

CS 421, Fall 2006

Sample Final Questions & Answers

1. Write a function `get_primes : int -> int list` that returns the list of primes less than or equal the input. You may use the built-in functions `/` and `mod`. You will probably want to write one or more auxiliary functions. Remember that 0 and 1 are not prime.

Answer:

```
let rec every p l = match l with [] -> true | x::xs -> p x && every p xs
let not_divides n q = not(n mod q = 0)
let rec get_primes n =
  match n with 0 -> []
  | 1 -> []
  | _ -> let primes = get_primes (n-1) in
        if every (not_divides n) primes then n::primes else primes
```

2. Write a tail-recursive function `largest: int list -> int option` that returns `Some` of the largest element in a list if there is one, or else `None` if the list is empty.

Answer:

```
let rec largest_aux lgst list =
  match list with [] -> lgst
  | x :: xs -> match lgst with None -> largest_aux (Some x) xs
                | Some l ->
                  largest_aux (if l > x then lgst else (Some x)) xs

let largest = largest_aux None

(* I would also accept *)
let largest list =
  List.fold_left
    (fun lgst x -> match lgst with None -> Some x
                  | Some l -> if l > x then lgst else Some x)
    None
  list
```

3. a. Give the types for following functions (you don't have to derive them):

```
let first lst = match lst with
  | a::aa -> a;;

let rest lst = match lst with
  | [] -> []
  | a::aa -> aa;;
```

Answer: first : α list $\rightarrow \alpha$
rest : α list $\rightarrow \alpha$ list

Use these types to derive the types for:

b. let rec foldright f lst z = if lst = [] then z else (f (first lst) (foldright f (rest lst) z)) in foldright (+) [2;3;4] 0

Answer: let rec foldright f lst z = if lst = [] then z else (f (first lst) (foldright f (rest lst) z))
in
foldright (+) [2;3;4] 0

Because of space constraints the proof tree has been broken up into four parts.

Let $\Gamma = \{ \text{first} : \alpha \text{ list} \rightarrow \alpha, \text{rest} : \alpha \text{ list} \rightarrow \alpha \text{ list} \}$,

and $\Gamma' = \Gamma \cup \{ \text{foldright} : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\alpha \text{ list}) \rightarrow \beta \rightarrow \beta, \}$.
 $f : \alpha \rightarrow \beta \rightarrow \beta,$
 $\text{lst} : \alpha \text{ list}, z : \beta$

Let *Tree1* =

$$\frac{\frac{\Gamma' \vdash f : \alpha \rightarrow \beta \rightarrow \beta}{\Gamma' \vdash f (\text{first lst}) : \beta \rightarrow \beta} \quad \frac{\Gamma' \vdash \text{first} : \alpha \text{ list} \rightarrow \alpha \quad \Gamma' \vdash \text{lst} : \alpha \text{ list}}{\Gamma' \vdash \text{first lst} : \alpha}}{\Gamma' \vdash f (\text{first lst}) : \beta \rightarrow \beta}$$

Let *Tree2* =

$$\frac{\frac{\frac{\Gamma' \vdash \text{foldright} : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\alpha \text{ list}) \rightarrow \beta \rightarrow \beta \quad \Gamma' \vdash f : (\alpha \rightarrow \beta \rightarrow \beta)}{\Gamma' \vdash \text{foldright } f : (\alpha \text{ list}) \rightarrow \beta \rightarrow \beta} \quad \frac{\Gamma' \vdash \text{rest} : \alpha \text{ list} \rightarrow \alpha \text{ list} \quad \Gamma' \vdash \text{lst} : \alpha \text{ list}}{\Gamma' \vdash \text{rest lst} : \alpha \text{ list}}}{\Gamma' \vdash \text{foldright } f (\text{rest lst}) : \beta \rightarrow \beta} \quad \Gamma' \vdash z : \beta}{\Gamma' \vdash \text{foldright } f (\text{rest lst}) z : \beta}$$

