CS1101S

AY25/26 sem 1

github.com/lostmusician

01 recursion

examples

```
function factorial(n) {
    if (n === 0) return 1; // base case
    return n * factorial(n - 1); // recursive case
}

function factorial(n) {
    return iter(1, 1, n);
}

function iter(product, counter, n) {
    return counter > n
        ? product // base case
        : iter(product * counter, counter + 1, n); // recursive case
}
```

```
function fib(n) {
    function f(n, k, x, y) {
        return (k > n)
            ? y
            : f(n, k + 1, y, x + y);
    }
    return (n < 2) ? n : f(n, 2, 0, 1);
}
```

```
function gcd(a, b) {
    return b === 0
        ? a
        : gcd(b, a % b);
}
```

wishful thinking: assuming that the back is already solved

CPS

instead of letting the call stack remember, we explicitly pass a function c that encodes "what to do next."

```
function append_iter(xs, ys){
    // iterative process
    function app(xs, ys, c) {
        return is_null(xs)
        ? c(ys)
        : app(tail(xs), ys,
              x => c(pair(head(xs), x))
              );
    }
    return app(xs, ys, x => x);
}
```

```
function fast_expt(b, n) {
    return n === 0
          ? 1
          : is_even(n)
          ? square(fast_expt(b, n / 2))
          : b * fast_expt(b, n - 1);
}

function fast_expt_cps(b, n, c) {
    return n === 0
           ? c(1)
           : is_even(n)
           ? fast_expt_cps(b, n / 2, x => c(square(x)))
           : fast_expt_cps(b, n - 1, x => c(b * x));
}
```

02 lists and trees

A tree of certain data items is a list whose elements are such data items, or trees of such data items.

make tree takes 3 args: entry, left branch, right branch

```
function BST_to_list(bst) {
    if (is_null(bst)) {
        return null;
    } else {
        const ltree = head(tail(bst));
        const num = head(bst);
        const rtree = head(tail(tail(bst)));
        return append(BST_to_list(ltree),
                      pair(num,
                      BST_to_list(rtree)));
    }
}
```

```
function map_tree(f, tree) {
    return map(sub_tree => !is_list(sub_tree)
                ? f(sub_tree)
                : map_tree(f, sub_tree)
              , tree);
}
```

```
function flatten_tree(xs) {
    function h(xs, prev) {
        return is_null(xs)
            ? prev
            : is_list(xs)
              ? append(flatten_tree(xs), prev)
              : pair(xs, prev);
    }
    return accumulate(h, null, xs);
}
```

```
function insert(bst, item) {
    if (is_empty_tree(bst)) {
        return make_tree(item,
                         make_empty_tree(),
                         make_empty_tree());
    } else {
        if (item < entry(bst)) {
            // smaller than entry(left branch)
            return make_tree(entry(bst),
                             insert(left_branch(bst),
                                    item),
                             right_branch(bst));
        } else if (item > entry(bst)) {
            // bigger than entry (right branch)
            return make_tree(entry(bst),
                             left_branch(bst),
                             insert(right_branch(bst),
                                    item));
        } else {
            // equal to entry.
            return bst;
        }
    }
}
```

```
function find(bst, name) {
    return is_empty_tree(bst)
        ? false
        : name === entry(bst)
         ? true
         : name < entry(bst)
          ? find(left_branch(bst), name)
          : find(right_branch(bst), name);
}
```

matrix

remember to use listref

```
function transpose(M) {
    const nR = length(M); // number of rows
    const nC = length(head(M)); // columns
    return map(c => map(row => list_ref(row, c), M), enum_list(0, nC - 1));
}

function row_sums(M) {
    return map(row => accumulate((x, sum) => x + sum, 0, row), M);
}
```

```
function map_using_accumulate(f, xs) {
    return accumulate((x, result) =>
                pair(f(x), result), null, xs);
}

function filter_using_accumulate(pred, xs) {
    return accumulate(
        (x, result) => pred(x)
        ? pair(x, result) : result,
        null,
        xs
    );
}
```

03 permutations and combinations

```
function permutations(s) {
    return is_null(s)
        ? list(null)
        : accumulate(append, null,
                     map(x => map(p => pair(x, p)),
                     permutations(remove(x, s))),
                     s));
}
```

```
function subsets(s) {
    return accumulate(
        (x, s1) => append(s1,
                   map(ss => pair(x, ss), s1)),
        list(null),
        s);
}
function choose(n, r) {
    if (n < 0 || r < 0) {
        return 0;
    } else if (r === 0) {
        return 1;
    } else {
        // Consider the 1st item, there are 2 choices:
        // To use, or not to use
        // Get remaining items with wishful thinking
        const to_use = choose(n - 1, r - 1);
        const not_to_use = choose(n - 1, r);

        return to_use + not_to_use;
    }
}

function combinations(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const no_choose =
        combinations(tail(xs), r);
        const yes_choose =
        combinations(tail(xs),
                                          r - 1);
        const yes_item = map(x =>
        pair(head(xs), x),
                        yes_choose);
        return append(no_choose, yes_item);
    }
}
```

04 sorting algorithms

insertion, selection, quicksort, mergesort

```
function insert(x, xs) {
    return is_null(xs)
        ? list(x)
        : x <= head(xs)
          ? pair(x, xs)
          : pair(head(xs), insert(x, tail(xs)));
}

function insertion_sort(xs) {
    return is_null(xs)
        ? xs
        : insert(head(xs),
```

```
                insertion_sort(tail(xs)));
}
```

