System Built-in Function
(up to Source Week 6)

1. Math constants

```
Math.E
Math.PI
Math.SQRT2
```

2. Math functions

```
Math.abs(x)
Math.sin(x)   Math.asin(x)
Math.cos(x)   Math.acos(x)
Math.tan(x)   Math.atan(x)
```

Return values - in radians.

```
Math.atan2(y, x)
```

Equivalent to Math.atan(y / x).

```
Math.floor(x)
Math.ceil(x)
Math.round(x)
Math.max(x,y,z,...,n)
Math.min(x,y,z,...,n)
Math.pow(x, y)
Math.exp(x)
```

The result of e in power of x

```
Math.sqrt(x)
Math.log(x)
```

The logarithm of x in base e

```
Math.log10(x)
```

The logarithm of x in base 10

```
Math.log2(x)
```

The logarithm of x in base 2

θ(n) → range
O(n) → upper bound
Ω(n) → lower bound

log(n) → log2(n)
lg(n) → log10(n)
ln(n) → $\log_e(n)$

3. List-related

```
Refer to the Appendix to the paper.
```

4. Others

```
is_number(x)
equal(x, y)
alert(string)
display(value)
prompt(string)
parseInt(string)
```

Returns the integer according to the input string.

Every list ends with a []. Be careful!

Important System Implementation

1. length

```
function length(xs) {
    if (is_empty_list(xs)) {
        return 0;
    } else {
        return 1 + length(tail(xs));
    }
}

function count_leaves(tree) {
    if (is_empty_list(tree)) {
        return 0;
    } else if (is__list(head(tree))) {
        return count_leaves(head(tree)) +
               count_leaves(tail(tree));
    } else {
        return 1 + count_leaves(tail(tree));
    }
}
```

2. reverse

```
function reverse(xs) {
    function rev(original, reversed) {
        if (is_empty_list(original)) {
            return reversed;
        } else {
            return rev(tail(original),
                       pair(head(original),
                            reversed));
        }
    }
    return rev(xs, []);
}

function tree_reverse(lst) {
    function op(origin, reversed) {
        if (is_empty_list(origin)) {
            return reversed;
        } else if (is_list(head(origin))) {
            return op(tail(origin),
                      pair(op(head(origin), []),
reversed));
        } else {
            return op(tail(origin),
                      pair(head(origin), reversed));
        }
    }
    return op(lst, []);
}
```

3. map

Notice: For map, filter and accumulate, we do not need to write the empty-list case when using them (because it has been built inside).

```
function map_tree(func, tree) {
    if (is_empty_list(tree)) {
        return [];
    } else if (is_list(head(tree))) {
        return pair(map_tree(func, head(tree)),
                    map_tree(func, tail(tree)));
    } else {
        return pair(func(head(tree)),
                    map_tree(func, tail(tree)));
    }
}

function map_tree(func, tree) {
    if (is_empty_list(tree)) {
        return [];
    } else if (is_list(head(tree))) {
        return pair(map_tree(func, head(tree)),
                    map_tree(func, tail(tree)));
    } else {
        return pair(func(head(tree)),
                    map_tree(func, tail(tree)));
    }
}
```

4. accumulate

Notice: accumulate means expanding from left to right and calculating from right to end.

```
function accumulate(op, initial, sequence) {
    if (is_empty_list(sequence)) {
        return initial;
    } else {
        return op(head(sequence),
                  accumulate(op, initial,
                             tail(sequence)));
    }
}
```

```
function accumulate_tree(op, init, tree) {
    if (is_empty_list(tree)) {
        return init;
    } else if (is_list(head(tree))) {
        return op(accumulate(op, init, head(tree)),
                  accumulate(op, init, tail(tree)));
    } else {
        return op(head(tree),
                  accumulate(op, init, tail(tree)));
    }
}
```

5. filter

```
function filter(func, lst) {
    if (func(head(lst))) {
        return pair(head(lst), filter(tail(lst)));
    } else {
        return filter(tail(lst));
    }
}
```

6. duplicate

```
function duplicates(lst) {
    return accumulate(function (x, accum) {
        if (is_empty_list(member(x, accum))) {
            return pair(x, accum);
        } else {
            return accum;
        }
    }, [], lst);
}
```

Important Applied Implementation

1. Hanoi

```
function hanoi(size, from, to, extra) {
    if (size === 0) {
        ;
    } else {
        hanoi(size - 1, from, extra, to);
        display("move from " + from + " to " + to);
        hanoi(size - 1, extra, to, from);
    }
}
```

2. coin changes

```
function ways_to_change(x) {
    function compute(amount, kind) {
        if (amount === 0) {
            return 1;
        } else if (amount < 0 || kinds === 0) {
            return 0;
        } else {
            return compute(amount, tail(kind)) +
            compute(amount - head(kind), tail(kind));
        }
    }
    return compute(x, 5);
}

function makeup_amount(x,lst) {
    if (is_pair(lst)) {
        var current = head(lst);
        var with_current = map(function (lst) {
            return pair(current, lst);
        }, makeup_amount(x - h, lst));
        var without_current = makeup_amount(x,
tail(lst));

        return append(with_current, without_current);
    } else if (x === 0) {
        return list([]);
    } else {
        return [];
    }
}
```

3. permutation

```
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 permutations_r(s, r) {
    if (r === 0) {
        // There is 1 permutation of length 0.
        return list([]);
    } else if (is_empty_list(s)) {
        // There is no permutation if s is empty but r is
not 0.
        return [];
    } else {
        return accumulate(append, [], map(function (x)
{
            return map(function (p) {
                return pair(x, p);
            },permutations_r(remove(x, s),r - 1));
        }, s));
    }
}
```

4. combination

```
function combinations(xs, k) {
    if (k === 0) {
        return list([]);
    } else if (is_empty_list(xs)) {
        return [];
    } else {
        var x = head(xs);
        var s1 = combinations(tail(xs), k - 1);
        var s2 = combinations(tail(xs), k);
        var with_x = map(function (s) {
                                return pair(x, s); }, s1);
        var without_x = s2;
        return append(with_x, without_x);
    }
}
```

(For personal use only)

Good luck!

--- End ---