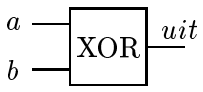
Tentamen
ORIENTATIE INFORMATICA
23 augustus 2000
14.00 – 17.00 uur

Niet elke opgave telt even zwaar. Bij elk onderdeel staat aangegeven hoeveel punten er maximaal gescoord kunnen worden. Bij 100 of meer punten is het tentamenresultaat een 10. In andere gevallen is het tentamenresultaat het aantal punten gedeeld door 10.
Bij de gevraagde algoritmen is het niet nodig een uitgebreide analyse te geven. We vragen wel enige toelichting.

Opgave 1

De *exclusive OR* (notatie XOR) is een electronische schakeling met de volgende functionaliteit:



a	b	uit
0	0	0
0	1	1
1	0	1
1	1	0

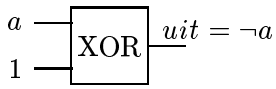
[7 pt] □ 1. Ontwerp systematisch een schakeling bestaande uit de standaard poorten (EN/AND, OF/OR, NIET/NOT) die deze schakeling implementeert.

uitwerking:

$$a \text{ XOR } b = (\neg a \ \wedge \ b) \ \vee \ (a \ \wedge \ \neg b)$$

[4 pt] □ 2. Met behulp van een XOR-poort is het mogelijk een NIET-schakeling te maken. Laat dit zien.

uitwerking:
Leeg één van de ingangen aan de 1:



Bij het onderwerp *foutdetecterende codes* hebben we gekeken naar *oneven pariteit*: alleen bitstrings met een oneven aantal 1-en zijn correct. Deze pariteitscontrole kan worden uitgevoerd door software, maar het is ook mogelijk een schakeling te bouwen die deze pariteitscontrole uitvoert.

[8 pt] □ 3. Ga uit van bitstrings ter lengte 4. Beschouw de vier bits als invoersignalen voor een controle-schakeling die uitvoer 1 geeft als het aantal 1-en oneven is en 0 als het aantal 1-en even is.
Ontwerp met behulp XOR-poorten zo'n controle-schakeling en toon aan dat je ontwerp

correct is.

uitwerking:

a	b	c	d	(a XOR b)	XOR	(c XOR d)	oneven par
0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1
0	0	1	0	0	1	1	1
0	0	1	1	0	0	0	0
0	1	0	0	1	1	0	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	0	1
1	0	0	0	1	1	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	0	0
1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	0	0	0

Uit deze tabel zien we dat

$$\text{oneven par}(a, b, c, d) = (a \text{ XOR } b) \text{ XOR } (c \text{ XOR } d)$$

Opgave 2

Deze opgave gaat over roosterpunten in het platte vlak en verzamelingen van dergelijke punten. Zo'n punt representeren we zoals gebruikelijk als een paar van gehele getallen. In Haskell declareren we daarvoor het type `Punt` als volgt

```
type Punt = (Integer, Integer)
```

Verzamelingen van punten representeren we met behulp van lijsten (`[Punt]`), waarbij we eisen dat er geen duplicaten in de lijst voorkomen. Als we in het vervolg praten over een verzameling, dan bedoelen we daar dus een lijst zonder duplicaten mee. We veronderstellen geen ordening in de lijst.

[6 pt] □ 4. Geef een functie `voegtoe :: Punt -> [Punt] -> [Punt]` zo, dat `(voegtoe x a)` de verzameling is die ontstaat door het element `x` toe te voegen aan de verzameling `a`.
NB: let goed op het vermijden van duplicaten.

uitwerking:

```
voegtoe :: Punt -> [Punt] -> [Punt]

voegtoe x [] = [x]
voegtoe x (a : lst)
  | x == a       = a : lst
  | x /= a       = a : (voegtoe x lst)
```

[7 pt] □ 5. Geef een functie `verenig :: [Punt] -> [Punt] -> [Punt]` zo, dat `(verenig a b)` de vereniging van de verzamelingen `a` en `b` oplevert.

uitwerking:

```
verenig :: [Punt] -> [Punt] -> [Punt]

verenig [] lst = lst
verenig (a : kst) lst = voegtoe a (verenig kst lst)
```

[7 pt] □ 6. Geef de worstcase tijdcomplexiteit van de functie `verenig` uit de vorige vraag. Licht je antwoord duidelijk toe.

