

```

1 //List operations
2 function remove_duplicates(lst){
3     if(is_empty_list(lst)){
4         return [];
5     }else{
6         var h = head(lst);
7         return pair(h,
8             accumulate(function(a,b){
9                 return a!=h?pair(a,b):b;
10                },[],remove_duplicates(tail(lst))));
11     }
12 }
13 function are_equal_sets(set1,set2){
14     var r = map(function(e){return !is_empty_list(member(e,set1));},set2);
15     var b = accumulate(function(a,b){return a&&b;},true,r);
16     if(length(set1)==0&&length(set2)==0){
17         return true;
18     }else if(length(set1)==length(set2)&&b){
19         return true;
20     }else{
21         return false;
22     }
23 }
24 function mergeB(xs, ys) {
25     if (is_empty_list(xs) && is_empty_list(ys)) {
26         return [];
27     } else if (is_empty_list(xs)) {
28         set_tail(ys, mergeB(xs, tail(ys)));
29         return ys;
30     } else if (is_empty_list(ys)) {
31         set_tail(xs, mergeB(tail(xs), ys));
32         return xs;
33     } else if (head(xs) <= head(ys)) {
34         set_tail(xs, mergeB(tail(xs), ys));
35         return xs;
36     } else {
37         set_tail(ys, mergeB(xs, tail(ys)));
38         return ys;
39     }
40 }
41 //Trees
42 function accumulate_tree(op,init,tree){
43     if(is_empty_list(tree)){
44         return init;
45     }else if(!is_list(head(tree))){
46         return op(head(tree),accumulate_tree(op,init,tail(tree)));
47     }else{
48         return op(accumulate_tree(op,init,head(tree)),
49             accumulate_tree(op,init,tail(tree)));
50     }
51 }
52
53 function count_leaves(tree) {
54     return (is_empty_list(tree))
55         ? 0
56         : (is_list(head(tree))
57             ? count_leaves(head(tree))
58             : 1)
59         + count_leaves(tail(tree));
60 }
61
62 function longest_path(tree){
63     if(!is_list(tree)){

```

```

64         return 1;
65     }else
66     if(is_empty_list(tree)){
67         return 1;
68     }else if(!is_list(head(tree)))
69     {
70         return 1+longest_path(tail(tree));
71     }else{
72         return 1+Math.max(longest_path(head(tree)),longest_path(tail(tree)));
73     }
74 }
75 //Somewhat related
76 function cc(amount, kinds) {
77     if (amount === 0) {
78         return 1;
79     } else if (amount < 0 || kinds === 0) {
80         return 0;
81     } else {
82         return cc(amount, kinds - 1) + cc(amount - highest_denom(kinds), kinds);
83     }
84 }
85 //Permutations
86 function permutations(s) {
87     if (is_empty_list(s)) {
88         return list([]);
89     } else {
90         return accumulate(
91             append, [],
92             map(function(x) {
93                 return map(function(p) {
94                     return pair(x,p);
95                 },
96                 permutations(remove(x,s)));
97             },
98             s)
99         );
100     }
101 }
102 function permutations_r(s,r) {
103     if (r===0||is_empty_list(s)) {
104         return list([]);
105     } else {
106         return accumulate(append, [],
107             map(function(x) {
108                 return map(function(p) {
109                     return pair(x, p);
110                 },
111                 permutations_r(remove(x, s),r-1));
112             }, s));
113     }
114 }
115 function combinations(xs,k){
116     if(k===0){
117         return list([]);
118     }else if(is_empty_list(xs)){
119         return [];
120     }else{
121         var s1 = combinations(tail(xs),k-1);
122         var s2 = combinations(tail(xs),k);
123         var x = head(xs);
124         var has_x = map(function(s){return pair(x,s);},s1);
125         return append(has_x,s2);
126     }
127 }

