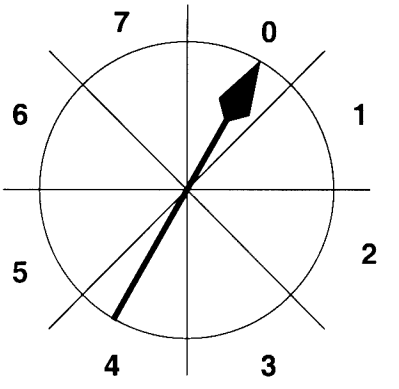
ORIENTATIE INFORMATICA
Toets A
14 oktober 2002; 09.00 – 12.00 uur

- Geef bij elke Haskell-functie ook de typering.
- In elk onderdeel mag je gebruik maken van functies uit vorige onderdelen, ook als je die niet hebt kunnen maken.
- Beargumenteer duidelijk je antwoorden.

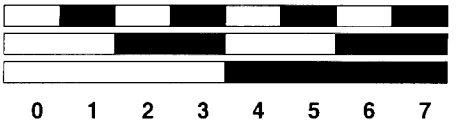
Opgave 1 (Gray-code)

Bekijk de tekening hiernaast. Een cirkel is verdeeld in acht sectoren, genummerd van 0 tot en met 7. De pijl is draaibaar opgesteld. Je zou kunnen denken aan een apparaatje om de windrichting te bepalen of aan een spelletje. In het eerste geval geeft het nummer van de sector een aanduiding van de windrichting; in het tweede geval wordt het winnende nummer aangewezen.

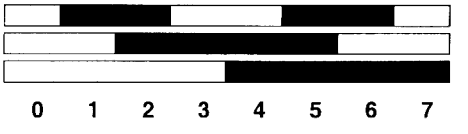


Het doel is nu om de aangewezen sector electronisch te bepalen. De cirkel is daarvoor uitgevoerd als een opstaande rand bestaande uit drie stroken en onder de pijlpunt zitten drie contacten. De stroken zijn verdeeld in isolerende en geleidende gebieden. Komt een contact in aanraking met een geleidend gebeid, dan wordt een spanning doorgegeven (`1`), anders niet (`0`).

Als we de cirkelrand uitvouwen krijgen we een patroon van geleidende en isolerende gebieden. Als we dit patroon kiezen overeenkomstig de binaire codering van de getallen 0 tot en met 7 (zie figuur 1-(a)), dan is uit de doorgegeven spanningen eenvoudig de aangewezen sector te bepalen.



(a)-Binaire codering



(b)-Gray codering

Figuur 1: Patronen op de cirkelrand

Als de pijl blijft staan op de grens tussen twee sectoren, dan kan dit problemen opleveren. Bijvoorbeeld als de pijl blijft staan op de grens tussen de sectoren 3 (`011`) en 4 (`100`) dan kan dat leiden tot het doorgeven van de code `111`, behorend bij sector 7. En dat ligt een heel eind uit de buurt.

lees verder

Een bit-codering die dit probleem enigszins ondervangt is een Gray-code (zie figuur 1-(b)). Een Gray-code heeft de eigenschap dat de coderingen van twee naast elkaar liggende sectoren in precies één positie verschillen. De volgende tabel geeft een overzicht van een aantal getallen in decimale, binaire en Gray-codering:

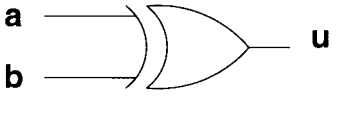
		gray-code	
decimaal	binair	2 bits	3 bits
0	0	00	000
1	1	01	001
2	10	11	011
3	11	10	010
4	100		110
5	101		111
6	110		101
7	111		100

De hier gepresenteerde Gray-codering wordt wel *Gespiegelde Gray-code* genoemd. Bekijk de lijst van 3 bits. In de eerste vier codes herkennen we rechtstreeks de lijst van 2 bits: ze zijn ontstaan door de lijst van 2 bits te nemen en voor elke bitstring een 0 te plaatsen.
In de tweede groep van vier herkennen we ook de lijst van 2 bits, maar nu van *beneden naar boven* en voorafgegaan door een 1.