Let *Tree3* =

$$\frac{\frac{\frac{\Gamma' \vdash \text{foldright} : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\alpha \text{ list}) \rightarrow \beta \rightarrow \beta \quad \Gamma' \vdash (+) : \text{int} \rightarrow \text{int} \rightarrow \text{int}}{\Gamma' \vdash \text{foldright } (+) : \text{int list} \rightarrow \text{int} \rightarrow \text{int}} \quad \Gamma' \vdash [2;3;4] : \text{int list}}{\Gamma' \vdash \text{foldright } (+) [2;3;4] : \text{int} \rightarrow \text{int}} \quad \Gamma' \vdash 0 : \text{int}}{\Gamma' \vdash \text{foldright } (+) [2;3;4] 0 : \text{int}}$$

Then we have:

$$\begin{array}{c}
 \frac{}{\Gamma' \vdash \text{lst} : \alpha \text{ list}} \quad \frac{}{\Gamma' \vdash [] : \alpha \text{ list}} \\
 \frac{}{\Gamma' \vdash \text{lst} = [] : \text{bool}} \quad \frac{}{\Gamma' \vdash z : \beta} \quad \frac{}{\Gamma' \vdash f \text{ (first lst) (foldright f (rest lst) z)} : \beta} \\
 \frac{}{\Gamma' \vdash \text{if lst} = [] \text{ then } z \text{ else } (f \text{ (first lst) (foldright f (rest lst) z))} : \beta} \\
 \Gamma \vdash \text{let rec foldright f lst z = if lst} = [] \text{ then } z \text{ else } (f \text{ (first lst) (foldright f (rest lst) z)) \\
 \text{in foldright (+) [2;3;4] 0} : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\alpha \text{ list}) \rightarrow \beta \rightarrow \beta
 \end{array}$$

4. Use the following encodings of **true**, **false** and **if** to define lambda terms **and**, **or**, **not**, **eq** which respectively return booleans corresponding to conjunction, disjunction, negation, and test for equality.

$$\mathbf{true} = \lambda a b. a \quad \mathbf{false} = \lambda a b. b \quad \mathbf{if} = \lambda c t e. c t e$$

Define functions which:

a. **and**

Answer: $\mathbf{and} = \lambda x y. x y \text{ false}$

or $\mathbf{and} = \lambda x y. x y x$

b. **or**:

Answer: $\mathbf{or} = \lambda x y. x \text{ true } y$

or $\mathbf{or} = \lambda x y. xxy$

c. **not**:

Answer: $\mathbf{not} = \lambda x. x \text{ false true}$

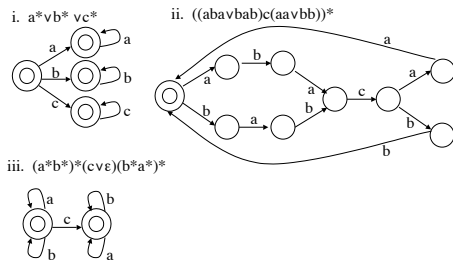
d. **eq**: $\mathbf{eq} = \lambda x y. x (y \text{ true false}) (y \text{ false true})$

5. For each of the regular expressions below (over the alphabet $\{a,b,c\}$), draw a nondeterministic finite state automaton that accepts exactly the same set of strings as the given regular expression.

i) $a^* \vee b^* \vee c^*$

ii) $((aba \vee bab)c(aa \vee bb))^*$

iii) $(a^*b^*)^*(c \vee \epsilon)(b^*a^*)^*$



Answer:

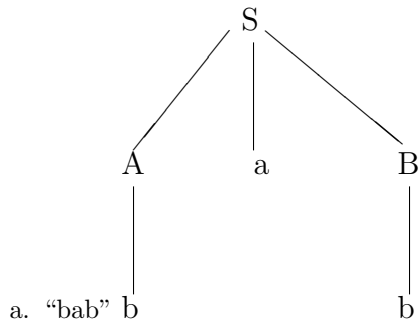
6. Consider the following ambiguous grammar (Capitals are nonterminals, lowercase are terminals):

$S \rightarrow A a B \mid B a A$

$A \rightarrow b \mid c$

$B \rightarrow a \mid b$ Give an example of a string for which this grammar has two different parse trees, and give their parse trees.

