

CS 421, Fall 2006

Sample Final Questions

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.
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 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;;
```

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`

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**
- b. **or**:
- c. **not**:
- d. **eq**:

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) $a^*vb^*vc^*$

- ii) $((aba \vee bab)c(aa \vee bb))^*$
- iii) $(a^*b^*)^*(c \vee \epsilon)(b^*a^*)^*$

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
```
- Give an example of a string for which this grammar has two different parse trees, and give their parse trees.
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, producing an option (**Some**) of an abstract syntax tree if a parse exists, or **None** otherwise.
8. 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)
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` and `RepeatUntil: comm * exp * 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 (exp * mem) | Done of mem
```
- Write Ocsml clauses for a function `eval_comm : (exp*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`.
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 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:

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

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 ();;
```

- What are the types of `delay` and `force`?
  - Using the above constructions, but not the `Lazy` module from OCaml, implement in OCaml a type of `'a lazy_list`.
  - Implement the function `take: int -> 'a lazy_list -> 'a list` which returns the first  $n$  elements of the lazy list.
  - 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. 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.