System Built-in Function
(up to Source Week 12)

1. Math constants

```
Math.E
Math.PI
Math.SQRT2
Math.SQRT1_2
Math.LN10
Math.LN2
```

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

$\theta(n) \rightarrow$ range
$O(n) \rightarrow$ upper bound
$\Omega(n) \rightarrow$ lower bound

$\log(n) \rightarrow \log_2(n)$
$\lg(n) \rightarrow \log_{10}(n)$
$\ln(n) \rightarrow \log_e(n)$

3. List library

```
pair(x, y)
head(xs)
tail(xs)
set_head(xs, m)
set_tail(xs, m)
list(x1, x2, x3, x4, …)
length(xs)
list_ref(xs, n)
reverse(xs)
append(xs, ys)
```

Every list ends with a []. Be careful!

```
map(func, xs)
for_each(func, xs)
accumulate(func, accum_init, xs)
filter(pred, xs)
member(x, xs)
remove(x, xs)
remove_all(x, xs)
build_list(n, func)
enum_list(a, b)
list_to_string(xs)
```

4. Stream library

```
stream(x1, x2, x3, x4, …)
eval_stream(stream, n)
stream_tail(stream)
list_to_stream(xs)
stream_to_list(stream)
stream_length(stream)
stream_ref(stream, n)
stream_reverse(stream)
stream_append(xs, ys)
stream_map(func, stream)
stream_for_each(func, stream)
stream_accumulate(func, init, stream)
stream_filter(pred, stream)
stream_member(x, stream)
stream_remove(x, stream)
stream_remove_all(x, stream)
build_stream(n, func)
enum_stream(a, b)
integers_from(n)
```

5. Array method

```
[a1, a2, a3, a4, …]
arr[m][n][…]
arr.length;
```

6. String method

```
str.substring(a, b)
```

Returns the sub-string from [a, b) or [a, b - 1], where counting starts from 0. If a ≥ b, it will return a null string.

7. Object-oriented Programming (OOP) support

Two ways to visit a field of a class:

1) `<expression>.<id>`
2) `<expression>["id"]`

Two ways to call a method of a class:

1) `<expression>.<id>(…, …, …)`
2) `(<expression>["id"]).call(this, …)`

Creating new object of a certain class

```
<expression> = new <id>(<…, …, …>)
```

Declaring a method for a certain class

```
F.prototype.<name> = function (…) {
    …
};
```

Inheritance from super-class

```
G.Inherits(F)
```

Calling a constructor (usually from super-class)

```
G.call(this, …)
```

Invoking a method from a foreign class

```
G.prototype.<name>.call(this, …)
```

8. Loop support

```
while(pred) { … }
for(init; pred; increment) { … }
break
continue
```

9. Data-type checking

```
is_undefined(x)
is_number(x)
is_string(x)
is_boolean(x)
is_object(obj)
is_pair(pair)
is_list(lst)
is_empty_list(lst)
is_stream(strm)
is_array(arr)
```

10. Others

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

Returns the integer according to the input string.

```
system.get_globals()
JSON.stringify(parse("<statements>"))
```

Important System Implementation

1. 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, []);
}
```

2. 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).

3. accumulate

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

4. 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);
    }
}
```

5. power-sets of a set

```
function power_set(xs) {
    if (is_empty_list(xs)) {
        return list([]);
    } else {
        var without_it = power_set(tail(xs));
        var with_it = map(function (x) {
            return pair(head(xs), x);
        }, without_it);

        return append(without_it, with_it);
    }
}
```

6. partition of a set

```
function partition(xs) {
    if (is_empty_list(xs)) {
        return list([]);
    } else if (is_empty_list(tail(xs))) {
        return list(list(list(head(xs))));
    } else {
        var after_this = partition(tail(xs));
        var cut = map(function (x) {
            return pair(list(head(xs)), x);
        }, after_this);
        var no_cut = map(function (x) {
            return pair(pair(head(xs), head(x)),
tail(x));
        }, after_this);

        return append(cut, no_cut);
    }
}
```

