CS1101S Finals Revision

By Wu Xiaoyun AY20/21 Sem 1

CPS

Lec 2: Substitution Models

- Applicative Order Reduction (Source):
  - evaluate the arg before applying the fxn
- Normal Order Reduction:
  - sub the arg as it is until the whole expression contains primitives before it is evaluated
  - "fully expand then reduce"

Lec 3: HOF & Scopes

HOF

- manipulate functions as arguments and return statements

Check for prime numbers

```
function is_prime(n) {
    function g(d){
        return d === 1
        ? true
        : ( n % d !== 0) && g(d - 1);
    }
    return g(n - 1);
}
```

Generalised sum fuction

```
function sum(term, a, next, b) {
    return a > b
           ? 0
           : term(a) + sum(term, next(a), next, b);
}
```

Example Usage

Computes the cubes from *a* to *b*:

```
function cube(x) {
    return x * x * x;
}

function sum(term, a, next, b) {
    return a > b
           ? 0
           : term(a) + sum(term, next(a), next, b);
}

function inc(n) {
```

```
    return n + 1;
}
function sum_cubes(a, b) {
    return sum(cube, a, inc, b);
}

sum_cubes(1, 10); // return 3025;
```

Computes the sum of intergers from *a* to *b*:

```
function sum(term, a, next, b) {
    return a > b
           ? 0
           : term(a) + sum(term, next(a), next, b);
}

function inc(n) {
    return n + 1;
}

function identity(x) {
    return x;
}

function sum_integers(a, b) {
    return sum(identity, a, inc, b);
}

sum_integers(1, 10); //returns 55;:
```

Computes the sum of *1 x 2 + 2 x 3 + ... + n(n - 1)*:

```
function my_sum(n) {
    return n === 1
        ? 1 * 2
        : my_sum(n - 1) + n * (n - 1);
}
```

Iterative version using sum:

```
function my_sum(n) {
    return sum(k => k * (k - 1), 1, k => k + 1, n);
}
```

Computes the first *n* odd numbers:

```
function sum_odd(n) {
    function identity(x) {
        return x;
}
    function plus_two(x) {
        return x + 2;
}
    const a = 1;
        return sum(identity, a, plus_two, 2 * n - 1);
```

```
}

//equivalent using lambda expression
function sum_odd(n) {
    return sum(x => x, 1, x => x + 2, 2 * n - 1);
}

sum_odd(5); //returns 25;
```

Implementing *accumulate* using sum:

```
function accumulate(combiner, term, a, next, b, base) {
    return base === 0
        ? a > b
            ? 0
            : term(a) + sum(term, next(a), next, b)
        : base === 1
            ? a > b
                ? 1
                : term(a) * product(term, next(a), next, b)
            : undefined;
}

// Usage examples:

function sum(term, a, next, b) {
  return accumulate( (x, y) => x + y, term, a, next, b, 0);
}

function product(term, a, next, b) {
  return accumulate( (x, y) => x * y, term, a, next, b, 1);
}

function fact(n) {
    return product(x => x, 1, x => x + 1, n);
}
```

Scopes

- All names must be declared
    - As pre-declared constants
    - In constant declaration statements
    - As parameters of function declaration statements and lambda expressions
    - As function name of function declaration statements
- A name occurrence refers to the closest surrounding declaration {...}

1. check parameters
2. check const/fxn declaration within the {...} block
3. check the whole program (before the {...} block)

```
function f() {
    return "hello";
}

f; // returns function
/*
function f() {
    return "hello";
}
*/

f(); // returns function application;
//"hello"
```

```
const f = () => 1;
const g = () => f;
const h = () => f();

g(); // ()=>f
h(); // 1
```

Lec 4: List and Trees

- A list of a certain data type is **null** or a pair whose head is of that data type and whose tail is a list of that data type (i.e. tail is null).
- A tree of a certain data type is a **list** whose elements are of that data type, or trees of that data type.
    - **null & pairs** are not considered data types.

Lec 5: CPS, Filter, Map, Accumulate

Continuation-Passing Style (CPS)

- Passing the deferred operation as a function in an extra argument
- can convert any recursive function this way

```
// Recursive process
function append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs), append(tail(xs), ys));
}

// Iterative process (CPS)
function app(current_xs, ys, c) {
    return is_null(current_xs)
        ? c(ys)
        : app(tail(current_xs), ys, x => c(pair(head(current_xs), x)));
}
```

```
14
15 function append_iter(xs, ys) {
16     return app(xs, ys, x => x);
17 }
```

```
append_iter(list(1, 2), list(3, 4));

app(list(1, 2), list(3, 4), x => x);

app(list(2), list(3, 4), y => (x => x)(pair(1, y)));

app(null, list(3, 4), z => (y => (x => x)(pair(1, y)))(pair(2, z)));

(z => (y => (x => x)(pair(1, y)))(pair(2, z)))(list(3, 4));

(y => (x => x)(pair(1, y)))(list(2, 3, 4));

(x => x)(list(1, 2, 3, 4));

list(1, 2, 3, 4);
```

```
1  function plus(x, y) {
2      return x + y;
3  }
4
5  function plus_cps(x, y, ret) {
6      return ret(x + y);
7  }
8  // In order to display the result of the addition of 1 and 2, we can use
   plus_cps as follows:
9  plus_cps(1, 2, display); // displays the value 3
10
11 function sum_cps(x, y, z, ret) {
12     return plus_cps(x, y, x_plus_y => plus_cps(x_plus_y, z, ret));
13 }
14 sum_cps(1, 2, 3, display); // displays the value 6
15
16 ------------------------------------------------------------
17
18 function length(xs) {
19     if (is_null(xs)) {
20         return 0;
21     } else {
22         return 1 + length(tail(xs));
23     }
24 }
25 function length_cps(xs, ret) {
26     if (is_null(xs)) {
27         return ret(0);
28     } else {
29         return length_cps(tail(xs), tail_result => ret(1 + tail_result));
30     }
31 }
32 length_cps(list(10, 20, 30), display); // displays value 3
33
34 ------------------------------------------------------------
35
36 function factorial(n) {
37     if (n <= 0) {
38         return 1;
```

```
    } else {
        return n * factorial(n - 1);
    }
}

function factorial_cps(n, ret) {
    if (n <= 0) {
        return ret(1);
    } else {
        return factorial_cps(n - 1, result => ret(n * result));
    }
}
factorial_cps(5, display); // displays the value 120

function fact_iter_cps(n, acc, ret) {
    if (n <= 0) {
        return ret(acc);
    } else {
        return fact_iter_cps(n - 1, n * acc, ret);
    }
}

function factorial_iter_cps(n, ret) {
    return fact_iter_cps(n, 1, ret);
}
factorial_iter_cps(5, display); // displays 120

// When we turn iterative functions into CPS, the continuation function is
passed unchanged in the recursive call
```

map(f, xs)

- applies function *f* to all elements insides xs
- does **NOT** modify the original list
- Parameters:
  - f: **unary** function (x => sth here)
  - xs: a list
- Returns:
  - the mapped list

```
list(a1, a2, ... , an-1, an); // list before mapping
list(f(a1), f(a2), ... , f(an-1), f(an)); // list after mapping

// Example
map(x => x + 1, list(1, 2, 3));
// returns list(2, 3, 4)
```

```
const xs = list(1, 2, 3);
const ys = map(x ⇒ x + 1, xs);

display(xs); // displays list(1, 2, 3)
display(ys); // displays list(2, 3, 4)
```

filter(pred, xs)

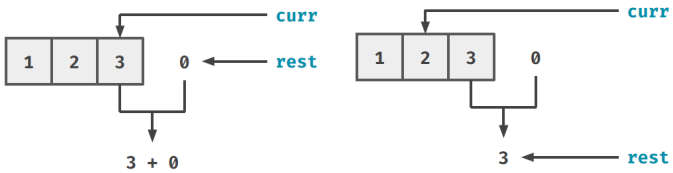
- If predicate (test) returns *true* for the element, then it will pass the test and be part of the resulting list.
- Parameters:
  - pred: **unary** predicate function (x => sth here)
  - xs: a list
- Returns
  - the filtered list