Answer:



- b. $S \rightarrow A a B \rightarrow b a B \rightarrow b a b$
 $S \rightarrow B a A \rightarrow b a A \rightarrow b a b$

7. Write a unambiguous grammar for regular expressions over the alphabet $\{a, b\}$. The Kleene star binds most tightly, followed by concatenation, and then choice. Here we will have concatenation and choice associate to the right. Write an Ocaml datatype corresponding to the tokens for parsing regular expressions, and one for capturing the abstract syntax trees corresponding to parses given by your grammar. Write a recursive descent parser for regular expressions, producing an option (**Some**) of an abstract syntax tree if a parse exists, or **None** otherwise.

Answer:

```
reg ::= a|b|reg ∨ reg|reg reg|reg*
Atom ::= a|b|(RegExp)
Star ::= Atom|Star *
Concat ::= Star|StarConcat
RegExp ::= Concat|Concat ∨ RegExp

type tokens = A_tk | B_tk | LParen | RParen | Star_tk | Or_tk
type atom = A | B | Paren of regexp
and star = Atom of atom | Star of star
and concat = StarAsConcat of star | Concat of (star * concat)
and regexp = ConcatAsRegExp of concat | Choice of (concat * regexp)

let rec mk_star (tree, tokens) =
  match tokens with Star_tk::more_toks -> mk_star (Star tree, more_toks)
  | _ -> (tree, tokens)

let rec atom tokens =
  match tokens with (A_tk::rem_toks) -> (Some A,rem_toks)
  | (B_tk::rem_toks) -> (Some B,rem_toks)
  | (LParen::more_toks) ->
    (match regexp more_toks with
     (Some tree, RParen::rem_toks) -> (Some(Paren tree),rem_toks)
     | (_, rem_toks) -> (None, rem_toks))
  | _ -> (None, tokens)
and star tokens =
  match atom tokens with
  (Some tree, rem_toks) ->
    (match mk_star (Atom tree, rem_toks) with
     (stree, toks) -> (Some stree, toks))
  | (None, rem_toks) -> (None, rem_toks)
and concat tokens =
  match star tokens with
  (Some tree, rem_toks) ->
    (match rem_toks with
     A_tk::_ ->
       (match concat rem_toks with (Some tree1, more_toks) ->
        (Some(Concat(tree,tree1)), more_toks)
        | (None, more_toks) -> (None, more_toks))
     | B_tk::_ ->
       (match concat rem_toks with (Some tree1, more_toks) ->
        (Some(Concat(tree,tree1)), more_toks)
        | (None, more_toks) -> (None, more_toks))
     | LParen::_ ->
       (match concat rem_toks with (Some tree1, more_toks) ->
        (Some(Concat(tree,tree1)), more_toks)
```

```

        | (None, more_toks) -> (None, more_toks))
    | _ -> (Some (StarAsConcat tree), rem_toks))
| (None, rem_toks) -> (None, rem_toks)
and regexp tokens =
match concat tokens with
(Some tree, more_tokens) ->
(match more_tokens with (Or_tk :: more_toks1) ->
(match regexp more_toks1 with
(Some tree1, rem_toks) -> (Some(Choice (tree, tree1)), rem_toks)
| (None, rem_toks) -> (None, rem_toks))
| _ -> (Some (ConcatAsRegExp tree), more_tokens))
| (None, rem_tokens) -> (None, rem_tokens)

```

8. Reduce the following expression: $(\lambda x \lambda y. yz)((\lambda x. xxx)(\lambda x. xx))$

a. Assuming Call by Name (Lazy Evaluation)