7. Mutable reverse of a list

```
function mutable_reverse1(xs) {
    if (is_empty_list(xs) || is_empty_list(tail(xs))) {
        return xs;
    } else {
        var temp = mutable_reverse1(tail(xs));
        set_tail(tail(xs), xs);
        set_tail(xs, []);
        return temp;
    }
}
function mutable_reverse2(xs) {
    function helper(prev, left) {
        if (is_empty_list(left)) {
            return prev;
        } else {
            var temp = tail(left);
            set_tail(left, prev);
            return helper(left, temp);
        }
    }

    return helper([], xs);
}
```

8. Interleave of a list of streams

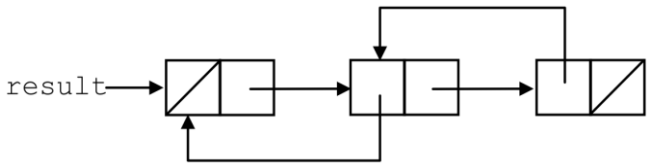
```
function merge_streams(ss) {
    if (is_empty_list(ss)) {
        return [];
    } else if (is_empty_list(head(ss))) {
        return merge_streams(tail(ss));
    } else {
        return pair(head(head(ss)), function () {
            return merge_streams(append(tail(ss),
                                        list(stream_tail(head(ss)))));
        });
    }
}
```

Drawing Diagrams

1. Box-and-pointer Diagrams

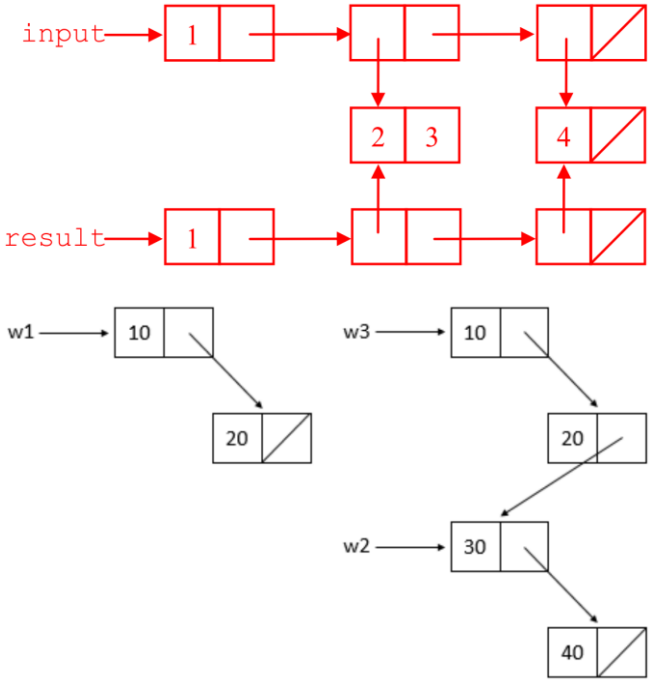
A few points to stress:

1) Without set_head or set_tail, we cannot create circular structures in the Source;



If you cannot figure out whether it is circular or not, represent each pair as a dot to get a directed graph and see whether it is circuit-free.

2) Differentiate reference-by-value / reference (address);

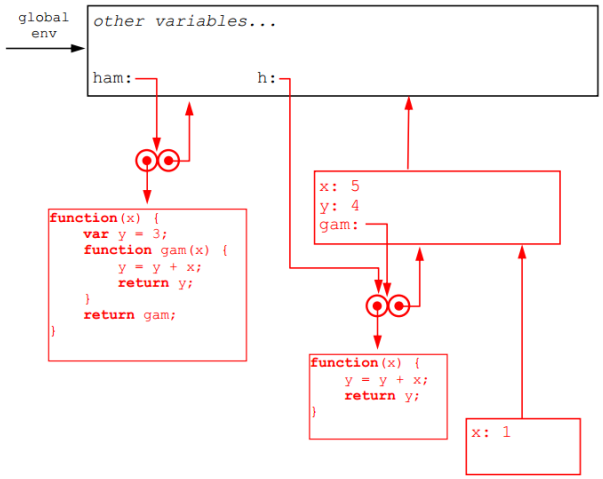


3) About layout: it will much tidier if you put input and result into two separate horizontal lines