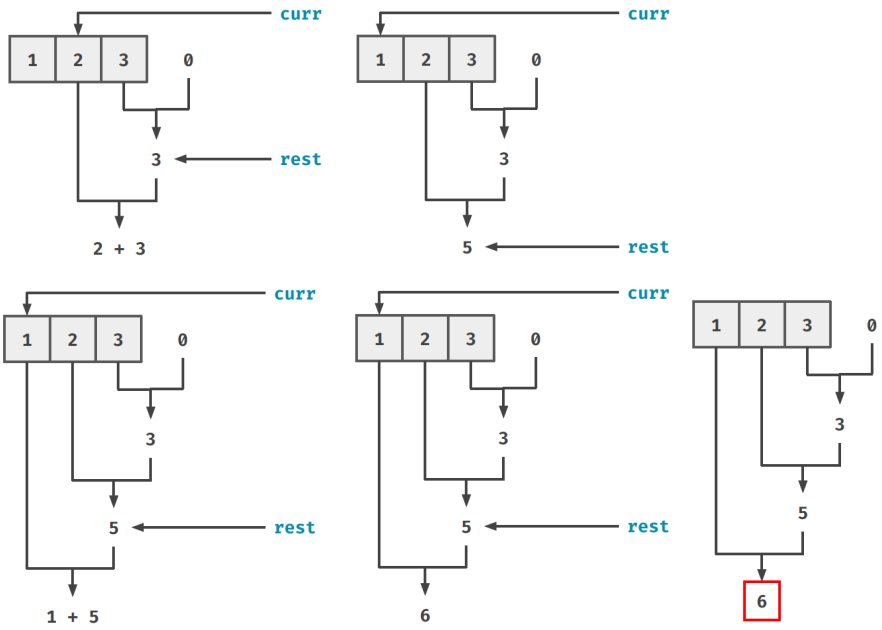
```
const is_even = x ⇒ x % 2 === 0;

filter(is_even, list(1, 2, 3, 4, 5, 6));
// Returns list(2, 4, 6)
// filter only allows the even numbers: 2, 4, 6 to pass through
```

accumulate(f, initial, xs)

- Things to note:
  - operations are done from **Right to Left** (**NOT** L to R)
  - 1st parameter refers to the current element,
  - 2nd parameter refers to the current accumulated value
- Parameters:
  - f: **binary** function ((x,y) => sth here)
  - initial: the initial value *(e.g. usually null for functions involving lists)*
  - xs: a list
- Returns:
  - the accumulated value

```
1  accumulate((x, y) => x + y, 0, list(1,2,3));
2  accumulate((curr, rest) => curr + rest, 0, list(1,2,3));
3
```





Lec 6: BST, Sorting

Binary Tree

- A binary tree of a certain type is `null` or a list with 3 elements, whose 1st element is of that type and whose 2nd and 3rd element are binary trees of that type.
- (value, left_subtree, right_subtree)

Binary Search Tree

- A BST of Strings is a binary tree of Strings where all entries in the left subtree are smaller than its value and all entries in the right subtree are larger than its value.
- There are **NO** duplicates

Selection Sort

Finding the smallest element in the list:

```
1  function smallest(xs) {
2      if (is_null(tail(xs)) || head(xs) < head(tail(xs))) {
3          return head(xs);
4      }  else {
5          return smallest(tail(xs));
6      }
7  }
8
9  function smallest(xs) {
10     return accumulate( (x,y) => is_null(y) ? x : math_min(x, y),
11                        null,
12                        xs);
13 }
14
15 function smallest(xs) {
```

```
    return accumulate(math_min, Infinity, xs);
}
```

Finding the largest element in the list:

```
function largest(xs) {
    return accumulate( (x,y) => is_null(y) ? x : math_max(x, y),
                       null,
                       xs);
}

function largest(xs) {
    return accumulate(math_max, null, xs);
}
```

```
// version 1
function smallest(xs) {
    return accumulate(math_min, Infinity, xs);
}

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

// version 2
function find_min(xs) {
    function helper(ys, smallest_so_far, acc) {
        return (is_null(ys))
            ? pair(smallest_so_far, acc)
            : (smallest_so_far > head(ys))
                ? helper(tail(ys), head(ys), pair(smallest_so_far, acc))
                : helper(tail(ys), smallest_so_far, pair(head(ys), acc));
    }
    return helper(tail(xs), head(xs), null);
}

find_min(list(2,4,1,3,6)); // return list(1,6,3,2,4);

function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const xss = find_min(xs);
        return pair(head(xss), selection_sort(tail(xss)));
    }
}

// order of growth = n^2
```

Insertion Sort

```
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)));
}

// order of growth = n^2
```

Merge Sort

```
// version 1
// put the first n elements of xs into a list
function take(xs, n) {
    return (n === 0)
            ? null
            : pair(head(xs), take(tail(xs), n - 1));

}

// drop the first n elements from list, return rest
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)));
    }
}
```

```
// version 2
function take_drop(xs, n) {
    function helper(ys, k, acc) {
        return k === 0
            ? pair(acc, ys)
            : helper(tail(ys), k - 1, pair(head(ys), acc));
    }
    return helper(xs, n, null);
} //returns pair(list of first k elements, list of remaining (n - k)
elements)

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 td = take_drop(xs, math_floor(length(xs) / 2));
        return merge(merge_sort(head(td)),
                     merge_sort(tail(td)));
    }
}

// order of growth = n log n
```

Quick Sort

```
function partition(xs, p) {
    function list_lte(xs, p) {
        return filter((x => x < p), xs);
    }

    function list_gt(xs, p) {
        return filter((x => x > p), xs);
    }

    return pair(list_lte(xs,p), list_gt(xs, p));
```

```
}

function quicksort(xs) {
    if (length(xs) <= 1) {
        return xs;
    } else {
        return append(quicksort(head(partition(tail(xs), head(xs)))),
                    pair(head(xs),
                            quicksort(tail(partition(tail(xs),
head(xs))))));
    }
}

/*
- order of growth in time for applying partition to a list of length n: n
- order of growth in time for applying quicksort to an already sorted list
of length n: n^2
- For lists of length n, the performance of quicksort may vary. Let f(n) be
the fastest runtime of quicksort for any list with length n. What is the
order of growth of the function f(n), using Θ notation?:  n log n
*/
```

Lec 7: Variable assignment, Mutable Data

```
// Variable
let name = expression;
// function parameters are variables
```

- assignment allows us to create objects with state => create mutable data structures
- state allows objects to behave differently over time => substitution model breaks down => envt model
- mutable data: data that can be modified

d_append

```
function append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs), append(tail(xs), ys));
}

function d_append(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else {
        set_tail(xs, d_append(tail(xs), ys));
        return xs;
    }
}
```

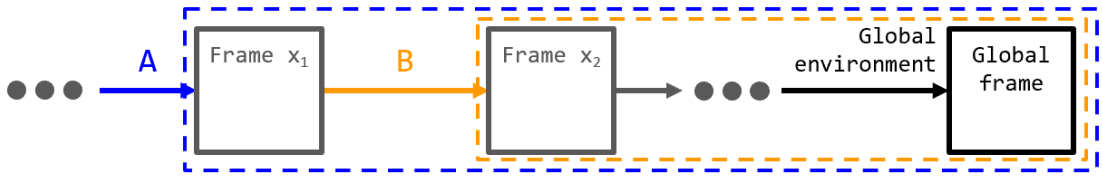
d_map

```
function map(f, xs) {
    return is_null(xs)
        ? null
        : pair(f(head(xs)),
               map(f, tail(xs)));
}

function d_map(f, xs) {
    if (!is_null(xs)) {
        set_head(xs, f(head(xs)));
        d_map(f, tail(xs));
    } else {}
}
// does not return the new xs; only changes the values in the list;
// new xs can be shown by calling xs in the last statement of the program;
```

Lec 8: Envt Model

- *refer to Alex's S9 slides for more details*
- An environment: a sequence of frames
  - Environment defines the *hierarchy of contexts* for which an expression should be evaluated under.
  - the global env: consists of a single frame with the binding s of primitive and pre-declared functions and constants
  - extending an envt means adding a new frame "inside" the old one
  - new frame not created if the function and parameter
  - Terminologies:
    - B is the *enclosing environment* of frame x1
    - Frame x1 is said to be *enclosed by* environment B
    - Frame x1 is said to *extend* environment B
    - Environment B is said to *enclose* frame x1

Environment A entails the sequence of frames represented by the blue dotted outline
Environment B entails the sequence of frames represented by the orange dotted outline



