Flippos — de nieuwe rage

Maarten Fokkinga

Versie van 2 april 1996

Inleiding. In iedere zak chips (van het juiste merk) zitten één of meer *flippos*: kleine ronde kunststof schijfjes. Je kunt ermee bouwen (de *techno*flippos), gooien (de gewone flippos), ruilen of gewoon sparen (hoe meer hoe beter, vindt de chips-fabrikant). Op sommige staat zelfs een puzzel; en daar gaat hier over. Op zo'n puzzel-flippo staan vier cijfers waarmee je een sommetje moet construeren met 24 als uitkomst; elk cijfer moet precies eenmaal in het sommetje voorkomen (als getal), en de bewerkingen $\times$, $/$, $+$, $-$ mogen daarbij worden gebruikt. Bijvoorbeeld, met de cijfers $1, 4, 7, 7$ kunnen we het sommetje $(1+7)\times(7-4)$ maken met 24 als uitkomst.

Wij zullen nu een script maken waarmee flippo-puzzels zijn op te lossen en te genereren. Ook willen we alle flippos bepalen die in essentie maar één oplossing hebben; bijvoorbeeld, $(1+7)\times(7-4)$ is in essentie hetzelfde als $(7-4)\times(7+1)$ omdat vermenigvuldiging en optelling commutatief zijn.

De volgende opgaven geven je enig gevoel voor de zaak:

Opgaven

1. Geef een recht-toe recht-aan expressie voor de bepaling van *het aantal* drietallen operatoren $(o1, o2, o3)$ met de eigenschap dat $((3\ \underline{o1}\ 3)\ \underline{o2}\ 7)\ \underline{o3}\ 7$ als uitkomst 24 heeft.
2. Geef nu niet het aantal drietallen $(o1, o2, o3)$, maar de drietallen zelf in een of andere representatie.
3. Geef een recht-toe recht-aan expressie voor de bepaling van *het aantal* drietallen $(o1, o2, o3)$ met de eigenschap dat $((a\ \underline{o1}\ b)\ \underline{o2}\ c)\ \underline{o3}\ d$ als uitkomst 24 heeft, voor een of andere permutatie $[a, b, c, d]$ van $[3, 3, 7, 7]$.
4. Geef een recht-toe recht-aan uitdrukking voor de bepaling van een zevental (of alle zeventallen) $(a, o1, b, o2, c, o3, d)$ met de eigenschap $((a\ \underline{o1}\ b)\ \underline{o2}\ c)\ \underline{o3}\ d = 24$, waarbij $[a, b, c, d]$ een of andere permutatie is van $[3, 3, 7, 7]$, en $o1, o2, o3$ rekenkundige operaties zijn.
5. Op hoeveel manieren kunnen er haakjes gezet worden in $a\ \underline{o1}\ b\ \underline{o2}\ c\ \underline{o3}\ d$?
6. Hoeveel verschillende expressies kunnen er gevormd worden (met vier operanden uit $1\,..\,9$, drie operaties uit *maal*, *deel*, *plus*, *minus*, en de haakjes-plaatsing willekeurig)?
7. (Alleen voor de liefhebbers.) Hoeveel verschillende flippos bestaan er? Bedenk dat '3,3,7,7' en '3,7,3,7' en '7,3,3,7' etcetera verschillende notaties zijn voor eenzelfde flippo!

Het is best mogelijk om nu voor flippos (viertallen uit $1\,..\,9$) de oplossing uit te programmeren op een ad-hoc manier. Maar we willen *natuurlijk* ook algemenere puzzels oplossen (n-tallen uit een verzameling xs). Dit noodzaakt ons om het een en ander beter te begrijpen, en heeft als potentieel voordeel dat het resulterende script *korter* is en zich beter leent voor een correctheids-argumentatie.

De rest van dit verhaal bestaat in feite uit de uiteindelijke versie van mijn eigen script. Sommige (veel) definities zijn als opgave gegeven. Opgaven in verschíllende paragrafen (genummerd met §...) zijn onafhankelijk van elkaar. In een praktikum-situatie kun je dus met meerdere personen tegelijk werken om het script te voltooien. Wel moet je voorgaande paragrafen doorlezen voor gemeenschappelijke definities en afspraken. Het totale script, maar zonder de delen die als opgave worden opgegeven, staat nog eens opgesomd in Aanhangsel A op bladzijde 11.