[7 pt] □ 1. Geef in een tabel van de getallen 8 tot en met 15 de binaire- en de Gray-code.

[6 pt] □ 2. Geef de Gray-code van het getal 140. Laat zien hoe je aan je antwoord komt.

Een schakeling die bij een Gray-code de bijbehorende binaire code geeft, maakt gebruik van de XOF-poort:

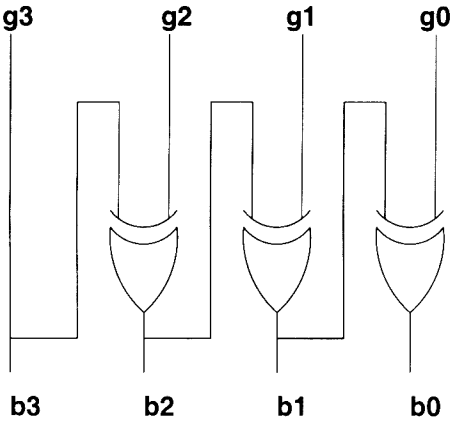


a	b	u
0	0	0
0	1	1
1	0	1
1	1	0

Figuur 2: De XOF*-poort: symbool en functietabel*

[6 pt] □ 3. Ontwerp een schakeling bestaande uit de standaard poorten (EN, OF, NIET) die de XOF-poort implementeert.

De tekening hiernaast geeft een conversie-schakeling die bij de vier-bits Gray-code $g_3g_2g_1g_0$ de bijbehorende binaire code $b_3b_2b_1b_0$ genereert.



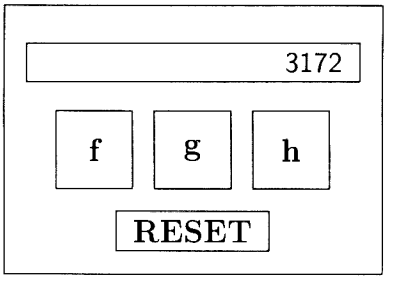
[8 pt] □ 4. Bereken met een vergelijkbare schakeling welk geheel getal de bitstring `11010101` als Gray-code heeft. Laat duidelijk zien hoe je aan je antwoord komt.

Het converteren kunnen we natuurlijk ook door een Haskell-functie laten uitvoeren. Dat gaat echter het eenvoudigst als het meest significante bit achteraan staat. Dat wil zeggen dat we de bitstring $a_n \ldots a_2a_1a_0$ representeren door de lijst `[a0, a1, a2, ... , an]`. We kiezen er voor om de bits te representeren met de karakters `0` en `1`.

[4 pt] □ 5. Geef een Haskell-functie `xof :: Char -> Char -> Char` die de XOF-schakeling implementeert.

[7 pt] □ 6. Geef bij de beschreven representatie van de bitstrings een Haskell-functie `gray2bin` die bij een gray-code de bijbehorende binaire code oplevert.

Opgave 2 (Stapje voor stapje)



Bekijk het machientje hiernaast. Het heeft een venster waarin (de staart van) een getal wordt getoond en verder vier knopjes. Het indrukken van de RESET-knop maakt het getal in het venster 1. Het indrukken van één van de andere knoppen voert een rekenkundige bewerking uit op het getal in het venster:

- **f**: het getal wordt met 2 vermenigvuldigd
- **g**: van het getal wordt 5 afgetrokken, mits het resultaat positief blijft. Anders gebeurt er niets.
- **h**: van het getal wordt 13 afgetrokken, mits het resultaat positief blijft. Anders gebeurt er niets.

