Supervision work

Tudor Avram, Homerton College, tma33@cam.ac.uk

10.11.2015

1 ML

1.1 Excercise 12.1

The $\boxed{\textit{int ref list}}$ declaration represents a list of references to integers. On the other hand, the $\boxed{\textit{int list ref}}$ declaration represents a single reference, to a list of integers. For the

1.2 Excercise 12.2

Code :

```
fun power_it (x, n) =
    let
        val Sol = ref 1
        val b = ref n
        val a = ref x
    in
        while (!b <> 0) do
        (
            if (!b mod 2 = 1) then
                ( Sol  := !Sol  * !a; b  :=  !b - 1 )
            else (a  :=  !a  *  !a;  b := !b  div 2 )
        );
        !Sol
    end;
```

1.3 Excercise 12.3

The expresion evaluates C1 before B. Note that C1 does not have to be a bool expresion. It can be any kind of expresion, and will always be evaluated more times than C2

1.4 Excercise 12.4

```
fun change (x, y) =
    let
        val n = ref (!x)
        val m = ref (!y)
    in
        (m,n)
    end;
```

1.5 Excercise 12.5

Create an *n*n matrix*

```
fun create n =
    Array.tabulate (n, fn x => Array.tabulate(n, fn y = > (x*n) + (y+1))) ;
```

Transpose an *m*n matrix*

```
fun transpose A n m =
    Array.tabulate (n, fn x => Array.tabulate(m, fn y =>
            Array.sub(Array.sub(A,y),x)));
```

2 Java

2.1 Excercise 1

a) In a functional language, the programmer just tells the compiler what the result of a function should be and the compiler can choose the method by which it gets to that answer. With an imperative programming language, the programmer can tell the compiler the exact steps it needs to take to get to the wanted answer.
b) While a functional language is based only on functions as the main control flow, an iterative one also uses loops and conditionals.
c) In an imperative language, the programmer can easily change the state of a variable, while in a functional language this is not easy to achieve.

2.2 Excercise 2

int sum(int[] a) :

```
public static int sum (int[] a)
{
    int n = a.length;
    int s = 0;
    for (int i=0;i <n;++i)
        s+=a[i];
    return s;
}
```

int[] cumsum(int[] a) :

```
public static int[] cumsum(int[] a)
{
    int n = a.length;
    int[] S = new int[n];
    S[0] = a[0]; // I assume it is not empty
    for (int i=1;i<n;++i)
        S[i] = S[i-1] + a[i];
    return S;
}
```

int[] positives(int[] a) :

```
public static int[] positives(int[] a)
     {
    int n = a.length;
    int[] Sol = new int[n];
    int k=0;
    for (int i=0;i<n;++i)
        if (a[i] >0) Sol[k++] = a[i];
    return Sol;
}
```

2.3 Excercise 3

Primitives : d and f
References : i, l, k and t
Classes : Computer, Tree and LinkedList

2.4 Excercise 4

a) The unit-matrix creator :

```
public static float[][] create (int n)
{
    float[][] Sol = new float[n][n];
    for (int i=0;i<n;++i)
        for (int j=0;j<n;++j)
            if (i==j) Sol[i][j]=1;
            else Sol[i][j]=0;
    return Sol;
}
```

*b) Transposing an n*m matrix :*

```
public static void transpose(int[][] A)
{
    int n = A.length;
    int m = A[0].length;
    for (int i = 0; i <n; ++i)
        for (int j=i; j <m; ++j)
        {
            int aux = A[i][j];
            A[i][j] = A[n-i-1][m-j-1];
            A[n-i-1][m-j-1] = aux;
        }
}
```

2.5 Excercise 5

In order to check if the Java environment performs tail-recursion optimisations, I used the following two functions :

```
public static int pow (int x,int exp)
{     if (exp == 0) return 1;
        else return x * pow(x,exp-1);
}

public static int tail_pow(int x, int exp, int sol)
{     if (exp == 0) return sol;
        else return tail_pow(x,exp-1,sol*x);
}
```

Even though both functions return x^{exp}, they are sightly different. While the first function performs normal recursion, the second one uses the concept of tail recursion, by keeping the partial result from every step in the parameter *sol*. In

ML this would have been an optimisation, but in Java both functions clash, due to stack overflow, for a big enough *exp* (I tested for $exp = 1000000$). Therefore, we can deduce that Java does not perform tail-recursion optimisation.

2.6 Excercise 6

Pointers allow the programmer to access and even modify almost any adress in the computer. Even though for a skilled and experienced programmer this can be a big advantage, things can go wrong at any time. References, on the other hand, are safer because, when they are used, the compiler restricts the operations that can be done on them, so less damage can be done. Moreover, references can be checked for viability, while pointers can't.

2.7 Excercise 7

Person p = null ======== >p exists, but doesn't refer to any value
Person p2 = new Person() == >p2 points to a *Person* value stored in memory
p = p2 ============ >p and p2 point to the same memory adress
p2 = new Person() ===== >p2 now points to a new adress in memory, where a *Person* variable is stored
p = null ============ >At this point, p2 points to a memory adress, while p does not point to anything.

2.8 Excercise 8

The given code fails because, even though function *vectorAdd* increments the 2D vector, it does not pass the results to the *main* function, so the vector (a, b) is not actually changed.

My aproach :

```
public static int vectorAdd(int x, int dx)
{
    return x + dx;
}

public static void main(String[] args)
{
    int a = 0;
    int b = 2;
    a = vectorAdd(a,1);
    b = vectorAdd(b,1);
    System.out.println(a+" "+b);
    //(a,b) becomes (1,3)

}
```

2.9 Excercise 9

To show the fact that the arrays are passed to functions as arguments, I used the following code :

```
public static void ChangeArray(int[] A)
{
    int n = A.length;
    for (int i = 0;i <n; ++i)
        A[i] = A[i] + 1;
}

public static void main(String[] args)
{
    int[] a = new int[3];
    a[0] = 1; a[1] = 2; a[2] = 3;
    ChangeArray(a);
    System.out.println(a[0]+" "+a[1]+" "+a[2]);
}
```

In this case, the programme prints *"2 3 4"*, which means that our array has been modified, even though at first sight we did not apply any change directly on it. Therefore, we can deduce that the array has been passed as a reference to function *ChangeArray*.

2.10 Excercise 10

An advantage of such an imperative language would be the fact that the variables can be passed as references to functions, in stead of making copies of them. This would lead to a small (but nevertheless existent) improvement in memory. On the other hand, if we pass all the variables as references, their value will be changed, no matter if we want this or not. If we need to pass the same variable to multiple functions, we would need to keep copies of the variable and we would end up in the same place where we started from. Our memory improvement would technically be non-existent.