Continuation Passing Style, Free Variables, `freeVarsInExp`

The purpose of this question is to help the student master:

- Deconstructing data structures built from algebraic data types

Open workspace

Using VSCode - General Instructions

▶ PicoML type information

Free Variables in a PicoML Expression

All points mentioned here are approximate and subject to scaling in any use of this problem on an assessment containing it. The points on the parts of the problem are out of the total number of points for the problem

Throughout this assignment you may use any library functions you wish.

Problem

▼ Overview and Background Details for `freeVarsInExp`

(16 pts total) A free variable in an expression is a variable that has an occurrence that isn't bound in that expression. In our setting, free variables are the variables that had to be given a value previously for the expression to be able to be evaluated. As an example, in (`let x = y in fun s -> a x s`) the variables `a` and `y` are free but `x` and `s` are not. Notice that to understand the free variables in the previous example, we also need to know for the inner declaration both the free variables of the expression in it, but also the variables that are being bound by it.

Write a function `freeVarsInExp : exp -> string list` that calculates the names of the free variables of an expression (where we represent sets via lists). The grader will cope with answers that have duplicate entries or the result list in a different order than our reference solution.

To assist you in writing this function, we have broken the problem down into groups of similar cases. We also give the precise mathematical definition (in cases) for a function ϕ calculating the free variables of an expression e. These definitions can be found in the linked pdf above.

In `common.ml`, we have supplied you with a related pair of functions `freeVarsInContCPS : cps_cont -> string list` and `freeVarsInExpCPS : exp_cps -> string list` for calculating the free variables in a continuation and CPS-transformed expression respectively. You should feel free to examine these definitions for inspiration in writing the code for this problem.

1. ▼ Description of Variable Case
   (1 pt.) We can define a function $\varphi(e)$ that calculates the free variables of an expression, where the expression is a variable v, by

   $$\varphi(\text{v}) = \{v\}$$

   The function `freeVarsInExp` should behave in a similar manner, returning the singleton name of the variable for a variable. Write the appropriate clause for `freeVarsInExp` to return the free variables of expressions that are variables.

   ```
   # let rec freeVarsInExp = ... ;;
   val freeVarsInExp : Common.exp -> string list = <fun>
   # freeVarsInExp (VarExp "x");;
       - : string list = ["x"]
   ```

2. ▼ Description of Constant Case
   (1 pt.) We can define the value of the function $\varphi(e)$ that calculates the free variables of an expression, where the expression is a a constant c by

   $$\varphi(c) = \{\ \}$$

   The function `freeVarsInExp` should behave in a similar manner, returning no names for a constant. Write the appropriate clause for `freeVarsInExp` to return the free variables of expressions that are constants.

   ```
   # let rec freeVarsInExp = ... ;;
   val freeVarsInExp : Common.exp -> string list = <fun>
   # List.length (freeVarsInExp (ConstExp (StringConst "hi")));;
       - : int = 0
   ```

3. ▼ Description of Unary Operator (Monadic Operator) Case
   (2 pts.) The set of free variables of the use of a unary operator is just thef the free variables the immediate subexpression.

   $$\varphi(\oplus e) = \varphi(e) \qquad \textbf{For unary operator } \oplus$$

   Write the clause for `freeVarsInExp` for expressions that are top-most the use of a unary operator.

   ```
   # freeVarsInExp (MonOpAppExp(IntNegOp, VarExp "x"));;
   - : string list = ["x"]
   ```

4. ▼ Description of Binary Operator Case
   (2 pts.) The set of free variables of the use of a binary operator is just the union of the free variables of the two immediate subexpressions.

   $$\varphi(e_1 \oplus e_2) = \varphi(e_1) \cup \varphi(e_2) \qquad \textbf{For binary operator } \oplus$$

   Write the clause for `freeVarsInExp` for the use of a binary operator.

   ```
   # freeVarsInExp (BinOpAppExp(IntTimesOp, VarExp "x", ConstExp(IntConst 3)));;
   - : string list = ["x"]
   ```

5. ▼ Description of If-Then-Else Case

(2 pts.) The set of free variables of an expression that is at the top-most level an if-then-else expression is just the union of the free variables of the three immediate subexpressions.

$$\varphi(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \varphi(e_1) \cup \varphi(e_2) \cup \varphi(e_3)$$

Write the clause for `freeVarsInExp` for expressions that are top-most an if-then-else expression. Please remember that it is alright for you code to return the variables in a different order.

```
# freeVarsInExp (IfExp(ConstExp (BoolConst true), VarExp "x", VarExp "y"));;
- : string list = ["x"; "y"]
```

6. ▼ Description of Application Case
   (2 pts.) The set of free variables of an expression that is at the top-most level an application of one expression to another is just the union of the free variables of the two immediate subexpressions.

$$\varphi(e_1\ e_2) = \varphi(e_1) \cup \varphi(e_2)$$

   Write the clauses for `freeVarsInExp` for expressions that are top-most an application of one expression to another.

```
# freeVarsInExp (AppExp(VarExp "f", VarExp "y"));;
- : string list = ["f"; "y"]
```

7. ▼ Description of Function Expression Case
   (3 pts.) The free variables of a function expression are all the free variables in the body of the expression except the variable that is the formal parameter. Any occurrence of that variable in the body of the function is bound by the formal parameter, and not free.

$$\varphi(\texttt{fun } x \texttt{ -> } e) = \varphi(e) - \{x\}$$

   Add clauses to `freeVarsInExp` to compute the free variables of a function expression.

```
# freeVarsInExp (FunExp("x", VarExp "x"));;
- : string list = []
```

   Note: You can implement set subtraction using the library function `List.filter : ('a -> bool) -> 'a list -> 'a list`.

8. ▼ Description of Let-In Case
   (3 pts.) The free variables of a `let`-expression are restricted by the variable being locally declared, and hence bound. In `let x = e_1 in e_2` the x in the `let` part binds any occurrence of x in e_2, but not in e_1.

$$\varphi(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \varphi(e_1) \cup (\varphi(e_2) - \{x\})$$

   Add the clause to `freeVarsInExp` to compute the free variables of `let`-expressions.

```
# freeVarsInExp (LetInExp("x", VarExp "y", VarExp "x"));;
- : string list = ["y"]
```

Extra Credit

9. ▼ Description of Let-Rec_in Case

(4 pts) The most complicated case for computing the free variables of an expression is that of a `let rec`-expression. In `let rec`-expressions, there are two bindings taking place, and they have two different scopes. In `let rec f x = e_1 in e_2`, the f binds all the occurrences of f in both e_1 and e_2, but the x only binds occurrences of x in e_1; if x is a free variable of e_2 it will be a free variable of `let rec f x = e_1 in e_2)`.

$$\varphi(\texttt{let rec } f\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = (\varphi(e_1) - \{f, x\}) \cup (\varphi(e_2) - \{f\})$$

Write the clause for `freeVarsInExp` for `let rec`-expressions.

```
# freeVarsInExp (LetRecInExp("f","x",AppExp(VarExp "f",VarExp "x"),
                        AppExp(VarExp "f",VarExp "y")));;
- : string list = ["y"]
```

Save & Grade *Unlimited attempts* | Save only

Correct answer