Het script

§**1** **Het hoogste nivo.** Laten we eerst wat terminologie vastleggen:

> || *Een 'flippo' is een bag ter grootte* 4 *over de getallen* 1 . . 9;
> || *de opgave is met deze getallen een expressie te vormen met uitkomst* 24,
> || *waarbij ieder getal precies een keer gebruikt wordt.*

Een *bag* (Engels voor: zak) is een 'verzameling met duplicaten' ofwel een 'een lijst waarin de volgorde niet ter zake doet'. Bags vormen een belangrijk begrip in de informatica (misschien nog wel een belangrijker begrip dan dat van verzameling en lijst!). In deze verhandeling geldt:

> || *Verzamelingen worden gerepresenteerd als lijsten zonder duplicaten.*
> || *Bags worden gerepresenteerd als lijsten met zo nodig duplicaten.*

Dus $[3, 3, 7, 7]$ en $[7, 3, 3, 7]$ stellen dezelfde bag voor, terwijl $[3, 3, 7, 7]$ en $[3, 3, 7]$ verschillende bags voorstellen. Geen van die vier lijsten representeert een verzameling, want er komen duplicaten in voor. We zullen geen *gebruik* maken van enige ordening in bags of verzamelingen. Het is echter wel toegestaan om bij het produceren van bags en verzamelingen misschien wel ordening te handhaven (om duplicaten te voorkomen, of zo).

We kunnen nu alvast de parameters van het script vastleggen, en enige constanten. Sommige grootheden worden verder op gedefinieerd: *maal* t/m *minus* in §3, en de cruciale functies *oplossingen* in §5 en *bags* in §7.

```
|| Script parameters en constanten :
flippoLengte = 4
flippoGetallen = [1 .. 9]
flippoUitkomst = 24
flippoOperaties = [maal, deel, plus, minus]

flippoOpln xs = oplossingen flippoUitkomst flippoOperaties xs
flippoVerz = bags flippoLengte flippoGetallen || = de verz van alle flippos
```

Nu kunnen we de functies definiëren waarin we geïnteresseerd zijn; mits we aannemen dat *showEx* een expressie netjes toont en dat *flippoOpln xs* een verzameling van *expressies* oplevert:

$$
\begin{aligned}
&opln\ xs\\
&\quad = \texttt{"}\mathit{Alle\ oplossingen\ voor\ flippo}\ \texttt{"}\ +\!\!+\ show\ xs\ +\!\!+\ \texttt{"}:\backslash n\texttt{"}\ +\!\!+\\
&\qquad (layn\ .\ map\ showEx\ .\ flippoOpln)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&flippos\ k\\
&\quad = \texttt{"}\mathit{Alle\ flippos\ met\ precies}\ \texttt{"}\ +\!\!+\ show\ k\ +\!\!+\ \texttt{"}\ \mathit{oplossingen}:\backslash n\backslash n\texttt{"}\ +\!\!+\\
&\qquad layn\\
&\qquad [show\ xs\ +\!\!+\ \texttt{"}:\ \texttt{"}\ +\!\!+\ concat\ (map\ showEx\ opln)\ |\\
&\qquad xs \leftarrow flippoVerz;\ opln \leftarrow [flippoOpln\ xs];\ \#opln\ =\ k]
\end{aligned}
$$

Merk op dat het deel '$opln \leftarrow [flippoOpln\ xs]$' een locale definitie van *opln* is: hierna geldt dat *opln* gelijk is aan *flippoOpln xs*. Met deze definitie worden, per flippo *xs*, alle *k* oplossingen op één regel getoond. Ze komen ieder op een aparte regel, voldoende ingesprongen, als we *concat* vervangen door *chain* ("\n   "), zie opgave 24.

Om te bepalen of een flippo 'in essentie' maar één oplossing heeft, veronderstellen we het bestaan van een functie *sv* die expressies tot een standaardvorm (sv) transformeert, en wel zó dat 'in essentie' gelijke expressies eenzelfde standaardvorm hebben (zie §6):

Opgaven

8. Neem het bestaan van *sv* aan; geef een definitie voor de functie:

   $\|\ oplnSv\ xs\ =\ \mathit{de\ verz\ van\ alle\ sv\ oplossingen\ voor\ flippo\ xs}$

   Hint: $mkset\ xs$ = een lijst gevormd uit *xs* door schrapping van duplicaten.
9. Geef, weer van *sv* gebruik makend, een definitie voor de functie:

   $\|\ flipposSv\ k\ =\ \mathit{de\ verz\ van\ alle\ flippos\ met\ precies\ k\ sv\ oplossingen}$

Alreeds in dit stadium kunnen we denken aan allerlei varianten van de flippo-puzzel:

Opgaven

10. Ga na welke van de volgende problemen opgelost kunnen worden door aanpassing van de hierboven gedefinieerde grootheden:
    1. Welke flippos hebben geen oplossing?
    2. Welke flippos met 1 oplossing bestaan uit vier verschillende getallen?
    3. Welke flippos met 1 oplossing bestaan uit een reeks $n\ ..\ n{+}3$?
    4. Welke flippos kunnen opgelost worden met drie verschillende operaties?
    5. Kan een gegeven flippo *xs* opgelost worden met één operatie (driemaal toegepast)?