```

```

128 //Power set
129 function power_set(s){
130   if(is_empty_list(s)){
131     return list([]);
132   }else{
133     var rest = power_set(tail(s));
134     return append(rest,map(function(x){return pair(head(s),x);},rest));
135   }
136 }
137 //Subsequence
138 function all_subsequences(xs){
139   if (is_empty_list(xs)){
140     return list([]);
141   }else{
142     var ys = all_subsequences(tail(xs));
143     return append(ys,
144       map(
145         function(y){
146           return pair(head(xs),y);
147         },
148         ys));
149   }
150 }
151 //Tower of Hanoi
152 function move_tower(size, from, to, extra) {
153   if (size === 0) {
154     ;
155   } else {
156     move_tower(size - 1, from, extra, to);
157     display("move from " + from + " to " + to);
158     move_tower(size-1, extra, to, from);
159   }
160 }
161 //OOP
162 function Class(arg1,arg2){
163   this.field1 = arg1;
164   this.field2 = arg2;
165   //or
166   //SuperClass.call(this,arg1,arg2);
167 }
168 Class.Inherits(Class);
169 Class.prototype.method1 = function(a,b){
170   //Superclass.prototype.methodname.call(a,b);
171 }
172 //Tream
173 var e = function(){return [];};
174 var a = pair(2,function(){return pair(4,e);});
175 var b = pair(3,function(){return pair(5,e);});
176 var t = pair(a,function(){return pair(b,e);});
177
178 function tream_map(f, t){
179   if (is_empty_list(t)){
180     return [];
181   } else {
182     var x = function() {
183       return tream_map(f, stream_tail(t));
184     };
185     if(is_pair(head(t))){
186       return pair(tream_map(f, head(tree)), x);
187     }else{
188       return pair(f(head(tree)), x);
189     }
190   }
191 }

```

```

192 //Loop
193 var count_pair= (function(seen){
194     function fn(x){
195
196         if(! is_pair(x)){
197             return 0;
198         }else if(!is_empty_list(member(x,seen))){
199             return 0;
200         }else{
201             //display(x); //For debugging
202             seen = append(list(x),seen);
203             //display("What is in seen is "+seen);
204             return count_pair(head(x))+count_pair(tail(x))+1;
205         }
206     }
207     return fn;
208 })([]);
209 var has_loop = (function(seen){
210     function fn(x){
211         if(is_empty_list(x)){
212             return false;
213         }else if(!is_empty_list(member(x,seen))){
214             return true;
215         }else{
216             seen = append(list(x),seen);
217             return has_loop(tail(x));
218         }
219     }
220     return fn;
221 })([]);
222 function has_loop(lst){
223     function next(xs){
224         if(equal(xs,false)||is_empty_list(xs)||is_empty_list(tail(xs))){
225             return false;
226         }else{
227             return tail(xs);
228         }
229     }
230     function fn(p1,p2){
231         display(p1);
232         display(p2);
233         if(equal(p1,false)||equal(p2,false)){
234             return false;
235         }else{
236             if(p1==p2){
237                 return true;
238             }else{
239                 return fn(next(p1),next(next(p2)));
240             }
241         }
242     }
243     return fn(next(lst),next(next(lst)));
244 }
245 function memoize(f) {
246     var table = make_table();
247     return function(x) {
248         if (has_key(x, table)) {
249             return lookup(x, table);
250         } else {
251             var result = f(x);
252             insert(x, result, table);
253             return result;
254         }
255     }

```

```

256 }
257 function memo_fun(fun) {
258   var already_run = false;
259   var result = undefined;
260   return function() {
261     if (!already_run) {
262       result = fun();
263       already_run = true;
264       return result;
265     } else {
266       return result;
267     }
268   };
269 }
270 //Mutable List
271 function mutable_reverse(xs) {
272   // VERSION 1
273   if (is_empty_list(xs)) {
274     return xs;
275   } else if (is_empty_list(tail(xs))) {
276     return xs;
277   } else {
278     var temp = mutable_reverse(tail(xs));
279     set_tail(tail(xs), xs);
280     set_tail(xs, []);
281     return temp;
282   // VERSION 2
283   function helper(prev, xs) {
284     if (is_empty_list(xs)) {
285       return prev;
286     } else {
287       var rest = tail(xs);
288       set_tail(xs, prev);
289       return helper(xs, rest);
290     }
291   }
292   return helper([], xs);
293 }
294 //Stream
295 function inverse_unit_series(s){
296   return pair(1,function(){return scale_series(-1,mul_series(stream_tail(s),
297     inverse_unit_series(s))));});
298 }
299 function div_series(s1,s2){//s1 / s2 check divisibility before dividing
300   var constant_term = head(s2);
301   if(constant_term===0){
302     return "Error: divided by 0";
303   }else{
304     return scale_stream(1/constant_term,function(){return mul_series(s1,inverse_unit_series(
305       scale_series(1/constant_term,s2))));});
306   }
307 }
308 function stream_append_pickle(xs, ys) {
309   if (is_empty_list(xs)) {
310     return ys();
311   } else {
312     return pair(head(xs),
313       function() {
314         return stream_append_pickle(stream_tail(xs),
315           ys);
316       });
317   }
318 }

