HW1. Ocaml

```
type ('nonterminal, 'terminal) symbol =
  | N of 'nonterminal
  | T of 'terminal

let rec subset a b =
    match a with
    [] -> true
    | h::t -> if List.mem h b then subset t b else false;;

let equal_sets a b = subset a b && subset b a;;

let rec set_union a b = match a with
    | [] -> b
    | h::t -> if List.exists (fun x -> x = h) b then set_union t b else set_union t (h::b);;

let set_intersection a b = List.find_all (fun x -> List.mem x a) b;;

let set_diff a b = List.filter (fun x -> List.for_all (fun elem -> elem <> x) b) a;;

let rec computed_fixed_point eq f x = if eq (f x) x then x else computed_fixed_point eq f (f x);;

let rhs (_, x) = x;;
let lhs (x, _) = x;;

let rec filter_reachable_rules start_nt grules =
    let rec get_nonterms rlist =
        match rlist with
        | [] -> []
        | N h::t -> (h)::(get_nonterms t)
        | T h::t -> get_nonterms t in
    let rec nonterm_parse cur_nonterms rt_list =
        match rt_list with
        | [] -> cur_nonterms
        | rules::tup_list ->
        if (List.mem (lhs rules) cur_nonterms)
            then nonterm_parse (set_union cur_nonterms (get_nonterms (rhs rules))) tup_list
        else nonterm_parse cur_nonterms tup_list in
    let rec all_nonterms cur_nonterms rt_list =
        let r1 = nonterm_parse cur_nonterms rt_list in
        let r2 = nonterm_parse r1 rt_list in
            if equal_sets r1 r2 then r1 else all_nonterms r2 rt_list in

    let rn = (all_nonterms [start_nt] grules) in
    List.filter (fun x -> List.mem (lhs x) rn) grules
;;

let filter_reachable g = match g with
    | (start_nt, rules) -> (start_nt, filter_reachable_rules start_nt rules)
    | _ -> g
;;
```

HW2. Ocaml

```
open List;;

type ('nonterminal, 'terminal) parse_tree =
| Node of 'nonterminal * ('nonterminal, 'terminal) parse_tree list
| Leaf of 'terminal

type ('nonterminal, 'terminal) symbol =
| N of 'nonterminal
| T of 'terminal

let acc_empty_suffix = function
| _::_ -> None
| x -> Some x;;


(* Number 1 *)
let rec get_rhs rules lhs = match rules with
| hd::tl ->
if (lhs == (fst hd)) then
(snd hd) :: (get_rhs tl lhs)
else
(get_rhs tl lhs)
| [] -> [];;

let convert_grammar gram =
(fst gram), (get_rhs (snd gram))
;;

(* Number 2 *)
let rec tree_helper tree_list = match tree_list with
| [] -> []
| h :: t -> match h with
| Leaf l -> l :: (tree_helper t)
| Node (nt, tree) -> (tree_helper tree) @ (tree_helper t)
;;

let parse_tree_leaves tree =
tree_helper [tree]
;;



let rec create tnode derv =
let rec c_helper tlist derv =
match tlist with
| [] -> None
| tnode::t -> match tnode with
          | Node (x, y) -> (match (create (Node (x, y)) derv) with
                                                  | Some z -> Some (z::t)
                                                  | None -> (match (c_helper t derv) with
                                                                              | Some z -> Some ((Node (x, y))::z)
                                                                             | None -> None))
          | Leaf x -> (match (c_helper t derv) with
                                      | None -> None
```

```
                                        | Some y -> Some ((Leaf x)::y))
in
let node t_n =
match t_n with
| T t -> Leaf t
| N n -> Node (n, []) in
match tnode with
| Node (x, y) -> (match y with
                   | y when y = [] -> Some (Node (x, map node derv))
                   | m -> (match (c_helper m derv) with
                                     | Some lst -> Some (Node (x, lst))
                                     | None -> None))
| Leaf x -> None
;;


let rec matcher_help_list sym rlist prules acc rules frag =
let rec match_helper rule prules acc rules frag =
(match rule with
| rule_h::rule_t -> (match frag with

          | frag_h::frag_t -> (match rule_h with
                    | N non_terminal -> matcher_help_list non_terminal (prules non_terminal) prules
(match_helper rule_t prules acc) rules frag
                    | T terminal -> (if terminal = frag_h
                                                                  then match_helper rule_t prules acc rules
frag_t
                                                                  else None, rules))
          | [] -> None, rules)
| [] -> (acc rules frag))
in
match rlist with
| [] -> None, rules
| h::t ->           match (create rules h) with
          | Some x ->
                    (match (match_helper h prules acc x frag) with
                    | (Some y, ur) -> (Some y, ur)
                    | (None, _) -> (matcher_help_list sym t prules acc rules frag))
          | None -> (None, rules)
                    ;;

(* Number 3 *)
let make_matcher gram =
let curr = (snd gram) (fst gram) in
let matcher acc frag =
let acceptor rules frag = (acc frag, rules) in
(fst (matcher_help_list (fst gram) curr (snd gram) acceptor (Node ((fst gram), [])) frag))

in

matcher;;

(* Number 4 *)
let make_parser gram =
let curr_expr = (snd gram) (fst gram) in

let get_matchrule acc frag =
let acceptor rules frag = (acc frag, rules) in
```

```
(matcher_help_list (fst gram) curr_expr (snd gram) acceptor (Node ((fst gram), [])) frag) in

let parser frag =
let rules = (snd (get_matchrule acc_empty_suffix frag)) in
let matcher = (fst (get_matchrule acc_empty_suffix frag)) in
match matcher with
| Some _ -> Some (rules)
| None -> None
in
parser;;
```

HW4. Prolog Tower.pl

```
rotate(Matrix, RotatedMatrix) :-
        transpose_matrix(Matrix, T),
        reverse_rows(T, RotatedMatrix).