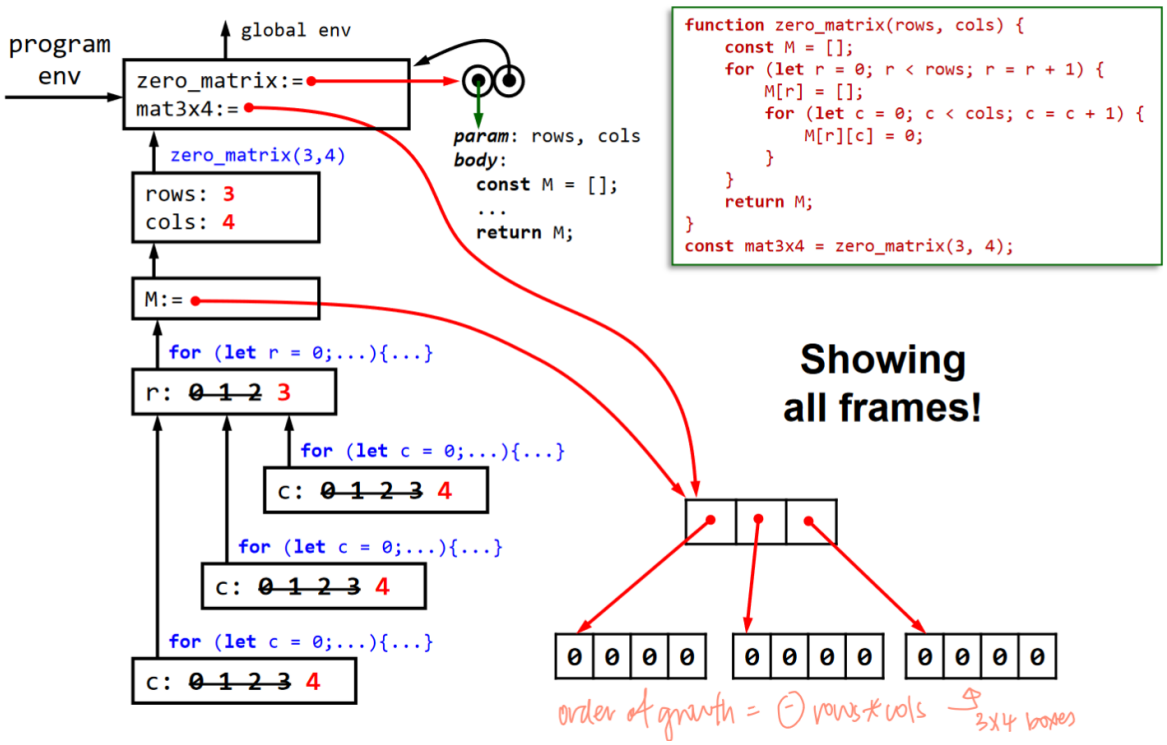
- Frames:
  - Captures the context in which the program is in
  - A set of names and bindings
  - each frame contains binding of symbols and values
- Bindings:
  - Each `symbol: value` pair is called a *binding*
  - a frame points to its enclosing envt, the next one in the sequence, unless the frame is global
  - function and constant declarations use `:=`

        - `(function name):=` & `const :=`
    - variables use `:`
        - `let name : expression`
    - primitive value (e.g. numbers, strings, Boolean values, null) in bindings are drawn inside frames
    - compound structures (e.g. pairs, lists, arrays, function objects) are drawn outside frames

Frame

$\langle symbol_1 \rangle$: $\langle value_1 \rangle$ $\langle symbol_2 \rangle$:= $\langle value_2 \rangle$ $\langle symbol_3 \rangle$: $\langle value_3 \rangle$ … $\langle symbol_n \rangle$: $\langle value_n \rangle$

- Process of looking up a variable
    - Current frame ⇒ enclosing frame ⇒ program frame ⇒ global frame
    - If not found in global frame: “Name __ not declared.” (unbound)
- Function Object
    - If named, it will have a binding in a frame, otherwise, it will just be a ‘floating’ eyeball
- Argument expressions are evaluated before the function body is called.
    - Uses the same frame in which the function application is evaluated to evaluate the arguments.
    - E.g. if function application occurs in the prog env, the arguments are also evaluated in prog env
- Loops:
    - every time when the body block is evaluated, it extends the envt by adding a new frame
    - no new frame is created if the block has no constant & var declaration



Identity

- Boolean values, string, numbers: a === b
- null === null;
- undefined === undefined;
- Functions & pairs: unique identity

```
function f() {
    return 1;
}
function g() {
    return 1;
}
// f !== g

const f = g;
// f === g
```

Lec 9: Array Sorting, Memoization

Array

- an array is a data structure that stores a sequence of data elements
- arrays are random access:
    - any value can be retrieved *O(1)* time.

```
function array_1_to_n(n) {
    const a = [];
    function iter(i) {
        if (i < n) {
            a[i] = i + 1;
            iter(i + 1);
        } else {}
    }
    iter(0);
    return a;
}

array_1_to_n(3); // [1, 2, 3]
// iter process
// if swap the sequence of a[i] and iter(i + 1), function still works but becomes a recursive process, last elements will be put into the array first

function map_array(f, arr) {
    const len = array_length(arr);
    function iter(i) {
        if (i < len) {
            arr[i] = f(arr[i]);
            iter(i + 1);
        } else {}
    }
    iter(0);
}
const seq = [3, 1, 5];
map_array(x => 2 * x, seq);
```

```
seq; // [6, 2, 10]; destructive process

function swap(A, i, j) {
    let temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

function reverse_array(A) {
    const len = array_length(A);
    const half_len = math_floor(len / 2);
    for (let i = 0 ; i < half_len; i = i + 1) {
        swap(A, i, len - 1 - i);
    }
}

function zero_matrix(rows, cols) {
    const M = [];
    for (let r = 0; r < rowsl r = r + 1){
        M[r] = []; // initialize each row to empty array so that it can be
extended later
        for (let c =  0; c < cols; c = c + 1) {
            M[r][c] = 0;
        }
    }
    return M;
}

const mat3x4 = zero_matrix(3, 4);
/*
[[0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]];
 */

function matrix_multiply_3x3(A, B) {
    const M = [];
    for (let r = 0; r < 3; r = r + 1) {
        M[r] = [];
        for (let c = 0; c < 3; c = c + 1) {
            M[r][c] = 0;
            for (let k = 0; k < 3; k = k + 1) {
                M[r][c] = M[r][c] + A[r][k] * B[k][c];
            }
        }
    }
    return M;
}
```

Linear / Sequential Search

```
function linear_search(A, v) {
    constlen = array_length(A);
    let i= 0;
    while (i < len && A[i] !== v) {
        i= i+ 1;
    }
```

```
    return (i < len);
}
linear_search([1,2,3,4,5,6,7,8,9], 5); // returns true

function make_optimized_search(A) {
    const len = array_length(A);

    const B = [];
    for (let i = 0; i < len; i = i + 1) {
        B[i] = A[i];
    }
    merge_sort(B);
    return x => binary_search(B, x);
}
```

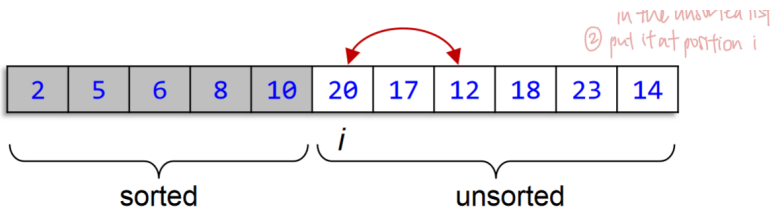
Binary Search

- input array of length n is sorted in ascending order
- idea: checking the mid element in the given range --> cut the search space into half (same as BST)
- runtime of *O (log n)*

```
function binary_search_recur(A, v) {
    function search(low, high) {
        if (low > high) {
            return false;
        } else{
            const mid = math_floor((low + high) / 2);
            return (v === A[mid]) ||(v < A[mid]
                ? search(low, mid -1)
                : search(mid + 1, high));
        }
    }
    return search(0, array_length(A) - 1);
}

function binary_search_loop(A, v) {
    let low = 0;
    let high = array_length(A) - 1;
    while (low <= high) {
        const mid = math_floor((low + high) / 2 );
        if (v === A[mid]) {
            break;
        } else if (v < A[mid]) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return (low <= high);
}
// return T/F
```