```

```

318
319 function sieve(s) {
320     return pair(head(s),
321         function() {
322             return sieve(stream_filter(function(x) {return !is_divisible(x, head(s));}, stream_tail(
323                 s)));
324         });
325 }
326 var primes = sieve(integers_from(2));
327 //Intepreter
328 //Question 2
329 function is_variable_assignment(stmt){
330     return stmt.tag==="assignment";
331 }
332 //Question 2
333 function is_variable_assignment(stmt){
334     return stmt.tag==="assignment";
335 }
336 function set_variable_value(variable, val, env) {
337     function env_loop(env) {
338         if (is_empty_environment(env)) {
339             error("Unbound variable: " + variable);
340         } else if (has_binding_in_frame(variable.name, first_frame(env))) {
341             first_frame(env)[variable.name] = force(evaluate(val, env));
342             return undefined;
343             /*
344             var val = force(first_frame(env)[variable]);
345             first_frame(env)[variable] = val;
346             return val;
347             */
348         } else {
349             return env_loop(enclosing_environment(env));
350         }
351     }
352     return env_loop(env);
353 }
354 //Question 3
355 function is_boolean_operation(stmt){
356     return stmt.tag==="boolean_op";
357 }
358 function evaluate_boolean_operation(val1, val2, operator, env){
359     var v1 = force(evaluate(val1, env));
360     var v2 = force(evaluate(val2, env));
361     if( operator==="||"){
362         return v1||v2;
363     }else{
364         return v1&&v2;
365     }
366 }
367 //Question 4
368 function is_while_statement(stmt){
369     return stmt.tag==="while";
370 }
371 function evaluate_while_statement(predicate, list_of_statements, env){
372     function loop(remaining_statements){
373         if(is_empty_list(remaining_statements)){
374             return evaluate_while_statement(predicate, list_of_statements, env);
375         }else{
376             force(evaluate(head(remaining_statements), env));
377             loop(tail(remaining_statements));
378         }
379     }
380     if(force(evaluate(predicate, env))===true){
381         loop(list_of_statements);

```

```

381     }else{
382         return undefined;
383     }
384 }
385 //For loop
386 function is_for_loop(stmt){
387     return stmt.tag=="for";
388 }
389 function evaluate_for_loop(initialiser,predicate,finaliser,statements,env){
390     function loop(remaining_statements){
391         if(is_empty_list(remaining_statements)){
392             force(evaluate(finaliser,env));
393             if(force(evaluate(predicate,env))===true){
394                 return evaluate_for_loop(undefined,predicate,finaliser,statements,env);
395             }else{
396                 return undefined;
397             }
398         }else{
399             var h = head(remaining_statements);
400             force(evaluate(head(remaining_statements),env));
401             loop(tail(remaining_statements));
402         }
403     }
404     if(initialiser===undefined){
405         ;
406     }else{
407         force(evaluate(initialiser,env));
408     }
409     loop(statements);
410     return undefined;
411 }
412
413 //Reference
414 build_list(n, f) //Makes a list with n elements
415     by applying the unary function f to the numbers 0 to n - 1.
416 for_each(f, xs) //Applies f to every element of the list xs, and then returns true.
417 remove(x, xs) remove_all(x, xs)
418 enum_list(start, end)
419     // Returns a list that enumerates numbers starting from start
420     // using a step size of 1, until the number exceeds (>) end.

```



All da best tomorrow!