Q1 Language Concepts (20 marks)

(i) If a function body has an if statement with a missing else case, then an exception is raised if its condition is false. Explain why this behavior is correct. However, this situation does not occur for precedures. Explain why not. (8 marks)

Ans : As expression/function need to return a result, each if statement must always return an answer. Since exception can be viewed as an error outcome, a missing else clause can be viewed as returning this an error result. Hence:

```
    if v then e1 end ==> if v then e1 else raise exception end
```

Another route to take is to return some default value in case the missing else was taken. However, since Oz is untyped, it is difficult to determine a suitable default, other than exception itself.

For procedures, we are often computing it for its effects. Hence, a missing else clause is perfectly legitimate as it denotes the Skip instruction without any effect. In case of if with statement, we can perform the following translation.

```
    if v then s1 end ==> if v then s1 else skip end
```

(ii) One can claim that both the if and the case statements are of equal expressive power. Elaborate on the truth or falsity of this claim. (7 marks)

Ans : we can translate any "if" to a "case" as follows:

```
   if V then E1 else E2 end
       ==> case V of true then El else E2 end
```

Similarly, it is possible to translate every case construct to an if

```
     case V of c(V1,..,Vn) then E1 else E2 end
       ==> if {Label V}=c andThen {Width V=n} then
                 V1=V.1; .. ; Vn=v.n ; E1
           else E2 end
```

Thus, strictly speaking they are of equal expressive power. However, case construct are more concise. In this sense, you could say that case construct is a more general and powerful mechanism since it can do more things easily.

(iii) Given the following procedure: (5 marks)

```
proc {Test X}
  case X of
   f(a Y c) then {Browse 'case 1'}
  else {Browse 'case 2'}
  end
end
```

Predict what would happen when you execute the following codes:

(a) declare X Y {Test f(X b Y)}
  => suspended since X,Y are undefined

(b) declare X Y {Test f(a Y d)}
  => 'case 2' displayed since else clause is taken

(c) declare X Y {Test f(a Y c)}

```
          => 'case 1' displayed


      (d) declare X Y {Test f(X Y d)}
           => suspended since X is undefined as mathing is strict.
              if lazy matching had been used, we would know that
              the 3rd argument d will fail to match c, regardless of
              what X is defined to be.

      (e) declare X Y {Test f(X Y c)}
            => suspended since X is undefined

Q2 Lambda Calculus (25 marks)

 (i) Consider the following lambda expressions. Mark the free
     variables in these expressions.  (7 marks)

  free vars are marked with #

    (a)  (\x . y#)

    (b)  (\x . x)

    (c)  (\x. (\y. y)) x#

    (d)  (\x. (\y. x)) x#

    (e)  (\x. (\y. x)) y#

    (f)  \z. ((\x. z) (\x.z))

    (g)  (\z. (\x. z)) (\x.z#)

(ii) Consider the following lambda expressions. Count the
     number of redexes (reducible subexpressions) in each of these
     lambda terms.  (5 marks)

  Ans : Redexes are shown underlined. These are expressions that
        can undergo beta-reduction.

    (a)  (\x. x) (\x. x)
         ---------------           ==> 1

    (b)  (\x. (\x.x) x) (\x. x)
              --------
         ----------------------      ==> 2

    (c)  (\x. x x) (\x. x x)
         -------------------      ==> 1

    (d)  (\x. y) ((\x. x x)(\x. x x))
                   -----------------
         ----------------------------  ==> 2

    (e)  (\x. x (\x. x))
                                ==> 0
```

(iii) Perform beta reductions using call-by-value (leftmost innermost) strategy for the following lambda expressions. If the reduction is non-terminating, suggest if there is an alternative reduction that terminates for the given code. (7 marks)

Let us assume leftmost-innermost but no evaluation inside a lambda term.

```
(a) (\x. x) (\x. x)
    ==> \x. x

(b) (\x. (\x.x) x) (\x. x)
    ==>  (\x.x) (\x. x)
    ==>  (\x. x)


(c) (\x. x x) (\x. x x)
    ==> (\x. x x) (\x. x x)
    ==> (\x. x x) (\x. x x)
    ==>  ....
        goes into a loop


(d) (\x. y) ((\x. x x)(\x. x x))
    ==> (\x. y) ((\x. x x)(\x. x x))
    ==> (\x. y) ((\x. x x)(\x. x x))
    ==>  ...
         goes into a loop since (\x. x x)(\x. x x) is chosen
         by leftmost-innermost
    If we had used leftmost-outermost, our reduction will
    terminate and give:
    ==> y
    which avoids the loop from innermost redex.



(e) (\x. x (\x. x))

    cannot reduce as no redex!
```

(iv) Given a lambda term T. How would you show that this term is a fix-point operator? Comment briefly on the significance of fix-point operators. (6 marks)

To show that T is a fix-point operator, we must prove for any F:

```
T F = F (T F)
```

Such an operator will return a fixpoint for any F, since we now have:

```
X = F X
```

where X is the fixpoint of F.

Fixpoint operators are important since they are the foundations for recursive functions. With it, we can implement recursion without any extra machinery.

Q3 Stack ADT (20 marks)

Consider a stack ADT that is non-declarative whose operations may have side-effects. An example operation is given below :

```
Push :: Stack<X>, X --> ()
// takes a stack and an element which is pushed
// to the top of the stack
```

which when executed will modify its stack by adding a new element to the top of the stack.

(a) Provide more stack ADT operations that would allow you to construct, modify and query the stack ADT. Give only the polymorphic type interface without implementation details. (8 marks)

```
construct:
  NewStack :: () -> Stack<X>
modify:
  Pop :: Stack<X> -> X
query:
  Top :: Stack<X> -> X
  IsEmpty :: Stack<X> -> Bool
```

(b) Show how you would implement this non-declarative stack ADT by showing how each of its operations may be implemented in Oz. (Hint : You may need to use mutable structure, such as Cell, Array or Dictionary.) (12 marks)

You just need to use Cell<List<X>> as its implementation

```
construct:
  % NewStack :: () -> Stack<X>
  fun {NewStack} {NewCell nil} end

modify:
  % Pop :: Stack<X> -> X
  fun {Pop S}
            case @S of
              H|T then S:=T
                       H
            end  % fails for empty stack
   end
  % Push :: {Stack<X>, X} -> ()
  fun {Push S X} S:= X|@S end
query:
  % Top :: Stack<X> -> X
  fun {Top S}
            case @S of
              H|T then H
              else raise exception?
            end  % fails for empty stack
   end
  % IsEmpty :: Stack<X> -> Bool
```

```
fun {IsEmpty S}
            case @S of
              H|T then false
              else true
            end
 end
```

Q4 Concurrency (15 marks)

The following is a naive attempt to increase the concurrency of the Filter function:

```
fun {Filter L F}
 case L of
   X|Xs then if thread {F X} end
             then X|{Filter Xs F}
             else {Filter Xs F} end
   else nil
 end
end
```

(i) Comment on the effectiveness of this attempt. (5-marks)

Ans : As the concurrent thread is in the conditional's test, the statement has to wait for the thread to complete before continuing. Due to this dependency, the concurrency here is useless.

(ii) Suggest how you may provide an alternative Filter operation with better concurrency. Outline the key steps that you need to make. Please provide a narrative of your solution, but do not provide any program code at all. (Hint : You may make make use of message-passing concurrency.) (10-marks)

Ans : To get effective parallelism, we will have to compute all {F X} in parallel, and collect successful X in a non-deterministic stream. To recover the order of the elements, we may have to attach a position to each X, and use this to sort the output to its original order.