

Sample Final Questions

1. Write a function `get_primes : int -> int list` that returns the list of integers less than or equal the input. You may use the built-in functions `div` and `rem`. 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. a. Give the types for following expressions (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;;
```

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))`
 - c. `foldright (+) [2;3;4] 0`
4. Reduce the following expression: $(\lambda x \lambda y. yz)((\lambda x. xxx)(\lambda x. xx))$
 - a. Assuming Call by Name (Eager Evaluation)
 - b. Assuming Call by Value (Lazy Evaluation)
 5. Using the following encodings of **true**, **false** and **if** to define lambda terms **and**, **or**, **not**, **eq** which return, respectively, 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**
- b. **or**:
- c. **not**:
- d. **eq**:

6. Give the value of the following expression:

```
let a = 3
  in let p x = x * a
      in let a = 5
          in a + (p 7);;
```

- a. assuming *static scope* (a.k.a. *lexical scope*):
- b. assuming *dynamic scope*:
7. a. 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) $(a^*b^*)^*(c \vee \epsilon)(b^*a^*)^*$
- b. For each nondeterministic finite state automaton given for part a, give a corresponding deterministic finite state automaton that accepts exactly the same set of strings.
8. 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. Give the two leftmost derivations corresponding to those parse trees.
9. 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).
10. 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.) Assuming a datatype for expressions in Ocaml has been declared, with constructors `IfThenElse: exp -> exp -> exp -> exp` and `RepeatUntil: exp -> exp -> exp`, write Ocsml clauses for a function `eval: exp -> value`. You may assume that all other needed clauses of `eval` have been defined, and the `TrueVal` and `FalseVal` are the values corresponding to true and false.

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`?
  - b. Using the above constructions, but not the `Lazy` module from Ocaml, implement in Ocaml a type of `'a lazy_list`.
  - c. Implement the function `take: int -> 'a lazy_list -> 'a list` which returns the first  $n$  elements of the lazy list.
  - d. Create an infinite lazy list whose elements are the successive even numbers starting with 0.
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 = ()
```

13. Assuming Ocaml had `callcc` and `throw` implemented as discussed in class (and as implemented in SML/NJ), what would be the result of the following:

- a. `callcc (fun k -> throw k (10 + 15 + 20 + 25));;`
- b. `callcc (fun k -> throw k (10 + 15) + 20 + 25);;`
- c. `callcc (fun k -> (10 + 15 + 20 + 25));;`

- d. Using `callcc` and `throw`, reimplement the following code for `find : ('a -> bool) -> 'a list -> 'a option` without using exceptions:

```
exception NotPresent
let find p lst =
 let rec find_aux p lst = match lst with [] -> raise NotPresent
 | x :: xs -> if p x then x else find_aux p xs
 in try Some (find_aux p lst) with NotPresent -> None
```