Implementing the Continuation Passing Style (CPS) Transformation `cps_exp`

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

- Constructing data structures from algebraic data types
- Deconstructing data structures built from algebraic data types
- Implementing continuation passing style transformations

Open workspace

Using VSCode - General Instructions

▶ PicoML type information

Implementing Transforming PicoML into CPS

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

▶ Overview and Background Details for `cps_exp`

Problem

1. ▼ Description of Variable Case
   The CPS transformation of a variable just applies the continuation to the variable, since during execution, when this point in the code is reached, variables are already fully evaluated (except for being looked up).

   $$[[v]]_\kappa = \kappa\, v$$

   The code for the function `cps_exp` should behave in a similar manner, creating the application of the continuation to the variable. Add the clause to `cps_exp` to implement the CPS-transformation of an expression that is a variable.

   ```
   # string_of_exp_cps (cps_exp (VarExp "x") (ContVarCPS Kvar));;
   - : string = "_k x"
   ```

2. ▼ Description of Constant Case
   The CPS transformation of a constant expression just applies the continuation to the constant, since constants are always already fully evaluated.

   $$[[c]]_\kappa = \kappa\, c$$

   The code for the function `cps_exp` should behave in a similar manner, creating the application of the continuation to the constant. Add code to `cps_exp` to implement the CPS-transformation of an expression that is a constant.

```
# string_of_exp_cps (cps_exp (ConstExp (BoolConst true)) (ContVarCPS
Kvar));;
- : string = "_k true"
```

3. ▼ Description of If-Then-Else Case
   Each CPS transformation should make explicit the order of evaluation of each subexpression. For if-then-else expressions, the first thing to be done is to evaluate the boolean guard. The resulting boolean value needs to be passed to an if-then-else that will choose a branch. When the boolean value is true, we need to evaluate the transformed then-branch, which will pass its value to the final continuation for the if-then-else expression. Similarly, when the boolean value is false we need to evaluate the transformed else-branch, which will pass its value to the final continuation for the if-then-else expression. To accomplish this, we recursively CPS-transform e_1 with the continuation with a formal parameter v that is fresh for e_2, e_3 and κ, where, based on the value of v, the continuation chooses either the CPS-transform of e_2 with the original continuation κ, or the CPS-transform of e_3, again with the original continuation κ.

$$[[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]]_\kappa = [[e_1]]_{\texttt{FN } v \texttt{ -> IF } v \texttt{ THEN } [[e_2]]_\kappa \texttt{ ELSE } [[e_3]]_\kappa}$$

   where v is fresh for e_2, e_3, and κ

   With $\texttt{FN } v \texttt{ -> IF } v \texttt{ THEN } [[e_2]]_\kappa \texttt{ ELSE } [[e_3]]_\kappa$ we are creating a new continutation from our old. This is not a function at the level of expressions, but rather at the level of continuations, and, as a result, should use the constructor `FnContCPS`.

   Add a clause to `cps_exp` for the case for if-then-else operators.

```
# string_of_exp_cps (cps_exp (IfExp (VarExp "b", ConstExp (IntConst 2),
                                     ConstExp (IntConst 5)))
                             (ContVarCPS Kvar));;
- : string = "(FN a -> IF a THEN _k 2 ELSE _k 5) b"
```

4. ▼ Description of Application Case
   The CPS transformation for application mirrors its evaluation order. In PicoML, we will uniformly use right-to-left evaluation. Therefore, to evaluate an application, first evaluate e_2 to a value, then evaluate the function, e_1, to a closure, and finally, evaluate the application of the closure to the value. To transform the application of e_1 to e_2, we create a new continuation that takes the result of e_2 and binds it to v_2, then evaluates e_1 and binds it to v_1, then finally, applies v_1 to v_2 and, since the CPS transformation makes all functions take a continuation, it is also applied to the current continuation κ. This is the continuation that is used in transforming e_2. Implement this rule:

$$[[e_1\ e_2]]_\kappa = [[e_2]]_{\texttt{FN } v_2 \texttt{ -> } [[e_1]]_{\texttt{FN } v_1 \texttt{ -> } v_1\ v_2\ \kappa}}$$
$$\text{where} \quad v_2 \text{ is fresh for } e_1 \text{ and } \kappa$$
$$\text{and} \quad v_1 \text{ is fresh for } v_2 \text{ and } \kappa$$

```
# string_of_exp_cps (cps_exp (AppExp (VarExp "f", VarExp "x"))
                         (ContVarCPS Kvar));;
- : string = "(FN a -> (FN b -> (b a _k)) f) x"
```

5. ▼ Description of Binary Operator Case
   The CPS transformation for a binary operator mirrors its evaluation order. It first evaluates its second argument, then its first before evaluating the binary operator applied to those two values. We create a new continuation that takes the result of the second argument,

e_2, binds it to v_2 then evaluates the first argument, e_1, and binds that result to v_1. As a last step it applies the current continuation to the result of $v_1 \oplus v_2$. Implement the following rule.