Answer: With Call by Name (Lazy Evaluation):

$(\lambda x. \lambda y. yz)((\lambda x. xxx)(\lambda x. xx)) - \beta \rightarrow (\lambda y. yz)$

b. Assuming Call by Value (Eager Evaluation)

Answer: With Call by Value (Eager Evaluation):

$(\lambda x. \lambda y. yz)((\lambda x. xxx)(\lambda x. xx)) - \beta \rightarrow$
 $(\lambda x. \lambda y. yz)((\lambda x. xx)(\lambda x. xx)(\lambda x. xx)) - \beta \rightarrow$
 $(\lambda x. \lambda y. yz)((\lambda x. xx)(\lambda x. xx)(\lambda x. xx)) - \beta \rightarrow$
 ... (the expression doesn't terminate)

9. a. Write the transition semantics rules for `if _ then _ else` and `repeat _ until _`. (A `repeat _ until _` executes the code in the body of the loop and then checks the condition, exiting if the condition is true.)

b. Assume we have an OCaml type `exp` for our language expressions, type `comm` for language commands with constructors `IfThenElse: exp * comm * comm`, `RepeatUntil: comm * exp * comm`, and `Seq: comm * comm`, a type value with constructors `TrueVal` and `FalseVal` corresponding to true and false, a type `mem` associating variables with values, and

```

type eval_exp_result = Inter_exp of (exp * mem) | Final of value
type eval_comm_result = Mid of (comm * mem) | Done of mem

```

Write Ocaml clauses for a function `eval_comm : (comm*mem) -> eval_comm_result` for the case of `IfThenElse` and `RepeatUntil`. You may assume that all other needed clauses of `eval_comm` have been defined, as well as the function `eval_exp : (exp*mem) -> eval_exp_result`.

Answer: The second part of this problem has a problem. There should have been a function `eval_exp: (env*exp) -> exp` that you could assume, Also, the function you are asked to write should have had the type `eval: (env*exp) -> (env*exp)` Let m represent the current state. `If_then_else` rules:

$$\frac{}{(\text{if true then } C_1 \text{ else } C_2 \text{ fi } , m) \rightarrow (C_1, m)}$$

$$\frac{}{(\text{if false then } C_1 \text{ else } C_2, m) \text{ fi } \rightarrow (C_2, m)}$$

$$\frac{}{(B, m) \rightarrow (B', m)}$$

$$\frac{}{(\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi } , m) \rightarrow (\text{if } B' \text{ then } C_1 \text{ else } C_2 \text{ fi } , m)}$$

$$\frac{}{(\text{repeat } C \text{ until } B, m) \rightarrow (C; \text{if } B \text{ then skip else (repeat } C \text{ until } B) \text{ fi } , m)}$$

```

let rec eval_comm (comm, env) =
  match comm with
  . . .
  | IfThenElse (TrueVal, thenclause, elseclause) -> Mid (thenclause, env)
  | IfThenElse (FalseVal, thenclause, elseclause) -> Mid (elseclause, env)
  | IfThenElse (b, thenclause, elseclause) ->
    (match eval_exp (b, env) with Final v ->
      (IfThenElse (v, thenclause, elseclause), env)
    | Inter_exp (new_b, env') ->
      (IfThenElse (new_b, thenclause, elseclause), env) )
  . . .
  | RepeatUntil(c,b) -> (Seq (c, IfThenElse (b, Skip, RepeatUntil(c,b))), env)

```

10. Assume you are given the OCaml types `exp`, `bool_exp` and `comm` with (partially given) type definitions:

```

type comm = ... | If of (bool_exp * comm * comm) | ...
type bool_exp = True_exp | False_exp | ...

