PLD Assignment 1

Matilde Broløs (jtw868)

February 26, 2020

A 1.1

a)

The contents of the file *assignment1.le* are shown below.
First the `reverse` function.

```
\ Assignment 1
(load listfunctions)

\ A function 'reverse' that reverses a list
(define reverse (lambda ((a . d)) (append (reverse d) (list a)) (a) a))
```

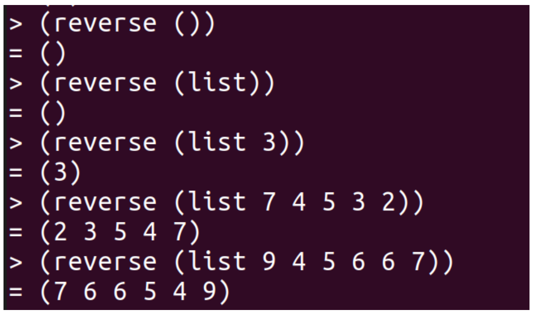
The function is recursively appending its first element to the back of the list that is generated by using the reverse function on the tail of the list.
To implement the second function, `sort`, I have implemented a number of helper functions, as seen below.

```
\ Helper function: append two lists (even if both lists are ())
(define append2
   (lambda (() bs) bs
           (bs ()) bs
           (() ()) ()
           ((a . as) bs) (cons a (append2 as bs))
           (bs (a . as)) (append (list bs a) as)
           (as bs) (list as bs)))

\ Helper function: Find all elements smaller than p
(define allSmallerThan (lambda ((a . d) p) (append2 (< a p) (allSmallerThan d p))
                               (() p) ()
                               (a p) (< a p)))

\ Helper function: Find all elements greater than p
(define allGreaterThan (lambda ((a . d) p) (append2 (> a p) (allGreaterThan d p))
                               (() p) ()
                               (a p) (> a p)))

\ A function 'sort' that sorts a list
(define sort (lambda ((a . d)) (append (append (sort (allSmallerThan d a))
                                    (list a)) (sort (allGreaterThan d a)))
                    (()) ()
                    a a))
```

The idea behind the code is to choose a pivot element, which is simply the first element of the given list, and then divide the rest of the list in to two different lists - one where all elements are smaller than the pivot element, and on where all elements are larger or equal. Now the sort function is used recursively on both lists. The function returns a list consisting of first the list with smaller elements (sorted), then the pivot element, and then the list with larger or equal elements (sorted).

b)

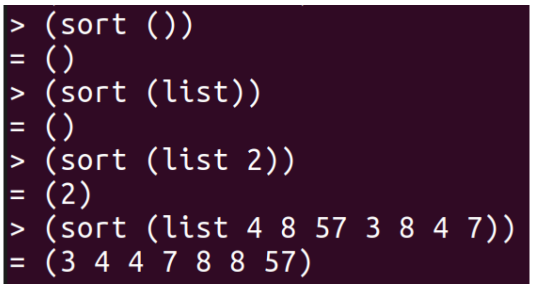
Here are some examples of running `reverse`.



```
> (reverse ())
= ()
> (reverse (list))
= ()
> (reverse (list 3))
= (3)
> (reverse (list 7 4 5 3 2))
= (2 3 5 4 7)
> (reverse (list 9 4 5 6 6 7))
= (7 6 6 5 4 9)
```

Figure 1: The reverse function

Here are some examples of running `sort`.



```
> (sort ())
= ()
> (sort (list))
= ()
> (sort (list 2))
= (2)
> (sort (list 4 8 57 3 8 4 7))
= (3 4 4 7 8 8 57)
```

Figure 2: The sort function

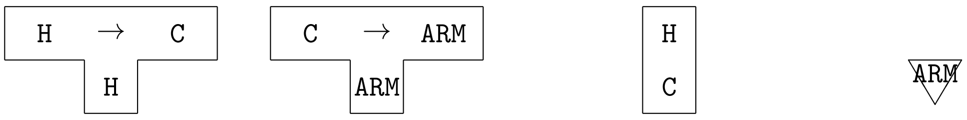
c)

I think the part of the syntax that was the most difficult for me to understand was the *cons* operater, and the way the pattern matching work. I think the idea is pretty straight forward, but i spent quite a lot of time just trying out all of the operators to figure out exactly how they worked and what they returned.

A 1.2

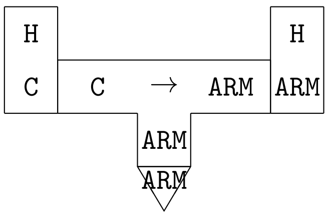
a)

Below the programs and the machine is represented as diagrams (H is short for Haskell).

