CS1101S Cheatsheet 2017

by ning

Basic Syntax and Built-in Functions

```
for (var i = 0; i < 5; i = i+1) {
    // do something
}

while (i < 10) {
    // do something
}

// join many lists together
accumulate(append, [], xxs);

// clone a list
map(function(x) { return x; }, xs);

// clone an object
JSON.parse(JSON.stringify(obj));
```

- `parseInt(string)`: Interprets the given `string` as an integer, and returns that integer
- `math_pi`: Refers to the number π, the mathematical constant
- `math_max(a, b, c, ...)`: Returns the largest argument given
- `math_floor(x)`: Returns the largest integer smaller than `x`
- `math_ceil(x)`: Returns the smallest integer larger than `x`
- `math_sqrt(x)`: Refers to the square root function, returns $\sqrt{x}$
- `math_pow(base, exponent)`: Returns `base` to the power of `exponent`, i.e. b^e
- `is_number(x)`, `is_boolean(x)`, `is_string(x)`: Returns `true` ifx is the respective primitive data type, else `false`
- `is_function(x)`: Returns `true` if `x` is a function, else `false`
- `is_object(x)`: Returns `true` if `x` is an object, else `false`. Note that functions and arrays are objects
- `is_array(x)`: Returns `true` if `x` is an array, else `false`. Note that the empty array `[]`, also known as the empty list, is an array
- `equal(x, y)`: Returns `true` if `x` and `y` have the same strucutre, and corresponding leaves are `===`, else `false`
- `array_length(x)`: Returns the current length of array `x`
- `apply_in_underlying_javascript(f, xs)`: calls the function `f` with arguments `xs`

Substitution Model

```
function plus_one(x) {
    return x + 1;
}

function twice(f) {
    return function(x) {
        return f(f(x));
    }
}

function n_times(f, n) {
    if (n === 1) {
        return f;
    } else {
        return function(x) {
            return f((n_times(f, n-1))(x);
        }
    }
}

function chain(f, n) {
    if (n === 1) {
        return f;
    } else {
        return (chain(f, n-1))(f);
    }
}
```

`plus_one(plus_one(0))`
$pp(0) \to p(1) \to 2$

`(twice(plus_one))(0)`
$t(p)0 \to pp(0) \to 2$

`(n_times(plus_one,4))(0)`
$n(p,4)(0) \to p(n(p,3))(0) \to pp(n(p,2))(0)$
$\to ppp(n(p,1))(0) \to pppp(0) \to 4$

`((n_times(twice,3))(plus_one))(0)`
$n(t,3)(p)(0) \to ttt(p)(0) \to ttpp(0) \to tpppp(0)$
$\to pppppppp(0) \to p^8(0) \to 8$

`((n_times(chain(twice,3),2))(plus_one))(0)`
$n(c(t,3),2)(p)(0) \to n(c(t,2)(t),2)(p)(0)$
$\to n(c(t,1)(tt),2)(p)(0) \to n(t(tt),2)(p)(0)$
$\to n(tttt,2)(p)(0) \to t^8(p)(0) \to p^{2^8}(0)$
$\to p^{256}(0) \to 256$

`((chain(twice,4))(plus_one))(0)`
$((c(t,4))(p))(0) \to c(t,3)(t)(p)(0) \to c(t,2)(tt)(p)(0)$
$\to c(t,1)(tttt)(p)(0) \to tttttttt(p)(0) \to t^8 p(0)$
$\to p^{2^8}(0) \to p^{256} \to 256$

Environment Model

To apply a function f,

1. Create a new frame, A
2. Point A back to the environment in which f was defined
3. In A, bind the parameters of f to the values of the arguments given during the function call
4. Evaluate the body of f within A

When checking through the workings, **make sure the frames all point back to some other frame**.

Lists: a list is either the empty list `[]` or a pair whose tail is a list.

