

CS 421, Fall 2008

Sample Final Questions & Answers

You should review the questions from the sample midterm exams, the real midterm exams, and the homework, as well as these question.

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 = ((q = 0) || 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. 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
```

4. a. Give the types for following functions (you don't have to derive them). Assume that `lst` is an `int list`.

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

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

Answer:

```
first : int list → int
rest : int list → int list
```

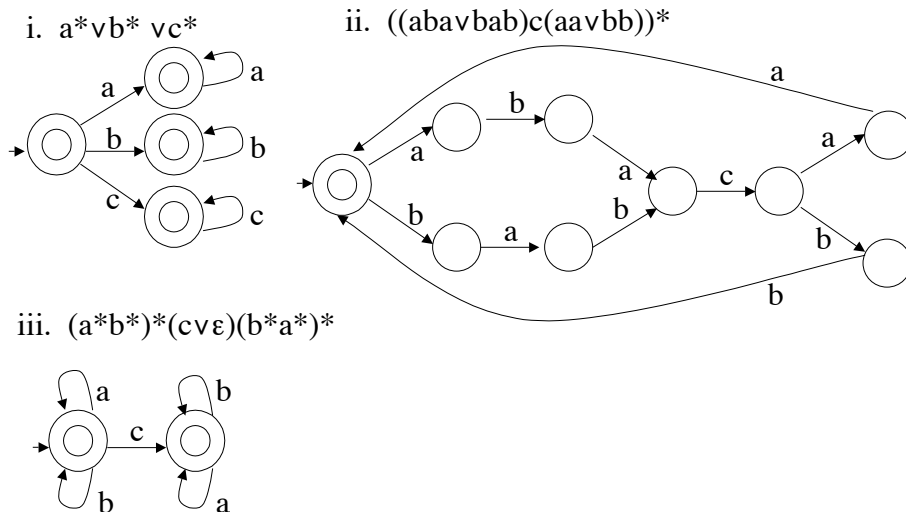
- b. Use these types (i.e., start in an environment with these identifiers bound to these types) to give a type derivation for:

```
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 $\Gamma = \{ \text{first} : \text{int list} \rightarrow \text{int}, \text{rest} : \text{int list} \rightarrow \text{int list} \}$,
 and $\Gamma' = \Gamma \cup \{ \text{foldright} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow (\text{int list}) \rightarrow \text{int} \rightarrow \text{int}, \}$.
 $f : \text{int} \rightarrow \text{int} \rightarrow \text{int}$,
 $\text{lst} : \text{int list}, z : \text{int}$



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

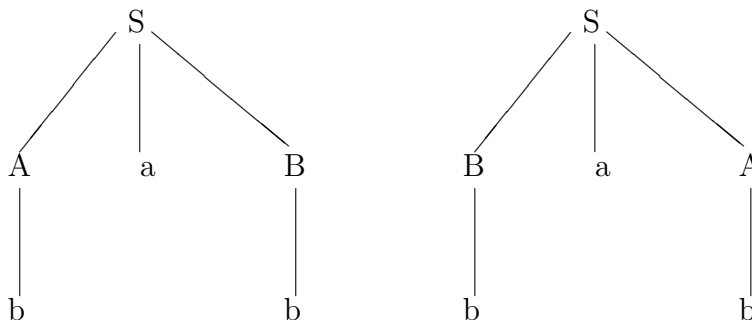
$$S \rightarrow A a B \mid B a A$$

$$A \rightarrow b \mid c$$

$$B \rightarrow a \mid b$$

Give an example of a string for which this grammar has two different parse trees, and give their parse trees.

Answer: A string with two parses is “bab”, and its parse trees are:



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:

$$reg ::= a \mid b \mid \varepsilon \mid (reg) \mid reg \vee reg \mid reg \text{ reg} \mid reg^*$$
$$Atom ::= a \mid b \mid \varepsilon \mid (RegExp)$$
$$Star ::= Atom \mid Star^*$$
$$Concat ::= Star \mid Star \text{ Concat}$$
$$RegExp ::= Concat \mid Concat \vee RegExp$$

```
type tokens = A_tk | B_tk | Epsilon_tk | LParen | RParen | Star_tk | Or_tk
type atom = A | B | | Epsilon | 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)
  | (Epsilon_tk::rem_toks) -> (Some Epsilon,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. 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: $\mathbf{and} = \lambda x y. x y \mathbf{false}$

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

b. **or**:

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

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

c. **not**:

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

d. **eq**:

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

9. 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)

c. To full $\alpha\beta$ -normal form.

Answer: Since lazy evaluation yielded an $\alpha\beta$ -normal form. we may use its reduction: $(\lambda x. \lambda y. yz)((\lambda x. xxx)(\lambda x. xx)) - \beta \rightarrow (\lambda y. yz)$

10. 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.)

Answer: Let m represent the current state. `If.then.else` rules:

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

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

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

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

- b. Assume we have an OCaml type `exp` with constructors `True` and `False` corresponding to true and false, a type `mem` of memory associating variables (represented by strings) with values, a type `exp` for our language expressions, a type `comm` for language commands with constructors `IfThenElse` of `exp * comm * comm`, `RepeatUntil` of `comm * exp`, and `Seq`: `comm * comm`, and types

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

Further suppose we have a function `eval_exp : (mem * exp) -> eval_exp_result` that returns the result of one step of evaluation of an expression.

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: `let rec eval_comm (comm, mem) =`
`match comm with`
`. . .`

