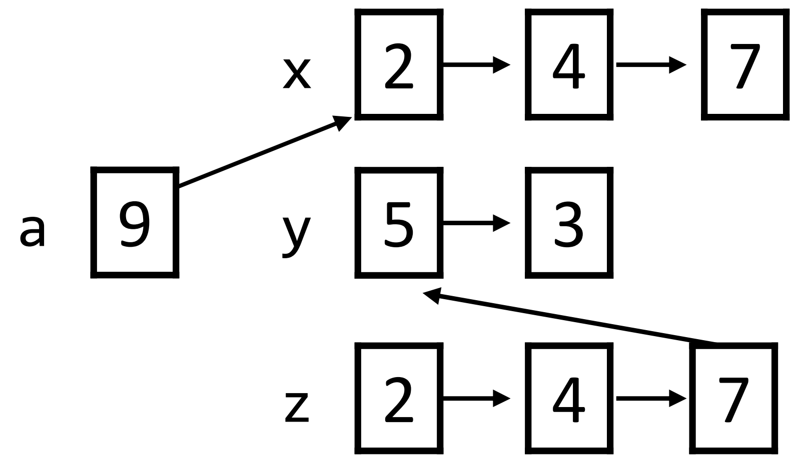
- These solutions are correct; but they have a problem...

```
fun reverse([]) = []
|   reverse(first :: rest) =
            reverse(rest) @ [first];

fun range(0) = []
|   range(n) = range(n - 1) @ [n];
```

Efficiency of the @ operator

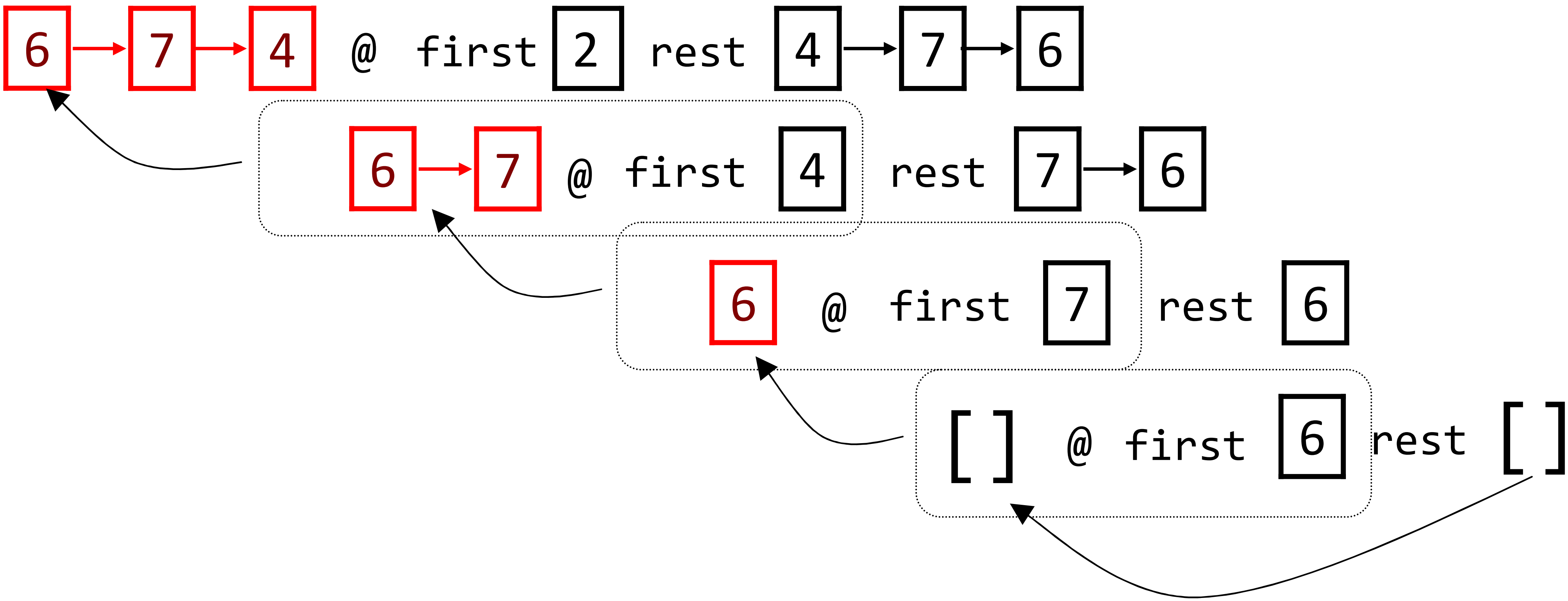
```
val x = [2, 4, 7];
val y = [5, 3];
val a = 9 :: x;
val z = x @ y;
```



- The `::` operator is fast: O(1)
  - simply creates a link from the first element to front of right
- The @ operator is slow: O(*n*)
  - must walk/copy the left list and then append the right one
  - using @ in a recursive function *n* times : function is O(n^2)

Flawed solution in action

```
fun reverse([]) = []
|   reverse(first :: rest) =
              reverse(rest) @ [first];
reverse([2, 4, 7, 6]);
```



Fixing inefficient reverse

- How can we improve the inefficient `reverse` code?

```
fun reverse([]) = []
|   reverse(first :: rest) =
            reverse(rest) @ [first];
```

  - *Hint:* Replace @ with `::` as much as possible.
  - `::` adds to the front of a list. How can we perform a reversal by repeatedly adding to the front of a list? (Think iteratively...)

