CS 425 - 2023 Week 6 Homework

Problem. Consider the following SML code:

```
val t = g h;
val u = let val x = 23 in g h end;
```

Are there definitions of g and h for which t and u have different values? If yes show these definitions, if not say why.

Answer. No. The scope of x is between the *in* and *end* delimiters. There are no occurrence of x in this scope, hence the value of x is irrelevant, never used.

Problem. Consider the following SML code:

```
val twice = fn f => fn x => f (f x);
val x = 1
val incr = fn y => y + x;
```

Since in a program *incr* appears several time as the first argument of *twice*, Joe Programmer attempts to optimize the code by specializing *twice* with $twice'$, so that, where appropriate, he can simply call $twice'$ instead of (*twice incr*):

```
val twice' = fn x => incr (incr x);
```

Does this alleged optimization preserve the meaning of *twice*?

Emboldened by the success of his first optimization, Joe attempts to further optimize the code by inlining the call to *incr* (replacing the call with the body):

```
val twice'' = fn x => (fn y => y + x) ((fn y => y + x) x);
```

Does this alleged optimization preserve the meaning of $twice'$?

Answer. Function $twice'$ preserves the meaning of *twice*. Function $twice''$ does not preserve the meaning of $twice'$. The problem is that the occurrence

of variable x in the body of *incr* is free, whereas the corresponding occurrences in the body of $twice''$ are bound. The fix is renaming (α-reduction) x in the body of $twice'$:

```
val twice' = fn z => incr (incr z);
```

before inlining.

Problem. Consider the following SML code:

```
fun f x
    = let fun g 0 = print("hi\n")
            | g 1 = f 2
            | g 2 = g 1
      in case x of
              3 => g 2
            | 2 => f 1
            | 1 => g 0
      end;
```

Draw the stack frames allocated during the evaluation of $f(3)$ at the time the program prints `"hi"`. Include control and access (static and dynamic) links.

Answer. The frames below are identified by function name and argument of a call. The stack grows from left to right, the top is $g0$:

$$f3 \;\; g2 \;\; g1 \;\; f2 \;\; f1 \;\; g0$$

The control link is from each frame to the previous frame (except for $f3$, which is undefined). The access links of f stack frames are undefined, from g frames are: $g0 \to f1$, $g1 \to f3$ and $g2 \to f3$.

Problem.
A stack frame of a function call typically allocates a memory location that stores the address of the returned value, not the returned value itself. Why this indirection? Would not be more efficient to store the returned value rather than its address?

Answer. If the returned value is stored in the stack frame of a call, when the call returns the stored value would be lost because the stack frame is deallocated. Instead, the returned value must be stored in a location available to the caller after the call returns. Through its address, it is stored there.

Problem.
Consider the following fragment of SML code:

```
fun f x = if ...  then f (g x)  else ...;
```

Where `...` is irrelevant code.

When the recursive call to f is executed, a stack frame, call it sf, of f is created. Say whether the following statements are true or false.

1. the stack frame below sf is a stack frame of g.
2. the stack frame below sf is a stack frame of f.
3. the stack frame below sf may be a stack frame of either f or g.
4. sf and the stack frame below it have the same control link
5. sf and the stack frame below it have the same access link

Answer. false, true, false, false, true

Problem.
Code in SML two versions of a function that takes a list of integers and returns the smallest element of the list. All your functions must be pure, no side effects, assignments, while loops and similar.

The first version should recur on the tail of the list and combine the result with the head. This is a classic pattern that you would use to compute the length of a list, add together all the elements of a list, and many other similar computations.

The second version may be familiar to a C program. This version uses an accumulator, a new variable that during a traversal of the list holds the minimum found so far. Upon looking at a new element of the list, if this element is not smaller than the accumulator, then it is thrown away, otherwise, its value becomes that new value of the accumulator.

Answer.

```
exception empty;

fun minlist [] = raise empty
  | minlist [x] = x
  | minlist (x::xs) = Int.min(x, minlist xs);
```

```
fun minacc' a [] = a
  | minacc' a (x::xs) = minacc' (Int.min (a,x)) xs;

fun minacc [] = raise empty
  | minacc (x::xs) = minacc' x xs;

val test1 = minlist [4,2,5,7,3]
val test2 = minacc  [4,2,5,7,3]
```

Problem.
Draw a few stack frames of a computation of the second version of the function of the previous problem when the list is being traversed. What can you say about the structure of these frames? What can you say about the content? What can you say about the use of the content?

Answer. These frame are all structurally equal, because each frame originate from a call to the same function. Some elements of a frame are equal in all the frames of the recursive call. While this depends on an implementation a candidate is the access link. Obviously, the local variables (arguments of a call) are different.