Selection Sort

- Main idea: take the correct (smallest) element and move it into the next place
- build the sorted array from L to R
- for each remaining unsorted portion to the right of position i, find the smallest element and swap it into position i



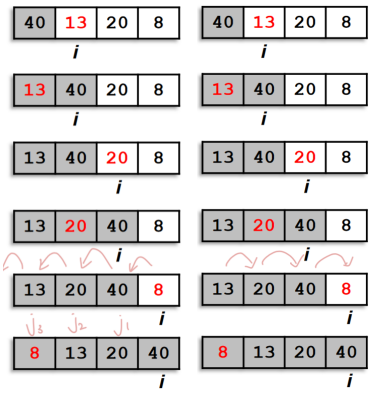

```
function swap(A, i, j) {
    let temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

function find_min_pos(A, low, high) {
    let min_pos = low;
    for (let j = low + 1; j <= high; j = j + 1) {
        if (A[j] < A[min_pos]) {
            min_pos = j;
        } else{}
    }
    return min_pos;
}

function selection_sort(A) {
    const len = array_length(A);
    for (let i= 0; i < len - 1; i = i+ 1) {
        let min_pos = find_min_pos(A, i, len-1);
        swap(A, i, min_pos);
    }
}
```

Insertion Sort

- Main idea: take the first element and move it into the correct place
- move a point i from L to R
- the array to the left of i is sorted
- swap the value at i with its neighbor to the left until the neighbor is smaller

```
function insertion_sort(A) {
    const len = array_length(A);
    for (let i = 1; i < len; i = i + 1) {
        let j = i - 1;
        while (j >= 0 && A[j] > A[j + 1]) {
            swap(A, j, j + 1);
            j = j - 1;
        }
    }
}

```

```
// Alternative Ver #2
// replaces the swaps by shifting elements right
function insertion_sort2(A) {
    const len = array_length(A);
    for (let i= 1; i < len; i = i+ 1) {
        const x = A[i];
        let j = i - 1;
        while (j >= 0 && A[j] > x) {
            // cannot rep w for loop cus A[j + 1] = x; is declared outside the while loop,
            // j cannot be accessed if the for loop does not include A[j + 1] = x;
            A[j + 1] = A[j]; // shift right
            j = j -1;
        }
        A[j + 1] = x;
    }
}

//Alternative Ver #3
function search_cond(A, cond) {
    const len = array_length(A);
    let i = 0;
    while (i < len && !con(A[i])) {
        i = i + 1;
    }
    return (i < len) ? i : -1;
    // return the pos of the 1st element that satisfies the cond
}

function insert(A, pos, x) {
    let j = array_length(A) - 1;
    while (j >= 0 && j >= pos) {
        A[j + 1] = A[j]; // shifts right
        j = j - 1;
    }
    A[pos] = x; // insert x into pos
}

function insertion_sort(A) {
    const B = [];
    const len = array_length(A);
    for (let i = len - 1; i >= 0; i = i - 1) {
        B[i] = A[i];
        insert(B, i - 1, search_cond(A, x => x < B[i]));
    }
    return B;
}
```

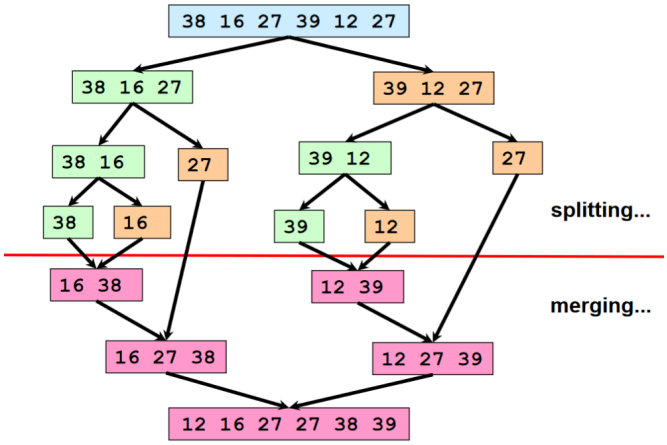


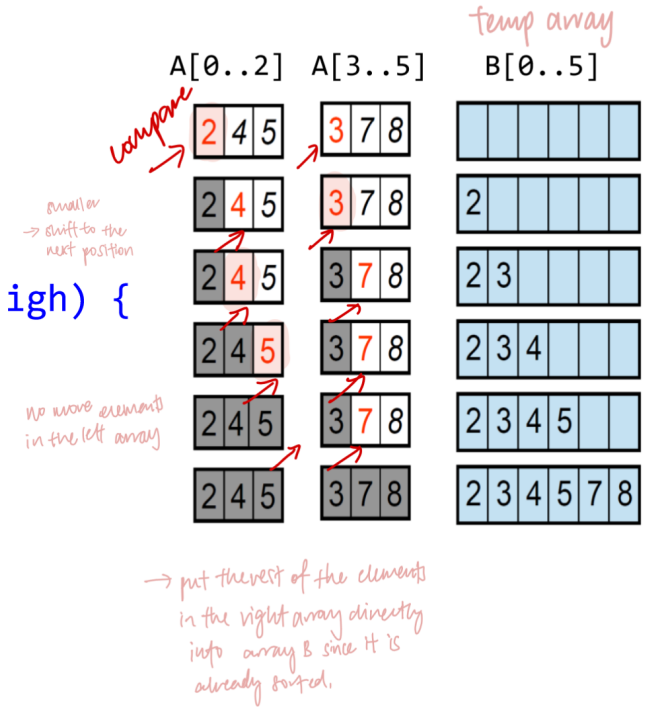
Merge Sort

- sort the halves => merge the halves (using temp arrays)

```
function merge_sort(A) {
    merge_sort_helper(A, 0, array_length(A) -1);
}

function merge_sort_helper(A, low, high) {
    if (low < high) {
        const mid = math_floor((low + high) / 2);
        merge_sort_helper(A, low, mid);
        merge_sort_helper(A, mid + 1, high);
        merge(A, low, mid, high);
    } else{ }
}

function merge(A, low, mid, high) {
    const B = []; // temporary array
    let left = low;
    let right = mid + 1;
    let Bidx = 0;

    while (left <= mid && right <= high) {
        if (A[left] <= A[right]) {
            B[Bidx] = A[left];
            left = left + 1;
        } else{
            B[Bidx] = A[right];
            right = right + 1;
        }
        Bidx = Bidx + 1;
    }

    while (left <= mid) {
        B[Bidx] = A[left];
        Bidx= Bidx+ 1;
        left = left + 1;
    }
    // right half is exhausted
    // no more elements in the right half
    // put the remaining elements in the left list into B directly

    while (right <= high) {
        B[Bidx] = A[right];
```

```
42          Bidx = Bidx + 1;
43          right = right + 1;
44      }
45      // left half is exhausted
46      // only one of the two while loops will be evaluated
47      // cannot have both halves being exhausted simultaneously
48
49      for (let k = 0; k < high -low + 1; k = k + 1) {
50          A[low + k] = B[k];
51      }
52      // [low + k] may not be [0]
53      // depends on where the half is split & merged
54  }
```





Memoization

- Reduces time complexity but increases space complexity (takes up more memory space due to storage of info)
- useful for recursive functions
- Idea: storing results that you have already computed somewhere for future use
    - Usually in a “local table”, which we can implement with an array

- The *first* call made to the function with a *certain set of arguments* will still incur a cost, but subsequent calls with the same input will return the remembered result rather than recalculating it, hence avoiding that cost

```
  const mem = [];
  function read(n, k) {
      return (mem[n] === undefined)
          ? undefined
          : mem[n][k];
  }

  function write(n, k, value) {
      if (mem[n] === undefined) {
          mem[n] = [];
      } else {}
      mem[n][k] = value;
  }
/*
mem must be accessible by all calls to that function,
so it must be declared in the global env
*/
  function memoised_fun(n, k) {
      if (n >= 0 && k >= 0 && read(n, k) !== undefined) {
          return read(n, k);
      } else {
          //calculate result normally
          if (n >= 0 && k >= 0) {
              write(n, k, result);
          } else { }
          return result;
      }
  }
/*
In the function, check if the result of this function call (i.e. with this
specific argument) has    already been computed before & is stored in mem

If yes, return the value stored in mem.

If not, compute the value, store it in mem, and return it.

The “local table” can be a 2-dimensional, 3-dimensional etc. array (i.e. 1
dimension for each parameter)

You need to check if each subsequent dimension is defined before you can
read / write -- e.g. mem[0][0] will throw you an error if mem[0] ===
undefined

To write to mem[0][0], you first need to declare mem[0] = [];
*/

  // Alternative Ver
function memoize(f) {
      const mem = []; // array mem serves as memory for alr computed results
of f
      function mf(x) {
          // test if f(x) has been computed alr
          if (mem[x] !== undefined) {
```

```
            return mem[x]; // just access memory
        } else {
            // compute f(x) and add result to mem
            const result = f(x);
            mem[x] = result;
            return result;
        }
    }
    return mf;
}
//reduces runtime of fib from exponential to n
```

