CS 421, Fall 2012

Sample Final Questions

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 to 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.

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.

3. Write a function `dividek: int -> int list -> (int -> 'a) -> 'a`, that is in full 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*. You should use

```
# let divk x y k = k(x/y);;
val divk : int -> int -> (int -> 'a) -> 'a = <fun>
```

   for the divisions. An example use of `dividek` would be

```
# let report n = print_string "Result: "; print_int n; print_string "\n";;
val report : int -> unit = <fun>
# dividek 6 [1;3;2] report;;
Result: 1
- : unit = ()
```

4. a. Give most general (polymorphic) 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;;
```

   b. Use these types (i.e., start in an environment with these identifiers bound to these types) to give a polymorphic 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
```

You should use the following types: `[]` : $\forall$`'a. 'a list`, and `(::)` : $\forall$`'a.'a` $\rightarrow$ `'a list` $\rightarrow$ `'a list`. Assume that the Relation Rule is extended to allow equality at all types.

5. For each of the regular expressions below (over the alphabet {a,b,c}), give a right regular gramar that derives exactly the same set of strings as the set of strings generated by the given regular expression.

   i) a*∨b*∨c*

   ii) ((aba∨bab)c(aa∨bb))*

   iii) (a*b*)*(c∨ϵ)(b*a*)*

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

   S → A a B | B a A
   A → b | c
   B → a | b

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

   b. Disambiguate this grammar.

7. Write a unambiguous grammar for regular expressions over the alphabet $\{\mathbf{a}, \mathbf{b}\}$. 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, taking a list of tokens and producing an option (**Some**) of an abstract syntax tree if a parse for the whole exists, or **None** otherwise.

8. 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 `bexp` with constructors `True` and `False` corresponding to true and false, and other constructors representing the syntax trees of non-value boolean expressions. Futher assume we have a type `mem` of memory associating variables (represented by strings) with values, a type `exp` for integer expressions in our language, a type `comm` for language commands with constructors including `IfThenElse of bexp * comm * comm`, `RepeatUntil of comm * bexp`, and `Seq: comm * comm`, and type

```
type eval_comm_result = Mid of (comm * mem) | Done of mem
```

Further suppose we have a function `eval_bexp : (bexp * mem) -> (bexp * mem)` 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_bexp: (bexp*mem) -> (bexp*mem)`.

9. 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 intergers (`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:

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

10. Using the natural semanitics 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.

11. Prove that $\lambda x.x(\lambda z.zxz)$ is α-equivalent $\lambda z.z(\lambda x.xzx)$. You should label every use of α-conversion and congruence.

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

    a. Assuming Call by Name (Lazy Evaluation)
    b. Assuming Call by Value (Eager Evaluation)
    c. To full $\alpha\beta$-normal form.

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

$$\{\texttt{True}\}\ \texttt{y := w; if x = y the z := x else z := y}\ \{\texttt{z = w}\}$$

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.