1. (9 p)

Lösning:

(a)
```
count::(Eq a)=>a->[a]->Int
count target list =
    countHelper target list 0
```

```
countHelper::(Eq a)=>a->[a]->Int->Int
countHelper target [] n = n
countHelper target (x:xs) n
    |x==target = countHelper target xs n+1
    |otherwise = countHelper target xs n
```

```
count _ [] = 0 -- Variant utan svansrekursion
count target (x:xs)
    |x==target = 1 + count target xs
    |otherwise = count target xs
```

```
count _ [] = 0 -- tredje variant med if then else
count target (x:xs) =
    if x==target then 1 + count target xs
    else count target xs
```

(b) `Bool, Char, Double, Integer`

(c) `countSpace = count ' '`
Egenskapen heter partiell funktionsapplikation men vi godkänner också currying.

Rättning:

På varje uppgift:
0p är minsta poäng som kan ges på varje deluppgift.
-2p logiska fel
-1p för varje slarvfel (men högst 1p för samma typ av slarvfel)
Inga avdrag för slarvfel där innebörden tydligt framgår t.ex. Intger istället Integer
Uppgiftsspecifik bedömning:

(a)
i. Ett funktionsanrop har rätt värde men fel typ ⇒ -1p
ii. Prologsyntaxen `[H|T]` istället för `(x:xs)` ⇒ -1p
iii. Det slutgiltiga returvärdet från `count` har fel returtyp ⇒ -2p
iv. Inget rekursivt anrop görs ⇒ -2p
v. Rekursivt anrop finns, men basfallet saknas ⇒ -2p
vi. Saknar en parameter vid anrop av hjälpfunktion ⇒ -2p
vii. Pilar av typen `->` mellan parametrarnas namn under signaturen ⇒ -1p
viii. Bryter mot typsignaturen för `count` med en extra parameter ⇒ -2p
ix. Parenteser runt en funktions parameterlista ⇒ -1p
x. Felaktig uppdatering av parametrar vid funktionsanrop ⇒ -2p

Specialfall på ovanligt(?) missförstånd:

```
{- Counts the occurences of target in (x:xs). This implementation uses pattern
matching. -}
count::(Eq a)=>a->[a]->Int
count target (target:xs) = 1 + count target xs -- Kompileringsfel
count target (_:xs) = count target xs
```

ghc säger: "Conflicting definitions for ‘x’"eftersom man inte kan använda target både som parameter och som bindning till huvudet av listan inne i funktionen. ⇒ -1p

(b) 4 av dessa typer med 0 eller flera par matchande `[]` ger 2p. Andra typer som implementerar Eq är också OK. Exempel: `Bool, Char, Double, Int, Integer, Float, String, Account, [Bool], [Char], [Double], [Int], [Integer], [Float], [[Char]]`

(c) För att få 2p krävs partiell funktionsapplikation kod och att något av följande står i texten:

- Partiell funktionsapplikation
- Partial function application
- Currying

För att bedöma koden, använd följande regler. Tyvärr hade vi tillåtit för många tecken då KS:en var live, vilket gav några lösningar utan partiell funktionsapplikation.

i. Inget avdrag för `->` istället för `=` mellan parameterlistan och funktionskroppen.
ii. Inget avdrag för newline istället för `=` mellan parameterlistan och funktionskroppen.
iii. Lösningen använder mer än 22 tecken ⇒ -1p
iv. I lösningen står det ‘‘ ’’ istället för ’ ’ ⇒ Inget avdrag.

2. (5 p)

Lösning:

(a) Algebraisk datatyp.

(b) Den första skriver ut en störtflod av de udda talen tills någon trycker ctrl-C, stänger av programmet, datorn kraschar eller strömavbrott inträffar.
Den andra raden skriver inte ut någonting.
Egenskapen kallas lat evaluering.

Rättning:

(a)
- Algebraisk datatyp ⇒ 2p.
- Record eller record syntax ⇒ 2p.
- Parametriserad datatyp ⇒ 0p.
- Typparametriserad datatyp ⇒ 0p.
- Godtycklig slumpad term från boken som: Statisk typning, dynamisk typning, stark typning, svag typning, explicit typning, implicit typning, typinferens, typsystem ⇒ 0p.

(b) Första raden skriver ut (oändligt) många (udda) tal (Skriver ut många tal ger godkänt) ⇒ +1p.
Andra raden skriver ingenting ⇒ +1p.
Egenskapen kallas lat evaluering (vi godkänner också sen bindning, late binding, lazy evaluation). "Den är lat" räcker för poäng. ⇒ +1p.

3. (6 p)

Lösning:

(a)
```
sumOfSquaredDifference::[Double]->[Double]->Double
sumOfSquaredDifference uppmätt extrapolerat =
    sum (map (\(a,b)->(a-b)^2) (zip uppmätt extrapolerat))
```

```
sumOfSquaredDifference uppmätt extrapolerat = -- alternativ lösning
    foldl (\a (b,c)->a+(b-c)^2) 0 (zip uppmätt extrapolerat)
```

(b)
```
map :: (a -> b) -> [a] -> [b]
sum :: (Num a, Foldable t) => t a -> a
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Rättning:

Man kan aldrig få mindre än 0p på en uppgift. Slarvfel som ger -1p ger inte ytterligare minus om de upprepas.

(a)
i. Lambdafunktionerna använder λ-symbol istället för \ ⇒ Inget avdrag.
ii. Det finns en hjälpfunktion i svaret ⇒ -2p
iii. Det finns ett rekursivt anrop i svaret ⇒ -2p
iv. Prologsyntaxen `[H|T]` förekommer någonstans ⇒ -1p
v. Pilar av typen `->` mellan parametrarnas namn under signaturen ⇒ -1p
vi. Typsignatur saknas ⇒ -1p
vii. Parenteser runt funktionens parameterlista ⇒ -1p
viii. Lambdanotationen saknar parenteser för tupeln `(\a b->a*b)` ⇒ -1p
ix. Lambdanotationen använder ordet lambda, som Python. ⇒ -1p
x. Har kombinerat listorna med en listomfattning som genererar alla kombinationer av tal: `[(a,b)|a<-uppmätt, b<-extrapolerat]` ⇒ -2p
xi. Anropen till foldfunktion saknar startvärde ⇒ -1p

(b)
i. Om alla anropade funktioner (utom `zip`, `+`, `-`, `*` och `/`) har sina typsignaturer här och de är nästan rätt ges full poäng. Det gäller även om den enda funktionen som blivit kvar efter uteslutningen av de andra är foldl eller foldr.
ii. Inget avdrag för att skriva List, Iterable eller annat kreativt namn istället för Foldable. (Trots att det är fel.)
iii. Inget avdrag för att ersätta Num a med ett kreativt namn som Numeric. (Trots att det också är fel)
iv. Inget avdrag för att sätta parenteserna fel i typsignaturen till `foldl` eller `foldr`
v. Inget avdrag för att blanda ihop typsignaturerna för foldl och foldr.
vi. En funktion har använts i (a) men saknar typsignatur här ⇒ -1p.
vii. Inget funktionsnamn i början av typsignaturen ⇒ -1p
viii. Bara ett eller inga kolon `::` mellan namnet och parametertyperna ⇒ -1p