reverse_rows([], []).
reverse_rows([R|Rem], [Rev|RemRevs]) :-
        reverse(R, Rev),
        reverse_rows(Rem, RemRevs).

transpose_matrix([], []).
transpose_matrix([R|Rem], T) :-
        transpose_matrix(R, [R|Rem], T).

transpose_matrix([], _, []).
transpose_matrix([_|Rem], CurM, [TRow|RemTrans]) :-
        lists_firsts_rests(CurM, TRow, NewMatrix),
        transpose_matrix(Rem, NewMatrix, RemTrans).

lists_firsts_rests([], [], []).
lists_firsts_rests([[F|Os]|Rest], [F|Fs], [Os|Oss]) :-
      lists_firsts_rests(Rest, Fs, Oss).

unique_mat(RightM, R, D, L, U) :-
        right_rules(RightM, R),
        rotate(RightM, UpMat),
        right_rules(UpMat, U),
        rotate(UpMat, LeftMat),
        right_rules(LeftMat, L),
        rotate(LeftMat, DownMat),
        right_rules(DownMat, D).

corr_vals([], _).
corr_vals([Cur|Rem], N) :-
        Cur #< N,
        corr_vals(Rem, N).

right_rules([], []).
right_rules([R|Rem], [RRule|RemRules]) :-
        right_rules(Rem, RemRules),
        corr_rule(R, RRule).

corr_rule([], F) :-
        F = 0.
```

```
corr_rule([H|T], F) :-
        \+ corr_vals(T, H),
        corr_rule(T, F).
corr_rule([H|T], F) :-
        corr_vals(T, H),
        corr_rule(T, X),
        F is X + 1.


check_lens(U, D, L, R, N) :-
        length(U, N), length(D, N), length(R, N), length(L, N).


row_len([], _).
row_len([R|Rem], N) :-
        length(R, N),
        row_len(Rem, N).


plain_dom_int(N, X) :-
        my_domain(X, 1, N).


dom_int(N, X) :-
        fd_domain(X, 1, N).


my_domain([], _, _).
my_domain([Cur|Rem], Start, End) :-
        Cur #>= Start,
        Cur #=< End,
        my_domain(Rem, Start, End).


my_label(N, L) :-
        findall(Num, between(1, N, Num), X),
        permutation(X, L).


all_diff_check([]).
all_diff_check([H|Rem]) :-
        not_in_list(H, Rem),
        all_diff_check(Rem).


not_in_list(_, []).
not_in_list(Cur, [H|Rem]) :-
        Cur #\= H,
        not_in_list(Cur, Rem).


prevent0(T0,T1,T):-
   T0 == T1,
   T is 1;
   T0 \= T1,
   T is T1-T0.
```

```
tower(N, T, C) :-
        length(T, N),
        row_len(T, N),
        C = counts(U, D, L, R),
        check_lens(U, D, L, R, N),
        maplist(dom_int(N), T),
        maplist(fd_all_different, T),
        rotate(T, Rot),
        maplist(dom_int(N), Rot),
        maplist(fd_all_different, Rot),
        maplist(fd_labeling, T),
        reverse(RevL, L),
        reverse(RevD, D),
        unique_mat(T, R, RevD, RevL, U).