uitwerking:
We kijken eerst naar de compelxiteit van `voegtoe`. Het aantal vergelijkingen dat worstcase moet worden uitgevoerd om een element `x` toe te voegen aan een lijst `lst` is gelijk aan het aantal elementen van `lst`.
Het verenigen van twee lijsten `kst` en `lst` komt overeen met het toevoegen `|kst|` elementen aan een steeds langer wordende lijst. Laat `n = |kst|` en `m = |lst|` dan vinden we voor het aantal vergelijkingen de uitdrukking:

```
n + (n + 1) + (n + 2) + . . . + (n + m - 1)
```

dit is gelijk aan $\frac{1}{2} \cdot m \cdot (2 \cdot n + m - 1)$.

We veronderstellen nu dat er een functie `hgt :: Punt -> Integer` bestaat. De waarde van `(hgt (u,v))` interpreteren we als de hoogte in het punt `(u,v)`. Op deze manier representeert `hgt` een 'landschap'. De bedoeling van de rest van de opgave is een functie te maken die bij een startpunt S, een punt P en een geheel getal n bepaalt of P vanuit S is te bereiken in *ten hoogste* n stappen via een strikt stijgend pad.

Het punt (x, y) is in één stap te bereiken vanuit het punt (u, v) als het een direct buurpunt is dat op een grotere hoogte ligt (notatie $(u, v) \to (x, y)$). In formules:

$$(u, v) \to (x, y) \quad \Leftrightarrow \quad (\, |x - u| + |y - v| = 1 \quad \wedge \quad hgt(x, y) > hgt(u, v) \,)$$

Hieronder vragen we je een aantal functies te construeren. In elk onderdeel mag je gebruik maken van de functies uit de vorige onderdelen, ook als je niet in staat bent geweest die te maken.
De functie `hgt` mag je globaal houden: je hoeft deze niet als parameter op te nemen bij de te construeren functies.

[6 pt] □ 7. Geef een functie `stap :: Punt -> [Punt]` die bij een punt P de lijst oplevert van alle punten die in één stap zijn te bereiken vanuit P.
HINT: maak eerst hulpfuncties `noordStap`, `oostStap`, `zuidStap` en `westStap`, van hetzelfde type als `stap`, die het elk voor de aangegeven richting doen. We vinden het voldoende als je één van deze hulpfuncties uitwerkt; de rest is toch analoog.

uitwerking:

```
noordStap :: Punt -> [Punt]

noordStap (u,v)
  | hgt (u,v+1) > hgt (u,v)  = [(u,v+1)]
  | otherwise                = []

stap :: Punt -> [Punt]

stap (u,v) = (noordStap (u,v)) ++ (oostStap (u,v)) ++
             (zuidStap (u,v)) ++ (westStap (u,v))
```

[5 pt] □ 8. Geef een functie `stapLijst :: [Punt] -> [Punt]` die bij een lijst `lst` van punten de lijst oplevert van punten die in één stap bereikbaar zijn vanuit punten van `lst`.

uitwerking:

```
stapLijst :: [Punt] -> [Punt]

stapLijst [] = []
stapLijst (p : lst) = verenig (stap p) (stapLijst lst)
```

[7 pt] □ 9. Geef een functie `isPad` zo, dat `(isPad start p n)` als resultaat oplevert of het punt `p` bereikbaar is vanuit het punt `start` in ten hoogste `n` stappen.
Je zult hiervoor een aantal hulpfuncties moeten maken. Geef bij elke functie het type en een (korte) toelichting.

uitwerking:

```
zitIn :: Punt -> [Punt] -> Bool
zitIn p [] = False
zitIn p (a : lst)
  | p == a       = True
  | p /= a       = zitIn p lst

zoekPad :: [Punt] -> Punt -> Integer -> Bool
zoekPad ( lst pnt n )
  | n  < 0           = False
  | zitIn pnt lst    = True
  | otherwise        = zoekPad (stapLijst lst) pnt (n - 1)

isPad :: Punt -> Punt -> Integer -> Bool
isPad start pnt n = zoekPad [start] pnt n
```

Bekijk nu het volgende beslissingsprobleem

Bereikbaarheid (REACH)
Parameter: Twee roosterpunten S en P en een Haskell functie
`hgt :  Punt -> Integer`
Gevraagd: Is P bereikbaar vanuit S via een strikt stijgend pad?