We vragen ons af, hoe vaak bij een gegeven getal n er ná het indrukken van de RESET-knop een knopje moet worden ingedrukt om het getal n in het venster te krijgen. De volgende voorbeelden laten zien hoe je het getal 6 in vijf stappen kunt bereiken en het getal 33 in acht stappen:

$$1 \xrightarrow{f} 2 \xrightarrow{f} 4 \xrightarrow{f} 8 \xrightarrow{g} 3 \xrightarrow{f} 6$$

$$1 \xrightarrow{f} 2 \xrightarrow{f} 4 \xrightarrow{f} 8 \xrightarrow{f} 16 \xrightarrow{f} 32 \xrightarrow{h} 19 \xrightarrow{f} 38 \xrightarrow{g} 33$$

[5 pt] ☐ 7. Laat zien hoe je het getal 14 kunt bereiken.

We ontwikkelen nu een algoritme dat bij een positief geheel getal n het kleinste aantal stappen berekent waarin het getal n bereikt kan worden. We noemen dit de *afstand* van het getal n. We zullen daarbij een wachtrij (queue) gebruiken.

De wachtrij implementeren we als een lijst van paren gehele getallen. Het eerste element in zo'n paar is een (bereikbaar) positief geheel getal `k`; het tweede element is het aantal stappen waarin `k` bereikt kan worden.

Het idee is nu om steeds te kijken of het eerste element in de wachtrij correspondeert met het gezochte getal. Zo ja, dan is de afstand bekend. Zo nee, dan voegen we de 'opvolgers' van het kopelement toe aan de wachtrij.

[6 pt] ☐ 8. Leg uit waarom in dit geval voor een wachtrij (queue) wordt gekozen en niet voor een stapel (stack).

[5 pt] ☐ 9. Geef een Haskell-functie `enQueue` die een element (dus een paar) toevoegt aan een wachtrij.

[8 pt] ☐ 10. Geef een Haskell-functie `enQueueKids` die de opvolgers van het paar `(x, k)` toevoegt aan de wachtrij.

[5 pt] ☐ 11. Geef een Haskell-functie

```
kleinste :: Integer -> [(Integer, Integer)] -> Integer
```

die bovenbeschreven algoritme implementeert.

[4 pt] ☐ 12. Geef een Haskell-functie die bij een positief geheel getal `n` de afstand van `n` oplevert.

lees verder

Opgave 3 (Numerieke expressies)

Bekijk de volgende grammatica voor numerieke expressies:

```
<expr>      ::=  <term>  | . . . . . .
<term>      ::=  <getal> | (<expr>) | <cond-expr>
<getal>     ::=  <cijf>  | <cijf><getal>
<cijf>      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<oper>      ::=  + | -
<cond-expr> ::=  [<bool-expr> ? <expr> : <expr>]
<bool-expr> ::=  LESS <expr> <expr> | ODD <expr>
```

Om de leesbaarheid te bevorderen spreken we af dat tussen non-terminale symbolen spaties mogen worden toegevoegd, met uitzondering van de cijfers in een getal.

De semantiek van de diverse (deel-)expressies ligt voor de hand. Zo staat de operator `+` voor optellen en `-` voor aftrekken.
Heeft in een conditionele expressie `[B ? E1 : E2]` de boolse expressie `B` de waarde `True`, dan is de waarde `E1`, anders de waarde `E2`.
Een boolse expressie `LESS A B` komt overeen met `A < B`; `ODD C` is `True` als `C` oneven is en anders `False`.

[7 pt] □ 13. Op de plaats van de stippeltjes in de eerste regel mag je kiezen tussen

`<expr> <oper> <term>`

en

`<term> <oper> <expr>`

Geef aan welke variant je kiest en waarom.