$$[[e_1 \oplus e_2]]\kappa = [[e_2]]_{\texttt{FN } v_2 \texttt{ -> } [[e_1]]_{\texttt{FN } v_1 \texttt{ -> } \kappa(v_1 \oplus v_2)}}$$
$$\textbf{where} \quad v_2 \textbf{ is fresh for } e_1 \textbf{ and } \kappa$$
$$\textbf{and} \quad v_1 \textbf{ is fresh for } \kappa, \textbf{ and } v_2$$

```
# string_of_exp_cps (cps_exp (BinOpAppExp
                              (IntPlusOp, ConstExp(IntConst 5),
                                          ConstExp(IntConst 1)))
                            (ContVarCPS Kvar));;
- : string = "(FN a -> (FN b -> _k(b + a)) 5) 1"
```

6. ▼ Description of Unary Operator (Monadic Operator) Case
   The CPS transformation for a unary operator mirrors its evaluation order. It first evaluates the argument of the operator and then applies the continuation to the result of applying that operator to the value. Thus we create a continuation that takes the result of evaluating the argument, e, and binds it to v then applies the continuation to the result of $\oplus v$. Implement the following rule.

$$[[\oplus e]]\kappa = [[e]]_{\texttt{FN } v \texttt{ -> } \kappa\ (\oplus v)} \qquad \textbf{Where } v \textbf{ is fresh for } \kappa$$

```
# string_of_exp_cps (cps_exp (MonOpAppExp (HdOp, ConstExp NilConst))
                            (ContVarCPS Kvar));;
- : string = "(FN a -> _k(hd a)) []"
```

7. ▼ Description of Function Expression Case
   A function expression by itself does not get evaluated (well, it gets turned into a closure), so it needs to be handed to the continuation directly, except that, when it eventually gets applied, it will need to additionally take a continuation as another argument, and its body will need to have been transformed with respect to this additional argument. Therefore, we need to use the continuation variable k (that is, `ContVarCPS Kvar`)to be the formal parameter for passing a continuation into the function. Then, we need to transform the body with k as its continuation, and put it inside a continuation function with the same original formal parameter together with k. The original continuation κ is then applied to the result.

$$[[\texttt{fun } x \texttt{ -> } e]]\kappa = \kappa\ (\texttt{FUN } x\ k \texttt{ -> } [[e]]k)$$

   The syntax for a function in PicoML, as in OCaml is (`fun _ -> _`). Write the clause for the case for functions.

```
# string_of_exp_cps (cps_exp (FunExp ("x", VarExp "x"))
                            (ContVarCPS Kvar));;
- : string = "_k (FUN x _k -> _k x)"
```

8. ▼ Description of Let-In Case
   A (`let x = e_1 in e_2`) expression first evaluates the expression e_1 generating a local binding, and then evaluates e_2 in the context of that new binding. Note that, in order to transform e_2 into CPS, we already have the necessary continuation κ because e_2 computes the value to be given as the final result. To transform the (`let x = e_1 in e_2`) expression, we create a continuation that takes the result of e_1, binds it to x, the locally bound variable, and calculates e_2 with the current continuation. This new continuation is then used to transform e_1. Implement the following rule.

$$[[\texttt{let } x = e_1 \texttt{ in } e_2]]\kappa = [[e_1]]_{\texttt{FN } x \texttt{ -> } [[e_2]]\kappa}$$

```
# string_of_exp_cps (cps_exp (LetInExp ("x", ConstExp(IntConst 2),
                                        VarExp "x"))
                            (ContVarCPS Kvar));;
- : string = "(FN x -> _k x) 2"
```

Extra Credit

9. ▼ Description of Let-Rec_in Case

   In PicoML, the only expressions that can be declared with `let rec` are functions. A (`let rec f x = e_1 in e_2`) expression creates a recursive function binding for f with formal parameter x and body e_1. The binding for f is then available for the evaluation of e_2. When e_1 is evaluated in the context of a function call in e_2, the environment for e_1 will need to be updated with this binding. Since we require `let rec` expressions to bind identifiers to functions, we do the CPS transform for this local declaration in a fairly similar way. We transform the body with respect to the continuation variable and parameterize by that continuation variable. We need to convert the CPS transformed expression waiting for the binding into a continuation taking a value for f. The main difference at the end is that we wrap it all up with a constructor representing a fixed-point operator. Implement the following rule.

$$[[\texttt{let rec } f\ x\ =\ e_1\ \texttt{in}\ e_2]]_\kappa = (\texttt{FN } f \texttt{ -> } [[e_2]]_\kappa)(\mu\, f.\ \texttt{FUN } x\ k \texttt{ -> } [[e_1]]_k)$$

```
# string_of_exp_cps (cps_exp (LetRecInExp ("f", "x",VarExp "x",
                                  ConstExp (IntConst 4)))
                            (ContVarCPS Kvar));;
- : string = "(FN f -> _k 4)(FIX f. FUN x _k -> _k x)"
```

Save & Grade *Unlimited attempts* | Save only

Correct answer