[5 pt] □ 10. Leg uit wat het betekent dat een beslissingsprobleem beslisbaar is.

uitwerking:
Een beslissingsprobleem P heet beslisbaar als er een algoritme A bestaat dat bij elke instantie I van P binnen eindige tijd uitmaakt of I een ja-instantie of een nee-instantie is.

[6 pt] □ 11. Is (**REACH**) beslisbaar? Beargumenteer duidelijk je antwoord.

uitwerking:
We kunnen het algoritme uit de voorgaande onderdelen eenvoudig aanpassen zodat er een algorime voor (**REACH**) ontstaat:

```
reach :: Punt -> Punt -> Bool
reach s p = isPad s p ( (hgt p) - (hgt s) )
```

Immers, als het pad strikt stijgend is, dan is het maximaal aantal stappen dat gezet kan worden gelijk aan het verschil in hoogte.
Maar dan is (**REACH**) dus beslisbaar.

Opgave 3

Laat $G = (V, E)$ een graaf zijn met knopenverzameling V en kantenverzameling E. Onder een *kliek* in G verstaan we een verzameling knopen $K \subseteq V$ waarvan elk tweetal elementen met elkaar is verbonden door een kant $t \in E$.

Bekijk nu het volgende beslissingsprobleem

(KLIEK)
Parameter: Een graaf $G = (V, E)$ en een natuurlijk getal n
Gevraagd: Bevat G een kliek met minstens n elementen?

[4 pt] □ 12. Geef een ja-instantie van (**KLIEK**).

uitwerking:
Laat G een totaal verbonden graaf zijn met 5 knopen. Dan is $(G, 5)$ een ja-instantie.

[4 pt] □ 13. Geef een nee-instantie van (**KLIEK**)

uitwerking:
Laat G een totaal verbonden graaf zijn met 5 knopen. Dan is $(G, 6)$ een nee-instantie.

[5 pt] □ 14. Beargumenteer dat (**KLIEK**) $\in$ **NP**

uitwerking:
Een certificaat bij (**KLIEK**) is een deelverzameling H van de knopen van de parameter G. We moeten nu nagaan of H minstens n elementen heeft, en of elk tweetal knopen in H door een kant in G zijn verbonden. De eerste controle is uit te voeren door gewoon te tellen. Dat kan in lineaire tijd. Bij n knopen zijn er $\frac{1}{2} \cdot n \cdot (n-1)$ paren van (verschillende) knopen die geverifieerd moeten worden. Dit zijn allemaal controles die in polynomiale tijd zijn uit te voeren.
Maar dat zit (**KLIEK**) in **NP** .

[5 pt] □ 15. Er is te bewijzen dat (**KLIEK**) $\in$ **NPC** door gebruik te maken van het feit dat (**3SAT**) $\in$ **NPC**. Leg uit in welke richting de reductie moet gaan en wat de bewijsverplichtingen zijn.

uitwerking:
Reduceer (**SAT**) naar (**KLIEK**).
De bewijsverplichtingen zijn: de reductie zelf is van polynomiale tijdcomplexiteit en beeldt ja-instanties af op ja-instanties en nee-instanties op nee-instanties.

Opgave 4

Bekijk de volgende grammatica voor een zeer eenvoudige programmeertaal:

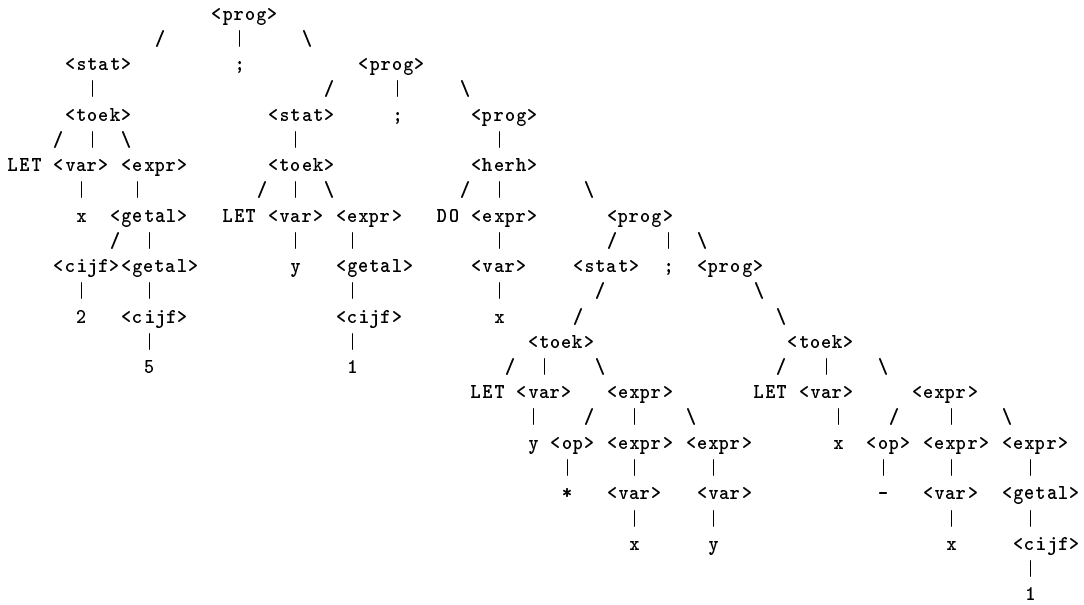
```
<prog>     ::=   <stat> | <stat> ';' <prog>
<stat>     ::=   <toek> | <keuze> | <herh>
<toek>     ::=   'LET' <var> <expr>
<keuze>    ::=   'IF' <expr> <prog>
<herh>     ::=   'DO' <expr> <prog>
<var>      ::=   'x' | 'y' | 'z'
<expr>     ::=   <var> | <getal> | <op> <expr> <expr>
<getal>    ::=   <cijfer> | <cijfer><getal>
<cijfer>   ::=   '0' | '1' | '2' | '3' | '4' |
                 '5' | '6' | '7' | '8' | '9'
<op>       ::=   '+' | '-' | '*'
```

We veronderstellen dat, met uitzondering van de cijfers in getallen, in de te produceren strings de terminalen zijn gescheiden door spaties.

[8 pt] □ 16. Geef een ontleedboom bij

```
LET x 25 ; LET y 1 ; DO x LET y * x y ; LET x - x 1
```

uitwerking:



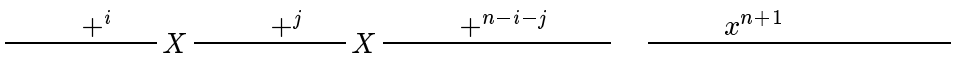
[5 pt] □ 17. Deze grammatica is ambigu. Toon dat aan.

uitwerking:
Het programma uit het vorige onderdeel kan ook met een andere boom worden opgebouwd: het statement `LET x - x 1` kan ook buiten de `DO` worden gehouden.
Dat houdt in dat er bij eenzelfde string twee verschillende syntaxbomen bestaan. Maar dan is de grammatica ambigu.

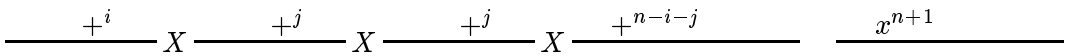
[6 pt] □ 18. Is er een eindige automaat bestaat die bepaalt of de invoer voldoet aan `<expr>`? Beargumenteer duidelijk je antwoord.
(LET OP: we vragen naar `<expr>` en niet naar `<prog>`.)

uitwerking:
STEL er is wel zo'n eindige automaat M. Laat dan n het aantal toestanden van M zijn. Bekijk nu de expressie $\texttt{+}^n\texttt{x}^{n+1}$. Het is eenvoudig in te zien dat deze string vanuit het hulpsymbool `<expr>` is te genereren.
Bekijk nu de 'wandeling' van deze string langs de toestanden van de automaat:

In de verwerking van $+^n$ passeren we $n+1$ toestanden. Er zijn echter maar n toestanden, dus ergens op dit traject zijn er twee toestanden gelijk:



Maar dan is ook een mogelijke wandeling:



Maar dat betekent dat M ook de string $+^{n+j}x^{n+1}$ accepteert. Maar dat is geen string die binnen deze grammatica gegenereert kan worden vanuit `<expr>`.
TEGENSPRAAK: dus zo'n automaat bestaat niet.

[5 pt] □ 19. Is er een Turingmachine die bepaalt of de invoer voldoet aan `<prog>`? Beargumenteer je antwoord.

uitwerking:
Er is wel een programma te maken die deze controle uitvoert. De these van Church-Turing zegt dan dat er ook wel een Turingmachine voor is te schrijven.

einde