Better reverse solution

```
fun reverse([]) = []
|   reverse(L)
    let (* lst accumulates reversed values *)
        fun helper(lst, []) = lst
        |   helper(lst, first::rest) =
                    helper(first::lst, rest)
    in
        helper([], L)
    end;
```

- The parameter `lst` here serves as an *accumulator*.

Fixing inefficient range

- How can we improve the inefficient `range` code?

```
fun range(0) = []
|   range(n) = range(n - 1) @ [n];
```

  - *Hint:* Replace @ with `::` as much as possible.
  - *Hint:* We can't build the list from front to back the way it's currently written, because *n* (the max of the range) is the only value we have available.
  - *Hint:* Consider a helper function that can build a range in order from smallest to largest value.

Better range solution

```
fun range(n) =
    let
        fun helper(lst, i) =
            if i = 0 then lst
            else helper(i :: lst, i - 1)
    in
        helper([], n)
    end;
```

- The parameter `lst` here serves as an *accumulator*.

Times-two function

- Consider the following function:

```
(* Multiplies n by 2; a silly function. *)
fun timesTwo(0) = 0
|   timesTwo(n) = 2 + timesTwo(n - 1);
```

- Run the function for large values of *n*.
  Q: Why is it so slow?

- **A:** Each call must wait for the results of all the other calls to return before it can add 2 and return its own result.

Tail recursion

- **tail recursion**: When the end result of a recursive function can be expressed entirely as one recursive call.

- Tail recursion is *good* .
  A smart functional language can detect and optimize it.
  - If a call `f(x)` makes a recursive call `f(y)`, as its **last** action, the interpreter can discard `f(x)` from the stack and just jump to `f(y)`.

- Essentially a way to implement iteration recursively.

Times-two function revisited

- This code is not tail recursive because of **`2 +`**

```
(* Multiplies n by 2; a silly function. *)
fun timesTwo(0) = 0
|   timesTwo(n) = 2 + timesTwo(n - 1);
```

- *Exercise:* Make the code faster using an *accumulator*.

- **accumulator**: An extra parameter that stores a partial result in progress, to facilitate tail recursion.

Iterative times-two in Java

```java
// Multiplies n by 2; a silly function.
public static int timesTwo(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum = sum + 2;
    }
    return sum;
}
```

Iterative times-two in Java, v2

```
// Multiplies n by 2; a silly function.
public static int timesTwo(int n) {
    int sum = 0;
    while (n > 0) {
        sum = sum + 2;
        n = n - 1;
    }
    return sum;
}
```

Tail recursive times-two in ML

```
(* Multiplies n by 2; a silly function. *)
fun timesTwo(n) =
    let
        help(sum, 0) = sum
    |   help(sum, k) = help(sum + 2, k - 1)
    in
        help(0, n)
    end;
```

- Accumulator variable `sum` grows as `n` (`k`) shrinks.

Efficiency and Fibonacci

- The `fibonacci` function we wrote previously is also inefficient, for a different reason.
  - It makes an exponential number of recursive calls!
  - Example: `fibonacci(5)`
    - `fibonacci(4)`
      - `fibonacci(3)`
        - » `fibonacci(2)`
        - » `fibonacci(1)`
      - `fibonacci(2)`
    - `fibonacci(3)`
      - `fibonacci(2)`
      - `fibonacci(1)`
  - How can we fix it to make fewer (O(*n*)) calls?

Iterative Fibonacci in Java

```java
// Returns the nth Fibonacci number.
// Precondition: n >= 1
public static int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    int curr = 1;    // the 2 most recent Fibonacci numbers
    int prev = 1;

    // k stores what fib number we are on now
    for (int k = 2; k < n; k++) {
        int next = curr + prev;   // advance to next
        prev = curr;              // Fibonacci number
        curr = next;
    }
    return curr;
}
```

Efficient Fibonacci in ML

```
(* Returns the nth Fibonacci number.
   Precondition: n >= 1 *)
fun fib(1) = 1
|   fib(2) = 1
|   fib(n) =
        let
            fun helper(k, prev, curr) =
                if k = n then curr
                else helper(k + 1, curr, prev + curr)
        in
            helper(2, 1, 1)
        end;
```

The print function (4.1)

`print(`***`string`***`);`

- The type of print is `fn : string -> unit`
  - `unit` is a type whose sole value is `()` (like `void` in Java)
  - unlike most ML functions, print has a *side effect* (output)

- `print` accepts only a string as its argument
  - can convert other types to string:
    `Int.toString(`***`int`***`)`, `Real.toString(`***`real`***`)`,
    `Bool.toString(`***`bool`***`)`, `str(`***`char`***`)`, etc.

"Statement" lists

`(`***`expression`***`; `***`expression`***`; `***`expression`***`)`

- evaluates a sequence of expressions; a bit like `{}` in Java
- the above is itself an expression
  - its result is the value of the *last* expression

- might seem similar to a `let`-expression...
  - but a `let` modifies the ML environment (defines symbols); a "statement" list simply evaluates expressions, each of which might have side effects

Using print

```
- fun printList([]) = ()
= |   printList(first::rest) = (
          print(first ^ "\n");
          printList(rest)
      );
```

val printList = fn : string list -> unit

```
- printList(["a", "b", "c"]);
```

a

b

c

val it = () : unit

print for debugging

```
(* Computes n!; not tail recursive. *)
fun factorial(0) = 0
|   factorial(n) = (
        print("n is " ^ str(n));
        n * factorial(n - 1)
    );
```

- Useful pattern for debugging:
  - (print(***whatever***); your original code)