Solutions for Sample Questions for Midterm 2 (CS 421 Spring 2024)

Some of these questions may be reused for the exam.

1. Put the following function in full continuation passing style:
   
   ```ocaml
   let rec sum_odd n = if n <= 0 then 0 else ((2 * n) – 1) + sum_odd (n - 1);;
   ```

   Use addk, subk, mulk, leqk, for the CPS forms of the primitive operations (+, -, *, <=). All other procedure calls and constructs must be put in CPS.

   **Solution:**
   ```ocaml
   let add_k a b k = k(a + b)
   let minus_k a b k = k(a - b)
   let times_k a b k = k(a * b)
   let leq_k a b k = k(a <= b)
   let rec sum_odd_k n k =
     leq_k n 0 (fun b -> if b then k 0
     else minus_k n 1
     (fun d -> sum_odd_k d
     (fun r -> times_k 2 n
     (fun t -> minus_k t 1
     (fun m -> add_k m r k ))))));
   ```

2. Given the following OCAML datatype:

   ```ocaml
type int_seq = Null | Snoc of (int_seq * int)
   ```

   write a tail-recursive function in OCAML `all_pos : int_seq -> bool` that returns `true` if every integer in the input `int_seq` to which `all_pos` is applied is strictly greater than 0 and `false` otherwise. Thus `all_pos` of `(Snoc(Snoc(Snoc(Null, 3), 5), 7))` should return `true`, but `all_pos` of `(Snoc(Null, -1))` and `all_pos` of `(Snoc(Snoc(Null, 3), 0))` should both return `false`.

   **Solution:**
   ```ocaml
   let rec all_pos s =
     (match s with Null -> true
     | Snoc(seq, x) -> if x <= 0 then false else all_pos seq);;
   ```

3. Write the definition of an OCAML variant type `reg_exp` to express abstract syntax trees for regular expressions over a base character set of booleans. Thus, a boolean is a `reg_exp`, epsilon is a `reg_exp`, a parenthesized `reg_exp` is a `reg_exp`, the concatenation of two `reg_exp`’s is a `reg_exp`, the “choice” of two `reg_exp`’s is a `reg_exp`, and the Kleene star of a `reg_exp` is a `reg_exp`.

   **Solution:**
   ```ocaml
   type reg_exp =
     Char of bool
     | Epsilon
   ```
Paren of reg_exp
Concat of (reg_exp * reg_exp)
Choice of (reg_exp * reg_exp)
Kleene_star of reg_exp

4. Given the following rules for CPS transformation:

\[
\begin{align*}
[[x]] & \Rightarrow K x \\
[[c]] & \Rightarrow K c \\
[[\text{let } x = e1 \text{ in } e2]] & \Rightarrow [[e1]] (\text{FN } x \Rightarrow [[e2]] K) \\
[[e1 \oplus e2]] & \Rightarrow [[e2]] (\text{FN } a \Rightarrow [[e1]] (\text{FN } b \Rightarrow K (b \oplus a)))
\end{align*}
\]

where e1 and e2 are OCaml expressions, K is any continuation, x is a variable and c is a constant, give the step-by-step transformation of

\[
[[\text{let } x = 2 + 3 \text{ in } x - 4]] \text{ REPORT } k
\]

Solution:

\[
[[\text{let } x = 2 + 3 \text{ in } x - 4]] \text{ REPORT } k \Rightarrow
[[2 + 3]] (\text{FN } x \Rightarrow [[x - 4]] \text{ REPORT } k) \Rightarrow
[[2 + 3]] (\text{FN } x \Rightarrow [[4]] (\text{FN } n \Rightarrow [[x]] (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) \Rightarrow
[[2 + 3]] (\text{FN } x \Rightarrow [[4]] (\text{FN } n \Rightarrow (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) x) \Rightarrow
[[2 + 3]] (\text{FN } x \Rightarrow (\text{FN } n \Rightarrow (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) x) 4) \Rightarrow
[[3]] (\text{FN } u \Rightarrow [[2]] (\text{FN } v \Rightarrow (\text{FN } x \Rightarrow (\text{FN } n \Rightarrow (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) x) 4) (v + u)) \Rightarrow
[[3]] (\text{FN } u \Rightarrow (\text{FN } v \Rightarrow (\text{FN } x \Rightarrow (\text{FN } n \Rightarrow (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) x) 4) (v + u)) 2) \Rightarrow
(\text{FN } u \Rightarrow (\text{FN } v \Rightarrow (\text{FN } x \Rightarrow (\text{FN } n \Rightarrow (\text{FN } m \Rightarrow \text{REPORT } k (m - n)))) x) 4) (v + u)) 2) 3
\]

5. Review and be able to write any give clause of \texttt{cps_exp} from MP5. On the exam, you would be given all the information you were given in MP5.