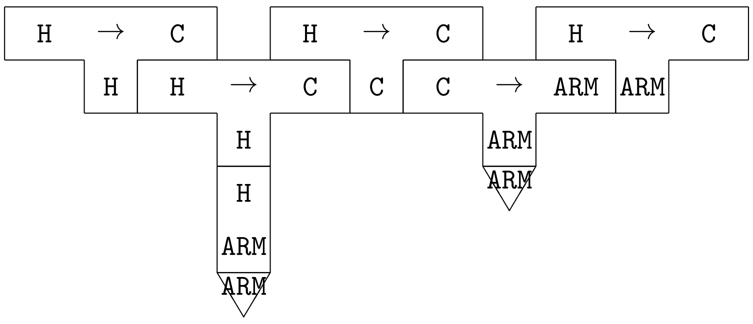


b)

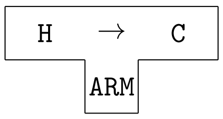
First we want to convert the interpreter of Haskell in C to an interpreter of Haskell in ARM:



So now we have the interpreter in ARM, and so we want to interpret the compiler from Haskell to C written in Haskell so we get a compiler from Haskell to C written in ARM.

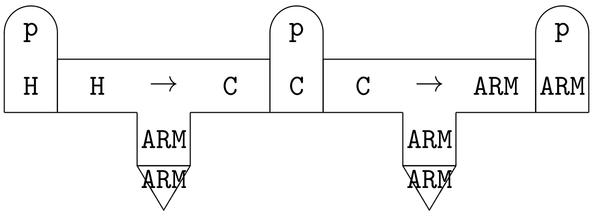


After running this, we have a compiler from Haskell to C written in ARM:



c)

Now we want to convert a program *p* written in Haskell to a program written in ARM, using our new compiler. This can be done on our ARM machine as shown below.



A 1.3

I assume that a bit representation has the least significant bit to the right, and the most significant bit to the left, and I therefore count from right to left when iterating through the bits.

a)

To make a function that can determine which type a bit representation *rep* has, we need to extract different bits to see their value. We know the following:

1. A *number* is the only representation that has 1 as the 2'nd bit.
2. A *NIL* value has only 0's.
3. A *symbol* has a 1 as the 34'th bit, and all bits after that bit is 0.

Now to make the function `kind` we need to check these 3 conditions. If neither of them are the case, the representation must be a *pair*. Below a suggestion for pseudocode is given. When using **AND** the operation is a *bitwise* and, and when /\ is used it is a logical and. When writing the bit string I indicate that it is a binary number by the suffix _2.

```
kind (rep):
  if (rep AND 3) == 2:
    return 1 ; A number
  else if rep == 0:
    return 0 ; NIL
  // Extract index i by bit shifting
  i = rep >> 33
  else if i == 1;
    return 2 ; A symbol
  else
    return 3 ; A pair
```

b)

To make a function that determines if two representations are equal, lets start by looking at the different types.

- If two representations are both NIL, we can determine if they are the same by simply checking if they have the same value.
- If two representations are both numbers, we can again determine if they are equal by checking if their values are the same.
- If two representations are both symbols, we know that for them to be equal both *j* indexes must be the same. The rest of the representation is predefined, so in this case we also simply have to check that the values of the bit representations are the same.
- Now, for a pair we know that two pairs can be equal, even if they dont point to the same indexes in the heap. Thus, we cannot just compare the values of the representations in this case. We need to extract the indexes from the representations, and then check whether the values at the given indexes in the heap are equal.

Thus, we have our cases for the function. Below a suggestion for the pseudocode is presented. When extracting the value at the x'th index from the heap is is represented as $heap(x)$.

```
equal (rep1, rep2):
  if kind(rep1) != kind(rep2):
    return FALSE
  else if kind(rep1) != 3:
    return (rep1 == rep2)
  else
    // Extract index i by bit shifting
    i1 = rep1 >> 33
    i2 = rep2 >> 33
    // Extract index j by AND'ing with bit value: 31x0 + 31x1 + 2x0 = 64 bits
    j1 = rep1 AND 0x1FFFFFFFC
    j2 = rep2 AND 0x1FFFFFFFC

    return (heap(i1) == heap(i2)) /\ (heap(j1) == heap(j2))
```

c)

What kind of fragmentation can mark-sweep cause? As I understand, the structure of the heap is that it is build up of a lot of chunks that are all 64 bits. The values stored in the heap cannot excede 64 bits, and at the same time we know that they all use exactly 64 - thus, as I see it, there can be no kind of fragmentation, if we assume that we can keep track of what indexes in the heap are unused, as we will always need exactly 64 bits for storing a value. Thus, the mark-sweep will not cause fragmentation, if we assume that the free indexes are kept track of.

d)

The following is a suggestion as to how the *mark-sweep-collection* could be modified to handle garbage collection in the symbol array. This assumes that all symbol values holds an index j to the symbol array, and not the heap.

```
mark-and-sweep()
  set b to start of array
  WHILE b =< end-of-array DO
    IF b == NULL DO
      go to next character in the array
    ELSE
      loop through all symbol pointers p
        IF p points to b DO
          go to next character in the array that is NULL
      set b = NULL
```

This code should loop through all indices in the symbol array: if the character is NULL, it goes to next character. If not, it loops through all symbols to check whether any of them points to this index. If the index is pointed to, we go to the next index that holds a NULL value. If no symbol points to the index, we set the value to NULL. After the mark-sweep all unused indices should have been set to NULL. When adding n new characters, one must find $n + 2$ indices in a row, all holding the value NULL.