```

| IfThenElse (True, thenclause, elseclause) -> Mid (thenclause, mem)
| IfThenElse (False, thenclause, elseclause) -> Mid (elseclause, mem)
| IfThenElse (b, thenclause, elseclause) ->
  (match eval_exp (b, mem) with Final e ->
    Mid (IfThenElse (e, thenclause, elseclause), mem)
  | Inter_exp (new_b, mem') ->
    Mid (IfThenElse (new_b, thenclause, elseclause), mem) )
. . .
| RepeatUntil(c,b) ->
  Mid (Seq (c, IfThenElse (b, Skip, RepeatUntil(c,b))), mem)

```

11. 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 (mem, comm) =
  match comm with . . .
| If (bexp,c1,c2) ->
  (match eval_bool (mem, bexp) with true -> eval_comm (mem, c1)
  | false -> eval_com (mem, c2))
. . .
```

12. Using the natural semantics rules given in class, give a proof that, starting with a memory that maps `x` to 5 and `y` to 3, `if x = y then z := x else z := x + y` evaluates to a memory where `x` maps to 5, `y` maps to 3. and `z` maps to 8.

Answer: Let $m = \{x \mapsto 5; y \mapsto 3\}$.

$$\frac{\frac{\overline{\langle m, x \rangle \Downarrow 5} \quad \overline{\langle m, y \rangle \Downarrow 3} \quad (5 = 3) = \text{false}}{\langle \{x \mapsto 5; y \mapsto 3\}, x = y \rangle \Downarrow \text{false}} \quad \frac{\overline{\langle m, x \rangle \Downarrow 5} \quad \overline{\langle m, y \rangle \Downarrow 3} \quad 5 + 3 = 8}{\langle \{x \mapsto 5; y \mapsto 3\}, z := x + y \rangle \Downarrow \langle \{x \mapsto 5; y \mapsto 3; z \mapsto 8\}}}{\langle \{x \mapsto 5; y \mapsto 3\}, \text{if } x = y \text{ then } z := x \text{ else } z := x + y \rangle \Downarrow \langle \{x \mapsto 5; y \mapsto 3; z \mapsto 8\}}$$

13. Give a proof in Floyd-Hoare logic of the partial correctness assertion:

$$\{\text{True}\} y := w; \text{ if } x = y \text{ then } z := x \text{ else } z := y \{z = w\}$$

Answer: Because this proof tree is rather wide, we shall break it up into pieces.

Let *Tree1* =

$$\frac{((y = w) \ \& \ (x = y)) \Rightarrow (x = w) \quad \overline{\{x = w\} z := x \{z = w\}}}{\{(y = w) \ \& \ (x = y)\} z := x \{z = w\}}$$

Let *Tree2* =

$$\frac{\{(y = w) \ \& \ (x \neq y)\} \Rightarrow (y = w) \quad \overline{\{y = w\} z := y \{z = w\}}}{\{(y = w) \ \& \ (x \neq y)\} z := y \{z = w\}}$$

Then the main proof tree is

$$\frac{\text{True} \Rightarrow (w = w) \quad \overline{\{w = w\} y := w \{y = w\}} \quad \frac{\text{Tree1} \quad \text{Tree2}}{\{y = w\} \text{ if } x = y \text{ then } z := x \text{ else } z := y \{z = w\}}}{\{\text{True}\} y := w; \text{ if } x = y \text{ then } z := x \text{ else } z := y \{z = w\}}$$

14. What should the Floyd-Hoare logic rule for **repeat** *C* **until** *B* be? The code causes *C* to be executed, and then, if *B* is true it completes, and otherwise it does **repeat** *C* **until** *B* again.

Answer:

$$\frac{\{Q\} \text{ repeat } C \text{ until } B \{P \wedge B\}}{\{Q \vee (P \wedge (B))\} C \{P\}}$$

But I would accept the weaker

$$\frac{\{P\} \text{ repeat } C \text{ until } B \{P \wedge B\}}{\{P\} C \{P\}}$$