Tribonacci

```
const mtrib = memoize(n => (n === 0)
                      ? 0
                      : (n === 1)
                          ? 1
                          : (n === 2)
                              ? 1
                              : mtrib(n -1) + mtrib(n -2) + mtrib(n -3));

mtrib(14);
//runtime = O(n)
```

Lec 10: Streams

Represent conditionals `E1 ? E2 : E3` using a function

```
function cond(x, y, z) {
    if (x) {
        return y();
    } else {
        return z();
    }
}

cond(E1, () => E2, () => E3);
```

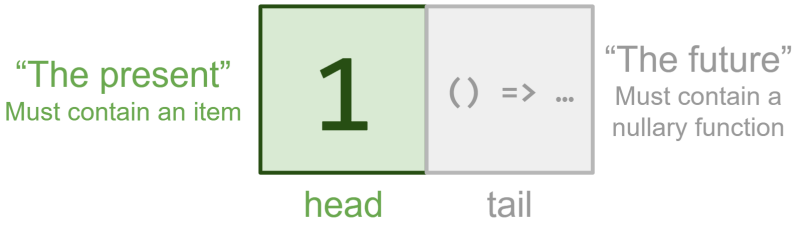
Delayed Evaluation:

- delay eval until we had enough info to decide which one is needed
- instrument of delay: functions allow us to describe an activity w/o actually doing it

Streams

- A stream is either **null**, or a pair whose **head** is a *data item* (as in a normal list) and whose **tail** is a nullary **function** that *returns a stream*
- Nullary function: function that takes no arguments
- When accessing elements in streams, remember that the tail is a function that needs to be applied / called with ()
- When returning a stream, wrap the tail in a function

```
head(tail(ones)())
head(tail(tail(ones)())());
//i.e. use stream_tail instead of tail
```




```
pair(
    1,
    () => // returns a stream
)
```

```
function stream_filter(p, s) {    <- returns a stream
   return is_null(s)
      ? null
      : p(head(s))
         ? pair(head(s),
               () => stream_filter(p, stream_tail(s)))
         : stream_filter(p, stream_tail(s));
}
```

Since stream_filter returns a stream with the head being evaluated, you need to wrap it in a nullary function to delay evaluation since it is the stream tail here

```
function stream_filter(p, s) {    <- returns a stream
   return is_null(s)
      ? null
      : p(head(s))
         ? pair(head(s),
               () => stream_filter(p, stream_tail(s)))
         : stream_filter(p, stream_tail(s));
}
```

On the other hand, this part is not the stream tail, so no need to wrap in nullary function

Memoized Streams

```
function memo_fun(fun) {
    let already_run = false;
    let result = undefined;
    function mfun() {
        if (!already_run) {
            result = fun();
            already_run = true;
            return result;
        } else {
            return result;
        }
```

```
    }
    return mfun;
}

function ms(m, s) {
    display(m);
    return s;
}

const onesA = pair(1, () => ms("A", onesA));

stream_ref(onesA, 3);
/* Output:
"A"
"A"
"A"
1 */

const onesB = pair(1, memo_fun(() => ms("B", onesB)));
stream_ref(onesB, 3); // "B" 1

-> stream_ref(display("B"); memoize onesB; return onesB, 2)
-> stream_ref(return memoized onesB, 1)
-> stream_ref(return memoized onesB, 0)
-> head(onesB)
-> 1

function m_integers_from(n) {
    return pair(n, memo_fun(() => ms("M: " + stringify(n), m_integers_from(n + 1))));
}
const m_integers = m_integers_from(1);
stream_ref(m_integers, 0); // 1
stream_ref(m_integers, 2);
// "M: 1"
// "M: 2"
// 3
stream_ref(m_integers, 5);
/* Output:
"M: 3"
"M: 4"
"M: 5"
6 */ //1-2 has been memoized => will not be displayed anyth
stream_ref(m_integers, 5); // only output 6
```

Stream Map

```
function stream_map(f, s) {
    return is_null(s)
        ? null
        : pair(f(head(s)),() => stream_map(f, stream_tail(s)));
}

    const x = stream_map(display, enum_stream(0, 10));
==> const x = stream_map(display, pair(0, () => enum_stream(1, 10)));
==> const x = pair(display(0), () => stream_map(display, enum_stream(1, 10)));
```

```
    // display(0) prints 0 and returns 0
==> const x = pair(0, () => stream_map(display, enum_stream(1, 10)));


    stream_ref(x, 3);
==> stream_ref(pair(0, () => stream_map(display, enum_stream(1, 10))), 3);
==> stream_ref(stream_map(display, enum_stream(1, 10)), 2);
==> stream_ref(stream_map(display, pair(1, () => enum_stream(2, 10))), 2);
==> stream_ref(pair(display(1), () => stream_map(display, enum_stream(2,
10))), 2);
    // display(1) prints 1 and returns 1, step 68
==> stream_ref(pair(1, () => stream_map(display, enum_stream(2, 10))), 2);
==> stream_ref(stream_map(display, enum_stream(2, 10)), 1);
==> stream_ref(stream_map(display, pair(2, () => enum_stream(3, 10))), 1);
==> stream_ref(pair(display(2), () => stream_map(display, enum_stream(3,
10))), 1);
    // display(2) prints 2 and returns 2, step 110
==> stream_ref(pair(2, () => stream_map(display, enum_stream(3, 10))), 1);
==> stream_ref(stream_map(display, enum_stream(3, 10)), 0);
==> stream_ref(stream_map(display, pair(3, () => enum_stream(4, 10))), 0);
==> stream_ref(pair(display(3), () => stream_map(display, enum_stream(4,
10))), 0);
    // display(3) prints 3 and returns 3, step 152
==> stream_ref(pair(3, () => stream_map(display, enum_stream(4, 10))), 0);
==> head(pair(3, () => stream_map(display, enum_stream(4, 10))));
==> 3


    stream_ref(x, 5);
==> stream_ref(pair(0, () => stream_map(display, enum_stream(1, 10))), 5);
...
==> stream_ref(pair(display(5), () => stream_map(display, enum_stream(6,
10))), 0);
    // display(5) prints 5 and returns 5, step 372
==> stream_ref(pair(5, () => stream_map(display, enum_stream(6, 10))), 0);
==> head(pair(5, () => stream_map(display, enum_stream(6, 10))));
==> 5
```

Lec 11: Metacircular Evaluator

- Essential idea: I have a string that represents a program, and now I want to get a return value out of it

Representations

- **Literal -** Just return the primitive value!
- **Names -** Find the primitive value or compound structure assigned to this name
- **Function Applications -** Get the function name and recursively evaluate it to get the function itself. Call the apply function with the function definition and the arguments supplied!