Insertion sort: Builds the sorted list one item at a time by taking each element and inserting it into its correct place among the already-sorted elements.

```
function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(x,
            selection_sort(remove(x, xs)));
    }
}

function smallest(xs) {
    function h(xs, min) {
        return xs === null
            ? min
            : head(xs) < min
                ? h(tail(xs), head(xs))
                : h(tail(xs), min);
    }
    return h(xs, head(xs));
}
```

Selection sort: Repeatedly finds the minimum (or maximum) element from the unsorted part and puts it at the beginning.

```
function partition(xs, p) {
    function h(xs, lte, gt) {
        if (is_null(xs)) {
            return pair(lte, gt);
        } else {
            const first = head(xs);
            return first <= p
                ? h(tail(xs), pair(first,
                    lte), gt)
                : h(tail(xs), lte, pair(first,
                    gt));
        }
    }
    return h(xs, null, null);
}


function quicksort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const pivot = head(xs);
        const splits = partition(tail(xs),
            pivot);
        const smaller =
            quicksort(head(splits));
        const bigger = quicksort(tail(splits));
        return append(smaller, pair(pivot,
            bigger));
    }
}
```

Quick sort: Picks a pivot element, partitions the list into elements less than the pivot and greater than the pivot, then recursively sorts the partitions.

```
function take(xs, n) {
    return n === 0
        ? null
        : pair(head(xs),
               take(tail(xs), n - 1));
}
function drop(xs, n) {
    return n === 0
        ? xs
        : drop(tail(xs), n - 1);
}

function merge(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else if (is_null(ys)) {
        return xs;
    } else {
        const x = head(xs);
        const y = head(ys);
        return (x < y)
            ? pair(x, merge(tail(xs), ys))
            : pair(y, merge(xs, tail(ys)));
    }
}

function merge_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const mid = math_floor(length(xs) / 2);
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs,
                         mid)));
    }
}
```

Merge sort: Recursively splits the list in half, sorts each half, then merges the two sorted halves back together.

05 time complexity identifiers

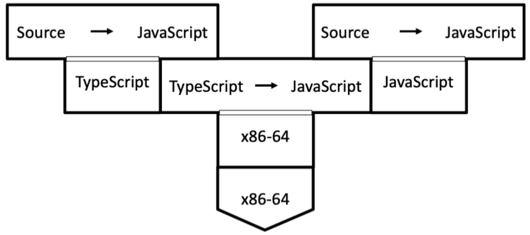
Recurrence	Complexity
$\mathcal{O}(1) + T(n-1)$	$\mathcal{O}(n)$
$\mathcal{O}(1) + 2T(n/2)$	$\mathcal{O}(n)$
$\mathcal{O}(n) + T(n/2)$	$\mathcal{O}(n)$
$\mathcal{O}(1) + T(n/2)$	$\mathcal{O}(\log n)$
$\mathcal{O}(\log n) + T(n-1)$	$\mathcal{O}(n \log n)$
$\mathcal{O}(n) + 2T(n/2)$	$\mathcal{O}(n \log n)$
$\mathcal{O}(n) + T(n-1)$	$\mathcal{O}(n^2)$
$\mathcal{O}(n^k) + T(n-1)$	$\mathcal{O}(n^{k+1})$
$\mathcal{O}(n) + 2T(n-1)$	$\mathcal{O}(2^n)$

Algorithm	Best/Avg Time	Worst Time	Space
Binary Search	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Selection Sort	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion Sort	$\Theta(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Merge Sort	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\Theta(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

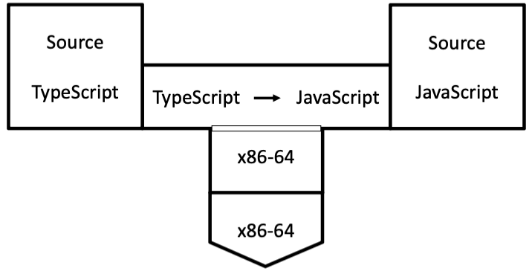
Time complexity will always be greater than or equal to space complexity since we need time to create space.

Big Theta of one call f(n)	Calls spawned	Big Theta of whole thing
Θ(1)	f(n - 1)	Θ(n)
Θ(1)	f(n/k)	Θ(logn)
Θ(1)	f(n - 1) and f(n - 1)	Θ(2^n)
Θ(1)	f(n/k) and f(n/k)	Θ(n)
Θ(n)	f(n - 1)	Θ(n^2)
Θ(n)	f(n/k)	Θ(n)
Θ(n)	f(n/k) and f(n/k)	Θ(nlogn)
Θ(logn)	f(n - 1)	Θ(nlogn)
Θ(n^k)	f(n - 1)	Θ(n^(k + 1))
Θ(2^n)	f(n - 1)	Θ(2^n)

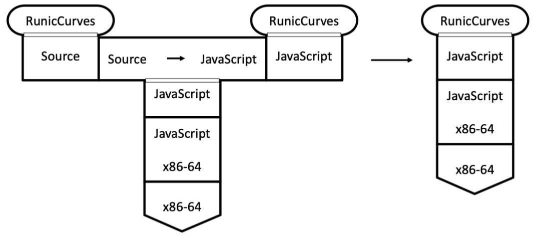
06 T diagrams



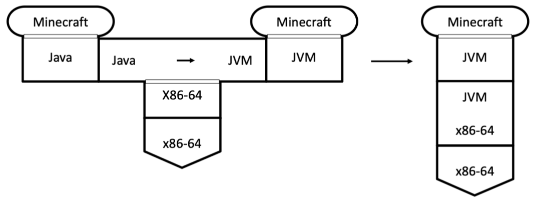
compiling a compiler



compiling the stepper from TS to JS



compiling and executing a web program



compiling and executing a program