[8 pt] □ 14. Geef een afleidingsboom voor de expressie `19 + [LESS 7 8 - 2 ? 5 : 3]`

Expressies die aan bovenstaande grammatica voldoen, kunnen we in Haskell representeren met behulp van de volgende datastructuur:

```
data Expr = Num Integer |
            Plus Expr Expr | Minus Expr Expr |
            CondLess Expr Expr Expr Expr |
            CondOdd  Expr Expr Expr
```

[6 pt] □ 15. Geef bij deze data-definitie de Haskell-respresentatie van de expressie uit vraag 14.

[8 pt] □ 16. Geef een Haskell-functie `eval :: Expr -> Integer` die de waarde van een expressie berekent.

einde

totaal: 100 punten.

Woonplaats: Gouda | Jaar van eerste inschrijving: 2002 | Naam docent: Bakker

1

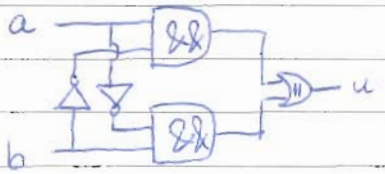
decimaal	binair	4-bits Gray-code
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1116
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

7 8 8

zie achterkant blad 2 voor vraag 2

~~$2^7 \leq 140 < 2^8$ dus er zijn hiervoor 8 bits nodig. Omdat het MSB niet overbodig is is de waarde hiervan 1. Van het volgende bit zou in de tabel de waarde 1 zijn als hij tot de middelste 2^7 bits behoort. Dit is waar, want $2^7 + 2^6 \leq 140 < 2^7 + 2^6$. Dus bit b is ook 1.~~

3

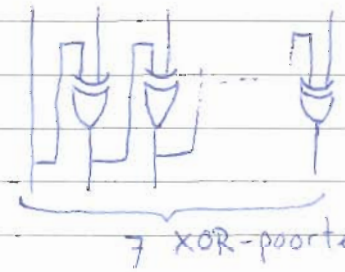


&& → AND
|| → OR

6

zie achterkant blad 2 voor vraag 2

4



$b_0 = g_0$ XOR b_1
$b_1 = g_1$ XOR b_2
⋮
$b_6 = g_6$ XOR b_7
$b_7 = g_7$

$g_7 = 1 \rightarrow b_7 = 1$
$g_6 = 1 \rightarrow b_6 = 0$
$g_5 = 0 \rightarrow b_5 = 0$
$g_4 = 1 \rightarrow b_4 = 1$
$g_3 = 0 \rightarrow b_3 = 1$
$g_2 = 1 \rightarrow b_2 = 0$
$g_1 = 0 \rightarrow b_1 = 0$
$g_0 = 1 \rightarrow b_0 = 1$

binair = 1001 1001

decimaal = 128 + 16 + 8 + 1 = 153

8

2.0.2.

5 xof :: Char → Char → Char
xof a b
4 | a == b = '0'
| otherwise = '1'

6 gray2bin :: [Char] -> [Char]
gray2bin a:[] = [a]
gray2bin (a:lst) = (a `xor` b) : b : brest
7 where (b:brest) = gray2bin lst

5 7 1 f→ 2 f→ 4 f→ 8 f→ 16 f→ 32 g→ 27 h→ 14

4 8 Een stack doorzoekt de boom met mogelijkheden 'depth-first', en aangezien er oneindig veel mogelijkheden zijn zou een stack meestal oneindig lang blijven doorrekenen. Hij zou bijvoorbeeld telkens één keer vakes op f drukken, wat in de meeste gevallen geen oplossing geeft. Een queue zoekt 'breadth-first' en heeft dit probleem dus niet.