- **Operator Combination -** Convert it into a function application, with the operator as the name!
- **Conditional -** Recursively evaluate the predicate, then recursively evaluate the required result
- **Lambda Expressions -** Make a compound function from the inputs
- **Sequences -** Evaluate each statement one by one
- **Block -** Update the environment and frames! Block and sequence are separate
- **Return Statements -** Returns a return value! Only occurs when apply calls evaluate, so handling of presence of return value is done by apply.
- **Assignment -** Reassign variable name to new value, name must exist in environment
- **Function Declaration -** Converts into lambda and constant declaration and recursively evaluate
- **Constant and Variable Declarations -** Recursively evaluate value and assign to the given symbol
- **Errors -** Thrown when all else fails

Name	Type	e.g.
Input	string	“const f = x => x + 1; f(2+3);”)
Program	Tagged list Form: pair(“tag”, whatever the tag is supposed to be)	[“sequence”, [[“constant_declaration”, [[“name”, “f’], [“lambda_expression”, … Note: ALL STRINGS, still strings at this point
Block	List of two elements • The tag “block” • A tagged list	
Frame	A list of two elements: • list of all symbols/keys, • list of all its values	[[“x”, “y”], [3, 4]]
Empty environment	null	
Normal environments	either null or a list of two elements • A frame (basically just a list of two lists) • Base environment (which is just another list of two elements, or null)	An environment that only contains “const x = 3; const y = 4;” [[[“x”, “y”], [3, 4]], list_from_base_env]
Literal values	Value could be • Number, Boolean, string, null	[“literal”, value]
Name (non-literal values)	• symbol is a String that constitutes the name • you can lookup the associated value of a symbol in the environment	[“name”, symbol]
Primitive functions	the lambda function is an *actual function*, not a string!	[“primitive”, lambda function]
Function application	list of three elements: • the tag, • function expression (which is a tagged list too), • list (not tagged!!) of argument expressions	[“application”, function expression, arg_list]
Return statement	Return can happen anywhere in the code • special behaviour to terminate program when return statement is reached	[“return statement”, return_expression] Example: return x + 1; [“return_statement”, [“operator_combination”, [“+”, [“name”, x], [“literal”, 1]]]

Application of a named function: [“application”, function name, list of args]

```
[ “application”,
  [ “name”, “name_of_function_as_a_string” ],
[ [ “literal”, 3],
    [ “literal”, 5],                                  } List of arguments
    [ “operator_combination”, [...] ] ]]
```

Application of a lambda function: [“application”, lambda exp, list of args]

```
[ “application”,
[ “lambda_expression”,
    [ [“name”, “x”], [“name”, “y”], [“name”, “z”] ],
    [ [ “return statement”, [...]]],
  [ [ “literal”, 3], [literal”, 5], ... ]]      ← List of arguments
```

Navigate through MCE

- Ctrl + F and Ctrl + L
- Alt + 0 to collapse all functions
- try to think of the desired outcome in terms of the environment model
- Sometimes, it’s also about changing what information is stored in the frames and changing the environments

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          : is_name(component)
          ? lookup_symbol_value(symbol_of_name(component), env)
          : is_application(component)
          ? apply(evaluate(function_expression(component), env),
                  list_of_values(arg_expressions(component), env))
          : is_operator_combination(component)
          ? evaluate(operator_combination_to_application(component), env)
          : is_conditional(component)
          ? eval_conditional(component, env)
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          : is_sequence(component)
          ? eval_sequence(sequence_statements(component), env)
          : is_block(component)
          ? eval_block(component, env)
          : is_return_statement(component)
          ? eval_return_statement(component, env)
          : is_assignment(component)
          ? eval_assignment(component, env)
          : is_function_declaration(component)
          ? evaluate(function_decl_to_constant_decl(component), env)
          : is_declaration(component)
          ? eval_declaration(component, env)
          : error(component, "Unknown syntax -- evaluate");
}

function apply(fun, args) {
   if (is_primitive_function(fun)) {
      return apply_primitive_function(fun, args);
   } else if (is_compound_function(fun)) {
      const result = evaluate(function_body(fun),
                              extend_environment(
```

```
                                    function_parameters(fun),
                                    args,
                                    function_environment(fun)));
      return is_return_value(result)
             ? return_value_content(result)
             : undefined;
   } else {
      error(fun, "Unknown function type -- apply");
   }
}

function list_of_values(exps, env) {
     return map(arg => evaluate(arg, env), exps);
}

function eval_conditional(component, env) {
    return is_truthy(evaluate(conditional_predicate(component), env))
           ? evaluate(conditional_consequent(component), env)
           : evaluate(conditional_alternative(component), env);
}

function eval_sequence(stmts, env) {
    if (is_empty_sequence(stmts)) {
        return undefined;
    } else if (is_last_statement(stmts)) {
        return evaluate(first_statement(stmts),env);
    } else {
        const first_stmt_value =
            evaluate(first_statement(stmts),env);
        if (is_return_value(first_stmt_value)) {
            return first_stmt_value;
        } else {
            return eval_sequence(
                rest_statements(stmts),env);
        }
    }
}

function list_of_unassigned(names) {
    return map(name => "*unassigned*", names);
}

function scan_out_declarations(component) {
    return is_sequence(component)
           ? accumulate(
                 append,
                 null,
                 map(scan_out_declarations,
                     sequence_statements(component)))
           : is_declaration(component)
           ? list(declaration_symbol(component))
           : null;
}

function eval_block(component, env) {
    const body = block_body(component);
    const locals = scan_out_declarations(body);
    const unassigneds = list_of_unassigned(locals);
```

```
    return evaluate(body, extend_environment(locals,
                                             unassigneds,
                                             env));
}

function eval_return_statement(component, env) {
    return make_return_value(
                evaluate(return_expression(component), env));
}

function eval_assignment(component, env) {
    const value = evaluate(assignment_value_expression(component), env);
    assign_symbol_value(assignment_symbol(component), value, env);
    return value;
}

function eval_declaration(component, env) {
    assign_symbol_value(declaration_symbol(component),
                        evaluate(declaration_value_expression(component),
                                 env),
                        env);
    return undefined;
}

// functions from SICP JS 4.1.2

function is_literal(component) {
    return is_tagged_list(component, "literal");
}
function literal_value(component) {
    return head(tail(component));
}

function is_tagged_list(component, the_tag) {
    return is_pair(component) && head(component) === the_tag;
}

function is_name(component) {
    return is_tagged_list(component, "name");
}

function make_name(symbol) {
    return list("name", symbol);
}

function symbol_of_name(component) {
    return head(tail(component));
}

function is_assignment(component) {
    return is_tagged_list(component, "assignment");
}
function assignment_symbol(component) {
    return head(tail(head(tail(component))));
}
function assignment_value_expression(component) {
    return head(tail(tail(component)));
}
```

```
153
154 function is_declaration(component) {
155     return is_tagged_list(component, "constant_declaration") ||
156            is_tagged_list(component, "variable_declaration") ||
157            is_tagged_list(component, "function_declaration");
158 }
159 function declaration_symbol(component) {
160    return head(tail(head(tail(component))));
161 }
162 function declaration_value_expression(component) {
163    return head(tail(tail(component)));
164 }
165
166 function make_constant_declaration(name, value_expression) {
167     return list("constant_declaration", name, value_expression);
168 }
169
170 function is_lambda_expression(component) {
171    return is_tagged_list(component, "lambda_expression");
172 }
173 function lambda_parameter_symbols(component) {
174    return map(symbol_of_name, head(tail(component)));
175 }
176 function lambda_body(component) {
177    return head(tail(tail(component)));
178 }
179
180 function make_lambda_expression(parameters, body) {
181     return list("lambda_expression", parameters, body);
182 }
183
184 function is_function_declaration(component) {
185     return is_tagged_list(component, "function_declaration");
186 }
187 function function_declaration_name(component) {
188     return list_ref(component, 1);
189 }
190 function function_declaration_parameters(component) {
191     return list_ref(component, 2);
192 }
193 function function_declaration_body(component) {
194     return list_ref(component, 3);
195 }
196 function function_decl_to_constant_decl(component) {
197     return make_constant_declaration(
198                function_declaration_name(component),
199                make_lambda_expression(
200                    function_declaration_parameters(component),
201                    function_declaration_body(component)));
202 }
203
204 function is_return_statement(component) {
205    return is_tagged_list(component, "return_statement");
206 }
207 function return_expression(component) {
208    return head(tail(component));
209 }
210
```

```
function is_conditional(component) {
    return is_tagged_list(component, "conditional_expression") ||
           is_tagged_list(component, "conditional_statement");
}
function conditional_predicate(component) {
   return list_ref(component, 1);
}
function conditional_consequent(component) {
   return list_ref(component, 2);
}
function conditional_alternative(component) {
   return list_ref(component, 3);
}

function is_sequence(stmt) {
   return is_tagged_list(stmt, "sequence");
}
function make_sequence(stmts) {
   return list("sequence", stmts);
}
function sequence_statements(stmt) {
   return head(tail(stmt));
}
function first_statement(stmts) {
   return head(stmts);
}
function rest_statements(stmts) {
   return tail(stmts);
}
function is_empty_sequence(stmts) {
   return is_null(stmts);
}
function is_last_statement(stmts) {
   return is_null(tail(stmts));
}

function is_block(component) {
    return is_tagged_list(component, "block");
}
function block_body(component) {
    return head(tail(component));
}
function make_block(statement) {
    return list("block", statement);
}

function is_operator_combination(component) {
    return is_tagged_list(component, "operator_combination");
}
function operator_combination_operator_symbol(component) {
    return list_ref(component, 1);
}
function operator_combination_first_operand(component) {
    return list_ref(component, 2);
}
function operator_combination_second_operand(component) {
    return list_ref(component, 3);
}
```