```

where the constructor `If` is for the abstract syntax of an `if_then_else_construct`. Also assume you have a type `mem` of memory associating values to identifiers, where values are just integers (`int`). Further assume you are given a function `eval_bool: (mem * bool_exp) -> bool` for evaluating boolean expressions. Write the OCaml code for the clause of `eval_comm: (mem * comm) -> mem` that implements the following natural semantics rules for the evaluation of `if_then_else_commands`:

$$\frac{\langle m, b \rangle \Downarrow \text{true} \quad \langle m, C_1 \rangle \Downarrow m'}{\langle m, \text{if } b \text{ then } C_1 \text{ else } C_2 \rangle \Downarrow m'} \quad \frac{\langle m, b \rangle \Downarrow \text{false} \quad \langle m, C_2 \rangle \Downarrow m''}{\langle m, \text{if } b \text{ then } C_1 \text{ else } C_2 \rangle \Downarrow m''}$$

Answer:

```

let rec eval_comm (env, comm) =
  match comm with . . .
  | If (bexp,c1,c2) ->
    (match eval_bool (env, bexp) with True_exp -> eval_comm (env, c1)
    | False_exp -> eval_com (env, c2))
  . . .

```

11. Recollect that we may implement thunks as follows:

```
type 'a thunk_type = Value of 'a | Susp of (unit -> 'a);;

let delay f =
  let thunk = ref (Susp f) in
  fun () -> match (!thunk) with
  | Value a -> a
  | Susp f -> let result = f () in (thunk := (Value result); result );;

let force f = f ();;
```

a. What are the types of `delay` and `force`?

Answer: `delay : (unit -> 'a) -> unit -> 'a` `force : (unit -> 'a) -> 'a`

b. Using the above constructions, but not the `Lazy` module from `Ocaml`, implement in `Ocaml` a type of `'a lazy_list`.

Answer: `type 'a lazy_list = Nil | Cons of ('a * (unit -> 'a lazy_list));;`

c. Implement the function `take : int -> 'a lazy_list -> 'a list` which returns the first n elements of the lazy list.

Answer: `let rec take n llst =
 match n,llst with
 | _,Nil -> []
 | 0,_ -> []
 | _,(Cons(x,xs)) -> x :: take (n-1) (force xs);;`

d. Create an infinite lazy list whose elements are the successive even numbers starting with 0.

Answer: `let rec map f l =
 match l with Nil -> Nil
 | Cons (x,xs) -> Cons (f x, delay (fun () -> map f (force xs)))
let rec evens = Cons(0, (fun () -> map ((+) 2) evens))`

12. Write a function `dividek n lst k`, using Continuation Passing Style (CPS), that divides n successively by every number in the list, starting from the *last* element in the list. If a zero is encountered in the list, the function should pass 0 to `k` immediately, *without doing any divisions*.

```
# dividek 6 [1;3;2] report;;
Result: 1
- : unit = ()
```

Answer:

```

let rec dividek n lst k =
  let rec dividek_aux n list inck =
    match list with
    | [] -> inck n
    | 0::xs -> k 0
    | x::xs -> dividek_aux n xs (fun r -> inck (r/x))
  in dividek_aux n lst k

```

13. Use the method dispatch method discussed in class to implement a bank account class. The bank account should have instance variables of `name`, `account_number`, a class method of `mk_account` taking a name and initial balance as arguments, and instance methods of `deposit`, `withdraw`, and `get_balance`. Each account should get its own account number.

Answer:

```

let mk_account =
  let new_account_number =
    let account_number = ref 1 in
    fun () -> (let acn = !account_number
              in (account_number := acn + 1; acn))
  in
  fun name init_balance ->
    let name = ref name in
    let balance = ref init_balance in
    let account_number = ref (new_account_number()) in
    ((name, account_number),
     (fun meth ->
      match meth with "deposit" ->
        fun n -> (balance := !balance + n; !balance)
      | "withdrawal" ->
        fun n -> (balance := !balance - n; !balance)
      | "get_balance" -> fun _ -> !balance
      | _ -> failwith "No such method")));;

```