9 enQueue :: (Integer, Integer) -> [(Integer, Integer)] -> hier geen regeleinde
5 [(Integer, Integer)]
enQueue (k, n) [] = [(k, n)]
enQueue (k, n) (a:lst) = a : (enQueue (k,n) lst)

hier geen regeleinde
10 enQueue Kids :: (Integer, Integer) → [(Integer, Integer)] → [(Integer, Integer)]
enQueue Kids (x, k) q
8 | x <= 5 = enQueue (2*x, k+1) q
| x <= 13 = enQueue (2*x, k+1) q
(enQueue (x-5, k+1) q)
| otherwise = enQueue (2*x, k+1)
(enQueue (x-5, k+1)
(enQueue (x-13, k+1) q))

11 kleinste :: Integer -> [(Integer, Integer)] -> Integer
kleinste n [] = "error" Stelling
5 kleinste n (a, b) : lst
| a == n = b
| otherwise = kleinste n (enQueue Kids (a,b) lst)

2 Hier gebruiken we het principe van vraag 4 maar dan omgekeerd:

$g_7 = b_7$
$g_6 = b_6$ XOR b_7
⋮
$g_0 = b_0$ XOR b_1

Dit is geldig dankzij de omkeerbaarheid van de XOR-poort, die blijkt uit zijn waarheidstabel: hierin kunnen inputs en outputs naar believen verwisseld worden zonder dat de werking van de poort verandert.

Dit levert op:

$b_7 = 1 \rightarrow g_7 = 1$
$b_6 = 0 \rightarrow g_6 = 1$
$b_5 = 0 \rightarrow g_5 = 0$
$b_4 = 1 \rightarrow g_4 = 1$
$b_3 = 1 \rightarrow g_3 = 0$
$b_2 = 0 \rightarrow g_2 = 1$
$b_1 = 0 \rightarrow g_1 = 0$
$b_0 = 1 \rightarrow g_2 = 1$

De gray-code van 140 is dus 11010101.

4

binaire notatie van $140_{10} = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 128 + 16 + 8 + 1$
$= 10011001_2$