```
269
270 function make_application(function_expression, argument_expressions) {
271     return list("application", function_expression, argument_expressions);
272 }
273
274 function operator_combination_to_application(component) {
275     const operator = operator_combination_operator_symbol(component);
276     return operator === "!" || operator === "-unary"
277            ? make_application(
278                  make_name(operator),
279                  list(operator_combination_first_operand(component)))
280            : make_application(
281                  make_name(operator),
282                  list(operator_combination_first_operand(component),
283                       operator_combination_second_operand(component)));
284 }
285
286 function is_application(component) {
287    return is_tagged_list(component, "application");
288 }
289 function function_expression(component) {
290    return head(tail(component));
291 }
292 function arg_expressions(component) {
293    return head(tail(tail(component)));
294 }
295
296 // functions from SICP JS 4.1.3
297
298 function is_truthy(x) {
299     return is_boolean(x)
300            ? x
301            : error(x, "boolean expected, received:");
302 }
303
304 function make_function(parameters, body, env) {
305     return list("compound_function",
306                 parameters, body, env);
307 }
308 function is_compound_function(f) {
309     return is_tagged_list(f, "compound_function");
310 }
311 function function_parameters(f) {
312     return list_ref(f, 1);
313 }
314 function function_body(f) {
315     return list_ref(f, 2);
316 }
317 function function_environment(f) {
318     return list_ref(f, 3);
319 }
320
321 function make_return_value(content) {
322     return list("return_value", content);
323 }
324 function is_return_value(value) {
325     return is_tagged_list(value, "return_value");
326 }
```

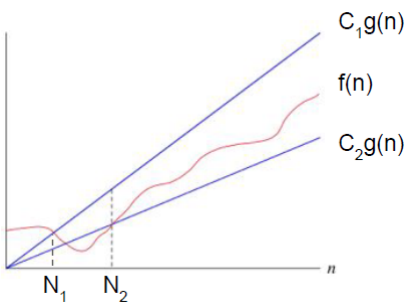
```
function return_value_content(value) {
    return head(tail(value));
}

function enclosing_environment(env) {
    return tail(env);
}
function first_frame(env) {
    return head(env);
}
const the_empty_environment = null;

function make_frame(symbols, values) {
    return pair(symbols, values);
}
function frame_symbols(frame) {
    return head(frame);
}
function frame_values(frame) {
    return tail(frame);
}

function extend_environment(symbols, vals, base_env) {
    //if (is_null(symbols)) {

    //} else {
    //    frame_counter = frame_counter + 1;
    //}
    return length(symbols) === length(vals)
               ? pair(make_frame(symbols, vals), base_env)
               : length(symbols) < length(vals)
                 ? error("Too many arguments supplied: " +
                         stringify(symbols) + ", " +
                         stringify(vals))
                 : error("Too few arguments supplied: " +
                         stringify(symbols) + ", " +
                         stringify(vals));
}

function lookup_symbol_value(symbol, env) {
    function env_loop(env) {
        function scan(symbols, vals) {
            return is_null(symbols)
                   ? env_loop(
                       enclosing_environment(env))
                   : symbol === head(symbols)
                     ? head(vals)
                     : scan(tail(symbols), tail(vals));
        }
        if (env === the_empty_environment) {
            error(symbol, "Unbound name");
        } else {
            const frame = first_frame(env);
            return scan(frame_symbols(frame),
                        frame_values(frame));
        }
    }
    return env_loop(env);
```

```
}

function assign_symbol_value(symbol, val, env) {
    function env_loop(env) {
        function scan(symbols, vals) {
            return is_null(symbols)
                ? env_loop(
                    enclosing_environment(env))
                : symbol === head(symbols)
                  ? set_head(vals, val)
                  : scan(tail(symbols), tail(vals));
        }
        if (env === the_empty_environment) {
            error(symbol, "Unbound name -- assignment");
        } else {
            const frame = first_frame(env);
            return scan(frame_symbols(frame),
                        frame_values(frame));
        }
    }
    return env_loop(env);
}

// functions from SICP JS 4.1.4

function is_primitive_function(fun) {
   return is_tagged_list(fun, "primitive");
}
function primitive_implementation(fun) {
   return head(tail(fun));
}

const primitive_functions = list(
       list("head",    head             ),
       list("tail",    tail             ),
       list("pair",    pair             ),
       list("list",    list             ),
       list("is_null", is_null          ),
       list("display", display          ),
       list("error",   error            ),
       list("math_abs",math_abs         ),
       list("+",       (x, y) => x + y  ),
       list("-",       (x, y) => x - y  ),
       list("-unary",   x     =>   - x  ),
       list("*",       (x, y) => x * y  ),
       list("/",       (x, y) => x / y  ),
       list("%",       (x, y) => x % y  ),
       list("===",     (x, y) => x === y),
       list("!==",     (x, y) => x !== y),
       list("<",       (x, y) => x <   y),
       list("<=",      (x, y) => x <=  y),
       list(">",       (x, y) => x >   y),
       list(">=",      (x, y) => x >=  y),
       list("!",        x     =>   !   x)
       );
const primitive_function_symbols =
        map(head, primitive_functions);
const primitive_function_objects =
```

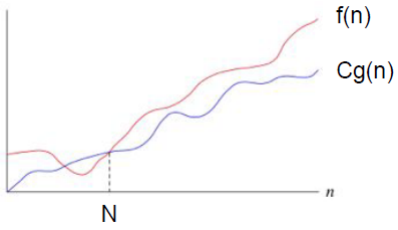
```
        map(fun => list("primitive", head(tail(fun))),
            primitive_functions);

const primitive_constants = list(list("undefined", undefined),
                                 list("Infinity",  Infinity),
                                 list("math_PI",   math_PI),
                                 list("math_E",    math_E),
                                 list("NaN",       NaN)
                                );
const primitive_constant_symbols =
        map(c => head(c), primitive_constants);
const primitive_constant_values =
        map(c => head(tail(c)), primitive_constants);

function apply_primitive_function(fun, arglist) {
    return apply_in_underlying_javascript(
                primitive_implementation(fun),
                arglist);
}

function setup_environment() {
    return extend_environment(
                append(primitive_function_symbols,
                       primitive_constant_symbols),
                append(primitive_function_objects,
                       primitive_constant_values),
                the_empty_environment);
}

let the_global_environment = setup_environment();