- `pair(x, y)`: Makes a pair from x and y.
- `is_pair(x)`: Returns true if x is a pair and false otherwise.
- `head(x)`: Returns the head (first component) of the pair x.
- `tail(x)`: Returns the tail (second component) of the pair x.
- `set_head(p, x)`: Sets the head (first component) of the pair p to be x; returns undefined.
- `set_tail(p, x)`: Sets the tail (second component) of the pair p to be x; returns undefined.
- `is_empty_list(xs)`: Returns true if xs is the empty list, and false otherwise.
- `is_list(x)`: Returns true if x is a list as defined in the lectures, and false otherwise. Iterative process; time: O(n), space: O(1), where n is the length of the chain of tail operations that can be applied to x.
- `list(x1, x2,..., xn)`: Returns a list with n elements. The first element is x1, the second x2, etc. Iterative process; time: O(n), space: O(n), since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `length(xs)`: Returns the length of the list xs. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `map(f, xs)`: Returns a list that results from list xs by element-wise application of f. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `build_list(n, f)`: Makes a list with n elements by applying the unary function f to the numbers 0 to n - 1. Recursive process; time: O(n), space: O(n).
- `for_each(f, xs)`: Applies f to every element of the list xs, and then returns true. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `list_to_string(xs)`: Returns a string that represents list xs using the text-based boxandpointer notation [...].
- `reverse(xs)`: Returns list xs in reverse order. Iterative process; time: O(n), space: O(n), where n is the length of xs. The process is iterative, but consumes space O(n) because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list ys to the list xs. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to x (===); returns [] if the element does not occur in the list. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `remove(x, xs)`: Returns a list that results from xs by removing the first item from xs that is identical (===) to x. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `remove_all(x, xs)`: Returns a list that results from xs by removing all items from xs that are identical (===) to x. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the oneargument function pred returns true. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from start using a step size of 1, until the number exceeds (¿) end. Recursive process; time: O(n), space: O(n), where n is the length of xs.
- `list_ref(xs, n)`: Returns the element of list xs at position n, where the first element has index 0. Iterative process; time: O(n), space: O(1), where n is the length of xs.
- `accumulate(op, initial, xs)`: Applies binary function op to the elements of xs from right-to-left order, first applying op to the last element and the value initial, resulting in r1, then to the second-last element and r1, resulting in r2, etc, and finally to the first element and rn1, where n is the length of the list. Thus, accumulate(op,zero,list(1,2,3)) results in op(1, op(2, op(3, zero))). Recursive process; time: O(n), space: O(n), where n is the length of xs, assuming op takes constant time.

Objects

```
var obj = {'aa': 4, 'bb': true,
    'cc': function(x) { return x * x; } };
obj['aa'] === obj.aa // true
obj['bb'] === obj.bb // true
obj['cc'] === obj.cc // true
obj['cc'](5) === obj.cc(5) // true
```

Pseudo-classical Inheritance

The `new` keyword,

1. Constructs an empty object
2. Calls the constructor function with that object as `this`
3. Returns the object

Class methods can be added within the constructor function, or dynamically by accessing the contructor's `prototype`.

```
function MyClass() {
    this.my_method = function() {
        // do something
    };
}

MyClass.prototype.another_method =
    function() {
        // do something else
    };

function OtherClass() { }

MyClass.Inherits(OtherClass);
```

Inheritance is created using `Inherits`, which sets the `__proto__` of `MyClass` to `ParentClass.prototype`.

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

```
var tree = list(list(1, 2), list(3, 4));

function count_data_items(tree) {
    return is_empty_list(tree)
        ? 0
        : (is_list(head(tree))
            ? count_data_items(head(tree))
            : 1)
        + count_data_items(tail(tree));
}

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

Besides the base case, these operations consider two cases. One, when the element is itself a tree, and another when it is not.

Binary Search Trees: A binary tree is the empty list or a list with three element, whose first element is a binary tree, whose second element is a data item, and whose third element is a binary tree.

The first element is called the left subtree and the third element is called the right subtree of the binary tree. The second element is called the value of the binary tree.

A binary *search* tree is a binary tree where all data items in the left subtree are smaller than its value and all data item in the right subtree are large than its value.

