COMP 301 Project 4

Baran Berkay Hökelek, 0060673
Berkay Barlas 0054512
Utku Noyan 0060717

January 10, 2021

Workload Breakdown

- Baran Berkay Hökelek: Tasks 1, 2, 3
- Utku Noyan : Task 5
- Berkay Barlas: Task 4, 5

Task 1

- **Call-by-value vs. call-by-reference:** If we deal with applying a procedure to a variable that we eventually evaluate sometime later in our expression, applying these two styles of parameter passing would result in different values. This is due to the fact that in call-by-value, a copy of the argument is passed to the procedure; meanwhile in call-by-reference, the address of the argument is passed to it. In call-by-value, this results in the procedure modifying a copy of the argument; and in call-by-reference this results in the procdure directly modifying the only existing copy of the argument. There are advantages and disadvantages of using both styles. For example, if we're using an interactive shell (such as MATLAB or IPython) and want to "peek" at a possible modification for a data (say, sorting an array) without changing the actual data itself, the language being call-by-value would be massively advantageous, as is the case in both MATLAB and IPython. However, if we wish to write a function that changes a data value, apply that function on said data value, and then use that data value for further computation; we normally prefer that our original data be modified to avoid confusion. In that case, call-by-reference is more useful.
- **Call-by-need vs. call-by-name:** In a purely functional language like Haskell, where every call to a function with the same variable results in the same output, and there are no side effects, call-by-need and call-by-name would always yield the same answers. However, if the language contains

global variables and procedures which have side effects (like most currently used progeamming languages), using call-by-name as the parameter passing style might result in a different answer compared to using call-by-need. That's because in call-by-name, we keep evaluating the operand of the procedure whenever it is needed, and in between these evaluations the state (or memory) of the program might've changed in a way that affects the evaluation of the operand, causing the subsequent evaluations to give a different answer. Call-by-need, in contrast, only evaluates the operand once and avoids the possible effects of state changes. In a pure functional language, call-by-need is much more advantageous because it allows memoisation, which can drastically reduce computation time if the program involves repeated evaluations of a same operation. However, if our language contains a state and we want to take the effects of state change into consideration, then call-by-name would be more advantageous.

Task 2

These code pieces are needed to introduce lazy evaluation, the notion of thunks and the call-by-need style into the language. In the changes made to the evaluation of var-exp, the first 2 lines after the switch-case predicate line (var-exp (var)... finds whatever corresponds to the given variable in the memory. Then in the next 2 lines, if the thing is an expressed value, then it is returned. By that point, we know that if it's not an expressed value, it must be a thunk because these are the only 2 possible types of values we have in that language. So, the next 4 lines deal with how to handle a thunk. This is where the specifics of call-by-need is implemented. Respectively, the thunk is evaluated, the reference corresponding to the variable is changed to this new value, and then this value is returned.
In the addition of the procedure value-of-thunk, the code basically lays the groundwork in how to evaluate thunks. The first 4 lines deal with setting up a switch-case statement which only accepts the thunk format whose first attribute is an expression and second one is an environment. Then it declares that the value of a thunk is the value of the expression that it contains, evaluated with respect to the environment it contains.
Both of the code pieces should be implemented in **interp.scm**.

Task 3

1. **Call-by-reference vs. Call-by-need**
   In CBR, this expression gives 5 as an answer, however, in CBName, it gives 3.

```
(cbr-vs-cbneed
"let p = proc(x) 11
      in let f = proc(x) begin set x = 5; 12 end
          in let x = 3
          in begin (p (f x)); x end" 5 #| or 3 |#)
```

2. **Call-by-reference vs. Call-by-name**
   In CBR, this expression gives 0 as an answer, however, in CBName, it gives 1.

```
(cbr-vs-cbname
       "let or = proc(s1)
                 if zero?(-(s1, 1))
                  then proc(s2) 1
                  else proc(s2) if zero?(-(s2, 1)) then 1 else 0
            in let exp1 = 1
                   in let exp2 = proc(x) begin set x = 0; 0 end
                          in let x = 1 in
                              begin
                              ((or exp1) (exp2 x));
                              x
                              end" 0 #| or 1 |#)
```

3. **Call-by-value vs. Call-by-need**
   In CBV, this expression gives 2 as an answer, however, in CBNeed, it gives -4.

```
(cbv-vs-cbneed
       "let k = 0
            in let p1 = proc(x) -(-(1,-1), k)
                   in let p2 = proc(x)
                        if zero?(x) then begin set x = 27; x end else x
                          in begin set k = 6; (p1 (p2 k)) end" 2 #| or -4 |#)
```

4. **Call-by-value vs. Call-by-name**
   In CBV, this expression gives 3 as an answer, however, in CBName, it gives 1.

```
(cbv-vs-cbname
       "let k = 3
            in let p = proc(x) k
                   in let f = proc(x) begin set x = 5; 12 end
                          in begin set k = 1; (p (f k)) end" 3 #| or 1 |#)
```

Task 4

In this task we implemented a function fibonacci, that takes a parameter n and returns the nth Fibonacci number, with Continuation Passing Style.

If n <= 2, it sends 1 to the continuation. Otherwise, it works on n 1 in a continuation that calls the result x1 and then works on n 2 in a continuation that calls the result x2 and then sends (+ x1 x2) to the continuation. The code is provided below:

```
(define fibo-cont
  (lambda (n c)
    (if (<= n 2)
        (c 1)
        (fibo-cont (- n 1)
        (lambda (x1)
          (fibo-cont (- n 2) (lambda (x2) (c (+ x1 x2)))))))))


(define fibonacci
  (lambda (n)
    (fibo-cont n (lambda (x) x))))
```

Task 5

Given a LETREC implementation that has CPS with data-structural representations for continuations, we extended this language to include list and map. In our CPS implementation, value-of calls are tail calls. In list implementation we add two new values to the language:

- pair value: Implemented an pair value as expval in "data-structures.scm."
- emptylist value: Implemented an emptylist value as expval in "data-structures.scm."

The list is composed of pairs, as list expression looks like:

- list(exp1, exp2, ..., expN)

The list operations we implemented are:

- car(expression): The grammatical specification of method is implemented in "lang.scm".The continuation implemented in "interp.scm". It takes expression and returns the left part of the pair value.
- cdr(expression): The grammatical specification of method is implemented in "lang.scm".The continuation implemented in "interp.scm". It takes expression and returns the right part of the pair value.
- null?(expression): The grammatical specification of method is implemented in "lang.scm".The continuation implemented in "interp.scm". It takes expression and returns true if the expression is an emptylist value.

- emptylist(): The grammatical specification of method is implemented in "lang.scm". Implemented emptylist expression case in "interp.sm". Creates an empty list, with the value emptylist.
- expval->car(expval): Implemented in "data-structures.scm". Takes expval and extracts the car of the expressed pair value. This method is extractor.
- expval->cdr(expval): Implemented in "data-structures.scm". Takes expval and extracts the cdr of the expressed pair value.This method is extractor.
- expval-null?(expval): Implemented in "data-structures.scm". Takes expval and returns true if the expressed value is an emptylist value. This method is predicate.

In the map implementation the map expression looks like: map(expression, expression) Here, the first expression will be treated like a proc expression with one parameter, and the second expression will be treated like a list expression.

- We implemented the map expression case in "interps.scm", value-of/k.
- We implemented 3 map-exp continuations in "interps.scm", the apply-cont.