// convenient function to deal with the top
// level:
// * parse input
// * wrap program in a block
// * evaluate block in global environment
function parse_and_evaluate(input) {
    const program = parse(input);
    const implicit_top_level_block = make_block(program);
    return evaluate(implicit_top_level_block,
                    the_global_environment);
}
```

Brief 2: Order of Growth

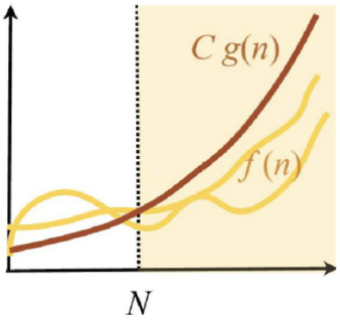
- Big-Θ: Asymptotic upper and lower bound



- Big-Ω: Asymptotic lower bound



- Big-O: Asymptotic upper bound



- The big-O, big-omega and big-theta can be calculated for all cases (best, worst, average), i.e. the bounds are independent of the type of case.
- Time complexity: no. of steps (is never exact)
- Space complexity: no. of deferred operations (Θ(1) for iterative processes)
    - Generally interested in the worst case

> Assume a resource function *(any function in terms of n, e.g. r(n) = 2n+1)* r(n).
>
> - r(n) has order of growth of Θ(r(n)), Ω(r(n)), O(r(n)).
>
> Assume a resource function r(n) with order of growth of Θ(g(n)).
>
> r(n) has order of growth of both Ω(g(n)), O(g(n))
>
> If f(n) = O(n^2), then f(n) = O(n^3)
>
> $$O(n^2) \subseteq O(n^3) \quad (1)$$
>
> $$f(n) \in O(n^2), then f(n) \in O(n^3) \quad (2)$$
>
> If f(n) = Ω(n^2), then f(n) = Ω(n)
>
> $$\Omega(n^2) \subseteq \Omega(n) \quad (3)$$
>
> $$f(n) \in \Omega(n^2), then f(n) \in \Omega(n). \quad (4)$$

> How to do better than Ω(n^2)?
>
> - Come up with a O(k) algorithm, where k is "lower in complexity" than n^2, i.e. less than n2 asymptotically

    - O(n log n)
    - O(n)
- Incorrect examples:
    - Ω(n log n): upper bound may not be better
    - O(n^2): upper bound may not be better
    - Θ(n^2): upper bound may not be better

Brief 5: T-diagrams, Interpreters & Compilers

Interpreters

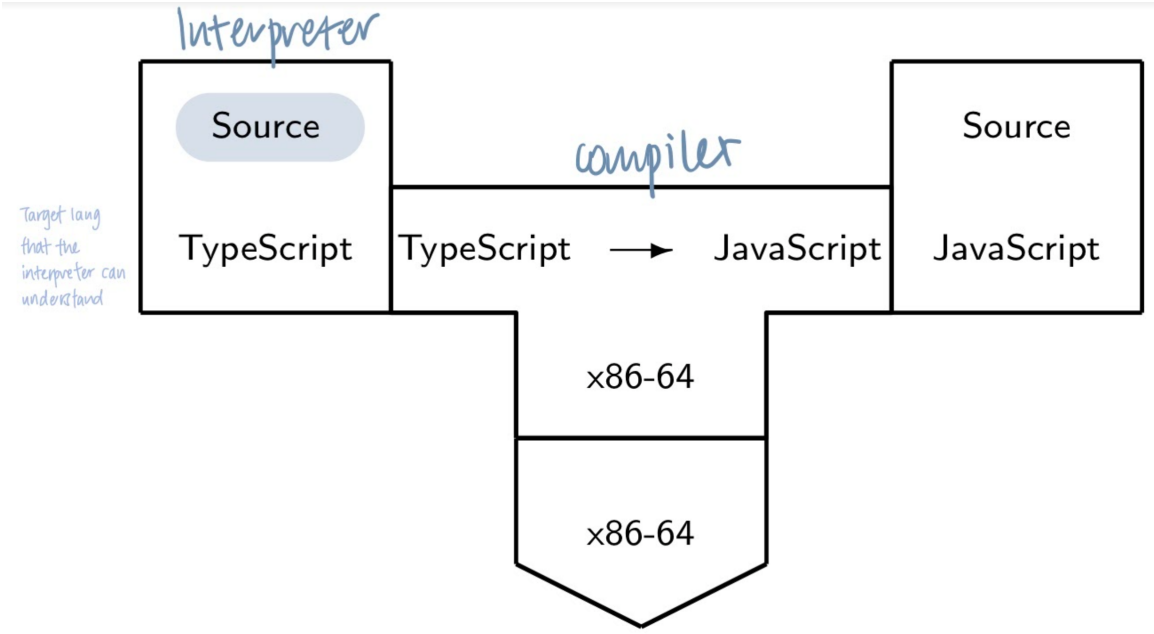
- a program that executes the written program

Compilers

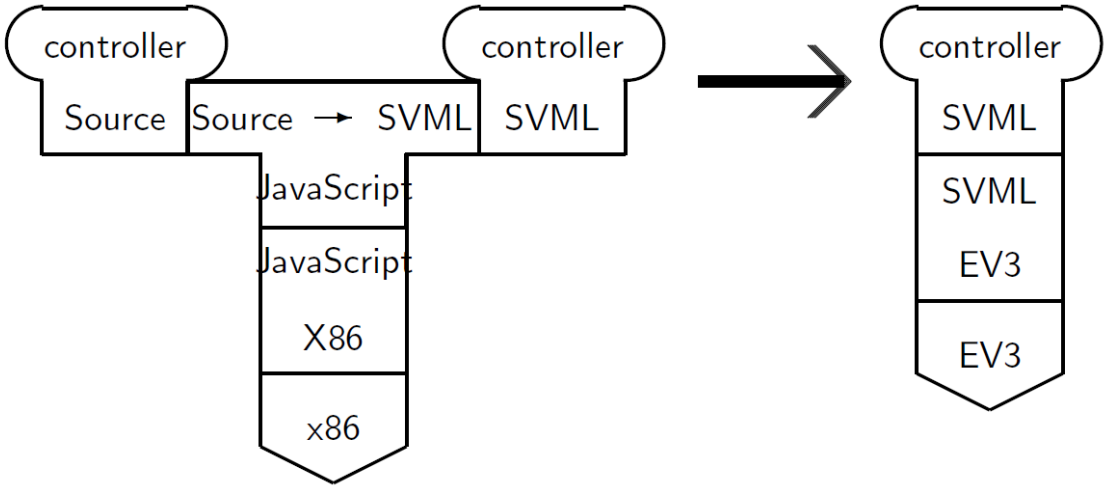
- a program that translates the written program's language to a language that the computer (interpreter) can evaluate

T-diagram

- L to R: Compiler
- Top to bottom: Interpreter



Overview: Source for Robotics



Compiling "controller" from Source to SVML on a PC, and then running the SVML program on the EV3 brick.

Brief 8: Loops

While Loop

- evaluates the condition expression
  - if true, executes the body statement of the loop, after which the process repeats
  - if false, the loop terminates
- return statement are not allowed in the bodies of while and for loops

```
function factorial_iter(n) {
    function f(acc, k) {
        if (k <= n) {
            return f(acc * k, k + 1);
        } else {
            return acc;
        }
    }
    return f(1, 1);
}

function factorial_while(n) {
    let acc = 1;
    let k = 1;
    while (k <= n) {
        acc = acc * k;
        k = k + 1;
    }
}

function factorial_for(n) {
    let acc = 1;
    for (let k = 1; k <= n; k = k + 1) {
        acc = acc * k;
    }
    return acc;
```

```
}
```

For Loop

- for loop is not tested for env model but while loop is tested
- the declared loop control var cannot be assignment to in the body
- all 3 components in the header of a for loop are compulsory
- ***break;***
  - terminates the current execution of the loop and also terminates the entire loop

```
for (let i= 0; i< 5; i= i+ 1) {
    display(stringify(i) + " here");
    if (i === 2) {
        break;
    } else{ }
    display(stringify(i) + " there");
}
display("OK");
/* Output:
"0 here"
"0 there"
"1 here"
"1 there"
"2 here"
"OK"
*/
```

- ***continue;***
  - terminates the current execution of the loop and continues with the loop

```
for (let i = 0; i < 5; i = i + 1) {
    display(stringify(i) + " here");
    if (i === 2) {
        continue;
    } else{ }
    display(stringify(i) + " there");
}
display("OK");
/* Output;
"0 here"
"0 there"
"1 here"
"1 there"
"2 here"
"3 here"
"3 there"
"4 here"
"4 there"
"OK"
*/
```

Syntax	Equivalent to

for (statement 1; expression; statement 2) {
body statement;
}
	{
	statement 1;
	while
	(expression) {
	body statement;
	assignment;
	}
	}

statement 1 can only be an assignment statement or a variable declaration statement e.g. let x = 1; the var is called a loop control var	

Recursive to Iterative Process

Order does not matter

```
// Recursive
function factorial(n) {
    return n === 1
        ? 1
        : n * factorial(n-1);
}

// Iterative
function fact_iter(n) {
    function helper(i, acc) {
        return i === 1 ? acc : helper(i - 1, i * acc);
    } return helper(n, 1);
}
```

Order Matters

- reverse the final list

```
// Recursive
function map(f, xs) {
    return is_null(xs)
        ? null
        : pair(f(head(xs)), map(f, tail(xs)));
}

// Iterative
function map_iter(f, xs) {
    function helper(ys, acc) {
        return is_null(ys)
            ? acc
            : helper(tail(ys), pair(f(head(ys)), acc));
    }
    return helper(reverse(xs), null);
}
```

CPS