Solution notes

Introduction to Functional Programming 2005 – Paper 13 Question 10 (RGR)

(*a*) `subsets`. 1 mark for a correct base case, 2 for a correct `map` (an explicitly recursive function that doesn't use map only gets 1 mark), 1 for the `let val` binding of the recursive call (take off this point if they call `subsets` twice in the recursive case), and 1 for correctly appending the two lists together to form the result. [5 marks]

```
fun subsets [] = [[]]
  | subsets (x::xs) =
    let val p = subsets xs
    in
      (map (fn elt => x :: elt) p) @ p
    end
```

(*b*) exceptions

(*i*) `exception NoFit` [1 mark]

(*ii*) `exception Success of int list` [1 mark]

(*c*) `knapsack`. 1 mark for raising the exception in the failure case, 2 for using a functional (it doesn't have to be `foldl` as used here–they could `map` each list to a `(sum, [list])` pair prior to iterating over the list or use `foldl` or `foldr` to replace the outer recursive loop). 1 for calling `subsets` correctly, and 1 for a correct recursion (full marks if they replace it with a functional that works, even, if it doesn't terminate immediately upon finding a solution). [5 marks]

```
fun knapsack target lst =
  let fun f [] = raise NoFit
        | f (x::xs) = if target = (foldl op+ 0 x) then x
                      else f xs
  in
    f (subsets lst)
  end
```

(*d*) `knapsack2`. 3 marks for correctly making both recursive calls (the order is not important), 2 for correctly maintaining the result list (−1 if they reverse the order and do not correct it), 1 for correctly tracking the sum found so far, 1 for correctly raising `Success` with the result, and 1 for terminating and returning `unit` on failure. [8 marks]

```
fun knapsack2 target lst =
let fun f _ _ [] = ()
      | f res sum (x::xs) =
        let val res' = res @ [x]
            val sum' = sum + x
        in
          if sum' = target then raise (Success res')
          else (f res sum xs; f res' sum' xs)
        end
in
  f [] 0 lst
end
```