Solution:

(* Problem 5 *)

let rec \texttt{cps\_exp} \texttt{e} \texttt{k} =
match \texttt{e} with
(*[[x]]k = k x*)
\texttt{VarExp} \texttt{x} \Rightarrow \texttt{VarCPS} (k, x)
(*[[c]]k = k x*)
| \texttt{ConstExp} \texttt{n} \Rightarrow \texttt{ConstCPS} (k, n)
(*[[\sim \ e]]k = [[e]] _ (fun \texttt{r} -> k (~ \texttt{r}) *)
| \texttt{MonOpAppExp} \texttt{(m, e)} \Rightarrow
let \texttt{r} = \texttt{freshFor} (\texttt{freeVarsInContCPS} \texttt{k})
in \texttt{cps\_exp} \texttt{e} (\texttt{FnContCPS} (r, \texttt{MonOpAppCPS} (k, m, r)))
(*[[\texttt{e1} + \texttt{e2}]])k = [[\texttt{e2}]] _ (fun \texttt{s} -> [[\texttt{e1}]] _ (fun \texttt{r} -> k (\texttt{r} + \texttt{s})*)
| \texttt{BinOpAppExp} \texttt{(b, e1, e2)} \Rightarrow
let \texttt{v2} = \texttt{freshFor} (\texttt{freeVarsInContCPS} \texttt{k} @ \texttt{freeVarsInExp} \texttt{e1}) in
let \texttt{v1} = \texttt{freshFor} (\texttt{v2 :: (freeVarsInContCPS} \texttt{k}) in
let \texttt{e2CPS} =
\texttt{cps\_exp} \texttt{e1} (\texttt{FnContCPS} (\texttt{v1}, \texttt{BinOpAppCPS}(k, b, v1, v2))) in
cps_exp e2 (FnContCPS (v2, e2CPS))
(*[[if e1 then e2 else e3]]k = [[e1]]_ (fun r -> if r then [[e2]]k else [[e3]]k)*)
| IfExp (e1,e2,e3) ->
  let r = freshFor (freeVarsInContCPS k @
      freeVarsInExp e2 @ freeVarsInExp e3) in
  let e2cps = cps_exp e2 k in
  let e3cps = cps_exp e3 k in
  cps_exp e1 (FnContCPS (r, IfCPS (r, e2cps, e3cps)))

(*[[e1 e2]]k = [[e2]]_ fun v2 -> [[e1]]_ fun v1 -> k (v1 v2)*)
| AppExp (e1,e2) ->
  let v2 = freshFor (freeVarsInContCPS k @ freeVarsInExp e1) in
  let v1 = freshFor (v2 :: freeVarsInContCPS k) in
  let e1cps = cps_exp e1 (FnContCPS (v1, AppCPS (k, v1, v2))) in
  cps_exp e2 (FnContCPS (v2, e1cps))

(*[[fun x -> e]]k = k(fnk x kx -> [[e]]kx)*)
| FunExp (x,e) ->
  let ecps = cps_exp e (ContVarCPS Kvar) in
  FunCPS (k, x, Kvar, ecps)

(*[[let x = e1 in e2]]k = [[e1]]_ fun x -> [[e2]]k *)
| LetInExp (x,e1,e2) ->
  let e2cps = cps_exp e2 k in
  let fx = FnContCPS (x, e2cps) in
  cps_exp e1 fx

(*[[let rec f x = e1 in e2]]k = (FN f -> [[e2]]_k)(FIX f. FUN x -> fn kx => [[e1]]kx)*)
| LetRecInExp(f,x,e1,e2) ->
  let e1cps = cps_exp e1 (ContVarCPS Kvar) in
  let e2cps = cps_exp e2 k in
  FixCPS (FnContCPS (f, e2cps), f, x, Kvar, e1cps)

6. Given a polymorphic type derivation for \{\} |- let id = fun x -> x in id id true : bool
Solution:
Let $\Gamma = \{id : \forall \ 'a. \ 'a -> 'a\}$

<table>
<thead>
<tr>
<th>Var</th>
<th>Var</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inc: 'a -&gt; bool -&gt; bool</td>
<td>Inc: 'a -&gt; bool</td>
<td>Inc: 'a -&gt; bool</td>
</tr>
<tr>
<td>$\Gamma \vdash id : (bool -&gt; bool) -&gt; bool -&gt; bool$</td>
<td>$\Gamma \vdash id : bool -&gt; bool$</td>
<td>$\Gamma \vdash true : bool$</td>
</tr>
<tr>
<td>${x : 'a} \mid - \mid x : 'a$</td>
<td>$\Gamma \mid - \mid id : bool -&gt; bool$</td>
<td>$\Gamma \mid - \mid true : bool$</td>
</tr>
<tr>
<td>${} \mid - \mid fun x -&gt; x$</td>
<td>${} \mid - \mid \forall \ 'a. \ 'a -&gt; 'a$</td>
<td>${} \mid - \mid id id true$</td>
</tr>
<tr>
<td>Let</td>
<td>Let</td>
<td>Let</td>
</tr>
<tr>
<td>Inc: 'a -&gt; bool -&gt; bool</td>
<td>Inc: 'a -&gt; bool -&gt; bool</td>
<td>Inc: 'a -&gt; bool</td>
</tr>
<tr>
<td>$\Gamma \vdash id = fun x -&gt; x in id id true : bool$</td>
<td>$\Gamma \vdash id = fun x -&gt; x in id id true : bool$</td>
<td>$\Gamma \vdash id = fun x -&gt; x in id id true : bool$</td>
</tr>
</tbody>
</table>