```
function find(bst, name) {
    if (is_empty_binary_tree(bst)) {
        return false;
    } else if (name < value_of(bst)) {
        return find(left_subtree_of(bst), name);
    } else if (value_of(bst) < name) {
        return find(right_subtree_of(bst), name);
    } else {
        return true;
    }
}
```

A balanced tree is a tree whose leaves have depths that differ by at most one.

Wishful Thinking is a critical technique in functional programming. Mastery of this technique, in my opinion, guarantees you an A.

1. Divide the solution into distinct steps
2. Find a repeating pattern in the solution steps that can be abstracted
3. Implement the first step in a function f
4. Use the function f within itself recursively to complete the rest of the solution steps

```
function append(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else {
        return pair(head(xs), append(tail(xs), ys));
    }
}
```

The `append` implementation above takes the first element of `xs`, then pairs it with the next element from `xs`, unless `xs` is the empty list—then it pairs with all of `ys`. The repeating step is the parring of an element with the next. The implementation of this step is with the built-in function `pair`.

Memoization

```
function mfib(n) {
    var mem = [];
    function fib(k) {
        if (mem[k] !== undefined) {
            return mem[k];
        } else {
            var result = (k <= 1)
                ? k
                : fib(k - 1) + fib(k - 2);
            mem[k] = result;
            return result;
        }
    }
    return fib(n);
}
```

Permutations & Combinations

```
function permutations(s) {
    if (is_empty_list(s)) {
        return list([]);
    } else {
        return accumulate(append, [],
            map(function(x) {
                return map(
                    function(p) {
                        return pair(x, p);
                    },
                    permutations(remove(x,s))
                );
            }, s)
        );
    }
}

function powerset(set) {
    if (is_empty_list(set)) {
        return list([]);
    } else {
        var rest_powerset = powerset(tail(set));
        var x = head(set);
        var has_x = map(function(s) {
            return pair(x, s);
            }, rest_powerset
        );
        return append(rest_powerset, has_x);
    }
}
```

Knapsap-like Problems

```
function coin_change(amount, coin_range) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || coin_range === 0) {
        return 0;
    } else {
        return coin_change(amount, coin_range - 1)
            + coin_change(
                amount - highest_value(coin_range),
                coin_range
            );
    }
}
```

Insertion sort takes elements from left to right, and *inserts* them into correct positions in the sorted portion of the list (or array) on the left. This is analagous to how most people would arrange playing cards.

```
function insertion_sort(xs) {
    return (is_empty_list(xs))
        ? xs
        : insert(head(xs),
            insertion_sort(tail(xs)));
}

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

Selection sort picks the smallest element from a list (or array) and puts them in order in a new list.

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

function smallest(xs) {
    function sm(x, ys) {
        return (is_empty_list(ys))
            ? x
            : (x < head(ys))
                ? sm(x, tail(ys))
                : sm(head(ys), tail(ys));
    }
    return sm(head(xs), tail(xs));
}
```

Quicksort is a divide-and-conquer algorithm. `Partition` takes a pivot, and positions all elements smaller than the pivot on one side, and those larger on the other. The two 'sides' are then partitioned again.

```
// QUICKSORT
function quicksort(xs) {
    if (is_empty_list(xs)) {
        return [];
    } else if (length(xs) === 1) {
        return xs;
    } else {
        var partitioned_xs =
            partition(tail(xs), head(xs));
        return append(
            quicksort(head(partitioned_xs)),
            pair(head(xs),
            quicksort(tail(partitioned_xs)))
        );
    }
}

function partition(xs, p) {
    function helper(elem, out_pair) {
        if (elem <= p) {
            return pair(
                append(head(out_pair), list(elem)),
                tail(out_pair)
            );
        } else {
            return pair(
                head(out_pair),
                append(tail(out_pair), list(elem))
            );
        }
    }
    return accumulate(helper, pair([], []), xs);
}
```

Bubble sort repeated swaps adjacent elements if they are in the wrong order, until the whole list (or array) is in the right order.

```
function sort(b) {
    var length_b = array_length(b);
    for (var num_sorted = 0;
        num_sorted < length_b;
        num_sorted = num_sorted + 1
    ) {
        for (var index = 0;
            index < length_b - 1;
            index = index + 1
        ) {
            if (b[index] > b[index+1]) {
                swap(index, index+1, b);
            } else { }
        }
    }
    return undefined;
}

function swap(left_index, right_index, array) {
    var tmp = array[left_index];
    array[left_index] = array[right_index];
    array[right_index] = tmp;
}
```