2. Environment Model

A few points to stress:

1) For function definition, draw two circles, one pointing to its environment while the other pointing to its body;
2) For function application, draw a new box and
   a. fill in the values of parameters;
   b. fill in the values of local variables;
   c. we cannot represent the existence of return value;
3) For recursive call, you need multiply boxes;
4) Always update the values bounding to variables no less and no more than the point of execution;
5) Special care to higher-order functions:
   a. keeps the chain-relationship of frames;
   b. understands the first-class feature of functions;
   c. knows when and where the functions are defined / evaluated.



Meta-circular Evaluator

1. Manipulation of environments and frames

```
function make_frame(variables, values) {
        if (is_empty_list(variables) &&
            is_empty_list(values)) {
            return {};
        } else {
    var frame = make_frame(tail(variables), tail(values));
            frame[head(variables)] = head(values);
            return frame;
        }
}
function extend_environment(vars, vals, base_env) {
        var var_length = length(vars);
        var val_length = length(vals);
        if (var_length === val_length) {
            var new_frame = make_frame(vars, vals);
            return enclose_by(new_frame, base_env);
        } else if (var_length < val_length) {
            error("Too many arguments supplied: " + vars
+ " " + vals);
        } else {
            error("Too few arguments supplied: " + vars +
" " + vals);
        }
}
function lookup_variable_value(variable, env) {
        function env_loop(env) {
            if (is_empty_environment(env)) {
                error("Unbound variable: " + variable);
            } else if (has_binding_in_frame(variable,
first_frame(env))) {
                return first_frame(env)[variable];
            } else {
                return env_loop(
                    enclosing_environment(env));
            }
        }
        return env_loop(env);
}
```

2. Handle with var definition

```
function evaluate_var_definition(stmt, env) {
        define_variable(var_definition_variable(stmt),
            evaluate(var_definition_value(stmt), env),
                env);
        return undefined;
}
```

3. Handle with function definition

```
function evaluate_function_definition(stmt, env) {
        return make_function_value(
            function_definition_parameters(stmt),
            function_definition_body(stmt),
            env);
}
function make_function_value(parameters, body, env) {
        return { tag: "function_value",
            parameters: parameters,
            body: body,
            environment: env
        };
}
```

4. Handle with function application

```
function apply(fun, args) {
        if (is_primitive_function(fun)) {
            return apply_primitive_function(fun, args);
        } else if (is_compound_function_value(fun)) {
            if (length(function_value_parameters(fun))
=== length(args)) {
                var env = extend_environment(
                    function_value_parameters(fun), args,
                    function_value_environment(fun));
                var result = evaluate(
                        function_value_body(fun), env);
                if (is_return_value(result)) {
                    return return_value_content(result);
                } else {
                    return undefined;
                }
            } else {
                error("Incorrect number of arguments
supplied for function");
            }
        } else {
            error("Unknown function type -- apply: " + fun);
        }
}
function list_of_values(exps, env) {
        if (no_operands(exps)) {
            return [];
        } else {
            return pair(evaluate(first_operand(exps), env),
                list_of_values(rest_operands(exps), env));
        }
}
```

5. Handle with sequences of statements

```
function evaluate_sequence(stmts, env) {
        if (is_last_statement(stmts)) {
            return evaluate(first_statement(stmts), env);
        } else {
            var first_stmt_value =
                evaluate(first_statement(stmts), env);
            if (is_return_value(first_stmt_value)) {
                return first_stmt_value;
            } else {
                return evaluate_sequence(
                rest_statements(stmts), env);
            }
        }
}
```

(For personal use only)

Good luck!

--- End ---