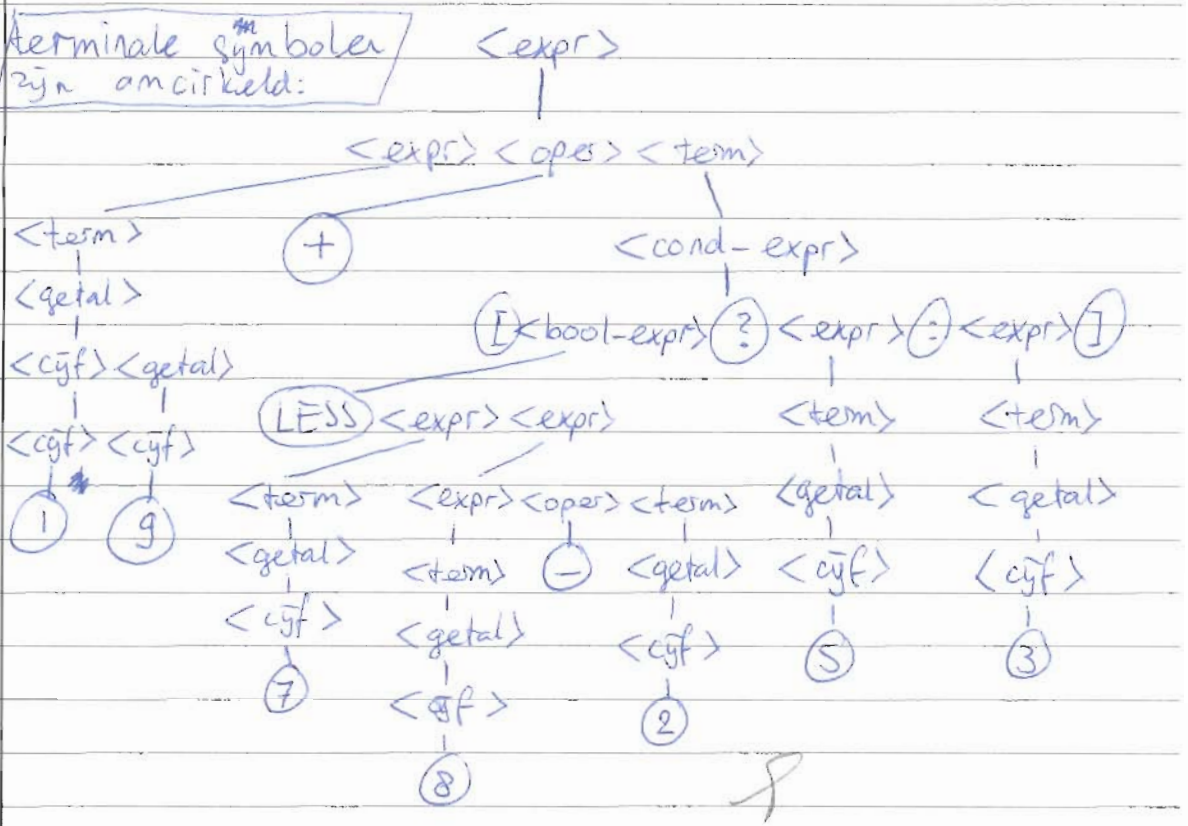
X

Woonplaats: 9616 PK Schaarsmo | Jaar van eerste inschrijving: 2002 | Naam docent: Bakker

4 12 afstand :: Integer -> Integer
afstand n = kleinste n [(1, 0)]

13 <expr> <oper> <term> is hier de beste keuze, omdat zo eerst het linker deel van de expressie wordt geëvalueerd, zoals dat in de wiskunde ook
7 gebeurt. Bijvoorbeeld:
~~10 - 5 + 3~~
betekent:
(10 - 5) + 3 = 8
en niet:
10 - (5 + 3) = 2

14 Terminale symbolen zijn omcirkeld:



8

15 (Plus ~~1~~ (Num 1)
(CondLess 7 idem
(Minus 8 ele.
2 ele.
4)
5
3
)
)

16
```
eval :: Expr -> Integer
eval (Num a) = a
eval (Plus a b) = (eval a) + (eval b)
eval (Minus a b) = (eval a) - (eval b)
eval (CondLess a b c d)
  | (eval a) < (eval b) = (eval c)
  | otherwise = (eval d)
eval (CondOdd a b c)
  | mod (eval a) 2 == 1 = (eval b)
  | otherwise = (eval c)
```

8

2a (vervolg van voorkant blad 1)
~~Het volgende bit (5) herhaalt zijn patroon twee keer; daarom kijken we of geldt:~~
$2^6 - 2^5 \leq 140 - 128 = 12 < 2^6 + 2^5$

Zie blad 3 voor vraag 2

Bit 7 (het MSB) is 1 als we in de tweede helft van de tabel met 8-bits Gray-codes zitten. Dit is waar, want $140 \geq 2^7 = 128$.
Bit 6 is 1 als we in de eerste helft van de tabel met 7-bits Gray-codes zitten (deze is immers gespiegeld omdat bit 7 gelijk was aan 1). Dit is waar, want $140 - 128 < 2^6 = 64$.
Bit 5 is 1 als we in de tweede helft van de tabel met 6-bits Gray-codes zitten (wederom een spiegeling), dus als $12 \geq 2^5 = 32$. Dit is niet waar, dus 0.
Bit 4 is 1 als we in de tweede helft van de tabel met 5-bits Gray-codes zitten (geen spiegeling), dus als $12 \geq 2^4 = 16$. Wederom een 0.
Bit 3 is 1 als we in de tweede helft van de tabel met 4-bits Gray-codes zitten, dus als $12 \geq 2^3 = 8$. Waar, dus een 1.
Bit 2 is 1 als we in de eerste helft van de tabel met 3-bits Gray-codes zitten, dus als $12 - 8 = 4 < 2^2 = 4$. Niet waar, dus 0.
Bit 1 → eerste helft 2-bits → $4 - 4 = 0 < 2^1 = 2$ → waar → 1
Bit 0 → tweede helft 1-bits → $0 \geq 2^0 = 1$ →