plain_tower(N, T, C) :-
        length(T, N),
        row_len(T, N),
        C = counts(U, D, L, R),
        check_lens(U, D, L, R, N),
        maplist(plain_dom_int(N), T),
        maplist(all_diff_check, T),
        rotate(T, Rot),
        maplist(dom_int(N), Rot),
        maplist(all_diff_check, Rot),
        maplist(my_label(N), T),
        reverse(RevL, L),
        reverse(RevD, D),
        unique_mat(T, R, RevD, RevL, U).


speedup(Ratio) :-
        % Get tower time
        statistics(cpu_time,[T0,_]),
   tower(5, _, counts([2,1,3,3,2],[3,4,3,2,1],[2,1,2,4,4],[2,3,3,2,1])),
   statistics(cpu_time,[T1,_]),

   % Get plain tower time
   plain_tower(5, _, counts([2,1,3,3,2],[3,4,3,2,1],[2,1,2,4,4],[2,3,3,2,1])),
   statistics(cpu_time,[T2,_]),
        prevent0(T0,T1,T),
        Ratio is (T2-T1)/T.


ambiguous(N, C, T1, T2) :-
        C = counts(_, _, _, _),
        tower(N, T1, C),
        tower(N, T2, C),
        T1 \= T2.
```

HW5. Scheme

```
(define (cons* a b) (cons a b))

(define (self-return a) a)

(define (bind a b)
	(string->symbol
		(string-append
			(symbol->string a) "!" (symbol->string b)
		)
	)
)

(define (replace-all v l) (cond
	([empty? l] '())
	([equal? (type-check l) 'lambda] (cons (car l) (replace-all v (cdr l))))
	([not (pair? (car l))] (cons (v (car l)) (replace-all v (cdr l))))
	(else (cons (replace-all v (car l)) (replace-all v (cdr l))))
))

(define (type-check x)
	(if [not (pair? x)] 'base
	(let ([hd (car x)]) (cond
		([and (pair? hd)
			(or (equal? (car hd) 'lambda) (equal? (car hd) (string->symbol "\u03BB")))]
'lambda)
		([equal? hd 'if] 'if)
		([equal? hd 'quote] 'base)
		(else 'list)
	))
))


(define (expr-compare-lambda cur_a cur_b ali bli) (cond
	([not (empty? ali)]
		(if [not (equal? (car ali) (car bli))]
			(let
				([new_a (lambda (x)
					(cond ([not (equal? x (car ali))] (cur_a x)) (else (bind (car ali)
(car bli)))))]
				[new_b (lambda (y)
					(cond ([not (equal? y (car bli))] (cur_b y)) (else (bind (car ali)
(car bli)))))])
				(cons
					(bind (car ali) (car bli))
					(expr-compare-lambda new_a new_b (cdr ali) (cdr bli))
				)
			)
			(cons (car ali) (expr-compare-lambda cur_a cur_b (cdr ali) (cdr bli))
			)
		)
	)
	(else
```

```
            (list cur_a cur_b)
        )
))

(define (expr-compare a b) (let ([atype (type-check a)] [btype (type-check b)]) (cond
        ([equal? atype 'quote] (cond
        ([and (equal? a #f) (equal? b #t)] '(not %))
        ([and (equal? a #t) (equal? b #f)] '%)
        ([equal? a b] a)
        (else (list 'if '% a b))))

        ([equal? atype 'base] (cond
        ([and (equal? a #f) (equal? b #t)] '(not %))
        ([and (equal? a #t) (equal? b #f)] '%)
        ([equal? a b] a)
        (else (list 'if '% a b))))

        ([equal? atype 'lambda]
            (cons (let ([c (cdr (car a))] [d (cdr (car b))]
                [lambda_new (if (not (equal? (car (car a)) (car (car b))))
                (string->symbol "\u03BB") (car (car a)))])
            (let* ([ret (expr-compare-lambda self-return self-return (car c) (car d))]
                    [r (take-right ret 2)] [res (drop-right ret 2)]
                    [ail (replace-all (first r) (cdr c))]
                    [bil (replace-all (second r) (cdr d))])
                    (cons* lambda_new (cons* res (expr-compare ail bil)))))
            (expr-compare (cdr a) (cdr b))))

        ([not (equal? atype btype)] (list 'if '% a b))
        ([not (equal? (length a) (length b))] (list 'if '% a b))

        (else (cons (expr-compare (car a) (car b)) (expr-compare (cdr a) (cdr b))))
)))


(define (test-expr-compare x y)
  (and (equal? (eval x) (eval (list 'let '((% #t)) (expr-compare x y))))
      (equal? (eval y) (eval (list 'let '((% #f)) (expr-compare x y))))
))

(define test-expr-x '(+ 3 ((lambda (a b) (list a b)) 1 2)))
(define test-expr-y '(+ 2 ((lambda (a c) (list a c)) 1 2)))
```