§**2 Het laagste nivo: de data-representatie.** In het voorgaande hebben we *top down* gewerkt: we hebben definities gegeven voor functies op het hoogste nivo, waarbij we andere functies bekend verondersteld hebben (dat zijn dan functies op een lager nivo). We gaan nu even *bottom up* te werk: functies en definities op het laagste nivo; deze gebruiken geen andere grootheden, maar worden eventueel zelf wel gebruikt in andere definities.

Op het laagste nivo wordt vastgelegd hoe we de data representeren. Hier horen ook de functies thuis die moeten wijzigen wanneer we de representatie zouden wijzigen.

$$
\begin{array}{l}
\| \; Datastructuren: \; operator, \; expressie. \\
\\
\| \; Bij \; elke \; datastructuur \; data \; definieren \; we \; (voor \; testdoeleinden) \; de \; functies \\
\| \quad showData \; :: \; data \to [char] \\
\| \quad readData \; :: \; [char] \to data \\
\| \; met \; hulpfunctie \\
\| \quad readData' \; :: \; [char] \to (data, \; [char]) \\
\| \\
\| \; Voorbeeld: \\
\| \; showEx \; e \; = \; \texttt{"}(1 + (2\times 3))\texttt{"} \\
\| \; readEx \; \texttt{"}(1 + (2\times 3))\texttt{"} \; = \; e
\end{array}
$$

Functies *show*... zorgen voor een nette afdruk. Functies *read*... zijn ongeveer de inverse van *show*...; zij produceren de Miranda-representatie uit een string. Met behulp van *read*... kun je test-data gemakkelijk invoeren. De accent-versies leveren een tweetal op: het tweede lid is de rest van de string. Bijvoorbeeld:

$$
\| \; readEx' \; \texttt{"}(1 + (2\times 3))blahblahblah\texttt{"} \; = \; (e, \; \texttt{"}blahblahblah\texttt{"})
$$

§**3 Operatoren.** We kiezen getallen ter representatie van operatoren. Dit doen we niet met een type-synoniem ("*operator* $\equiv$ *num*") maar met een algebraïsch type ("*operator* ::= *Op num*"), zodat onbedoeld gebruik (van een operator op de plaats van een getal) door de compiler aangemerkt wordt als een type-fout.

$$
\begin{array}{l}
operator \; ::= \; Op \; num \\
maal \; = \; Op \; 0 \\
deel \; = \; Op \; 1 \\
plus \; = \; Op \; 2 \\
minus \; = \; Op \; 3 \\
\| \; exp, \; log, \; div, \; mod, \; \ldots
\end{array}
$$

Hierbij horen de definities die van deze representatie gebruik maken:

$$
\begin{array}{l}
showOp \; (Op \; o) \; = \; [\texttt{"}\times\texttt{"}, \; \texttt{"}/\texttt{"}, \; \texttt{"}+\texttt{"}, \; \texttt{"}-\texttt{"}] \; ! \; o \\
readOp' \; (x : xs) \; = \; (hd \; [o \mid o \leftarrow [maal, deel, plus, minus]; \; showOp \; o \; = \; [x]], \; \; xs)
\end{array}
$$

Er zijn talloze alternatieve formuleringen voor *readOp'*.

Opgaven

11. Definieer de volgende functies: *evalOp*, *prio*, *assoc*, *comm*. De gewenste eigenschap wordt (voldoende) gesuggereerd door deze voorbeelden:

$$\begin{array}{l} \mathit{evalOp\ maal} \ = \ (\times) \\ \mathit{prio\ maal} \ < \ \mathit{prio\ deel} \ < \ \mathit{prio\ plus} \ = \ \mathit{prio\ minus} \\ \mathit{assoc\ maal} \ = \ \mathit{True} \\ \mathit{assoc\ deel} \ = \ \mathit{False} \\ \mathit{comm\ plus} \ = \ \mathit{True} \\ \mathit{comm\ minus} \ = \ \mathit{False} \end{array}$$

§**4 Expressies.** We kiezen voor de volgende representatie:

$\|$ *Vb* : $3 + (4{\times}5)$:=: *OpEx* (*NumEx* 3) *plus* (*OpEx* (*NumEx* 4) *maal* (*NumEx* 5))

Dus per operatie een *OpEx* en per getal een *NumEx*:

expressie ::= *NumEx num* | *OpEx expressie operator expressie*

Voor het tonen van expressies zijn er deze twee hulpfuncties:

behaakt x = `"("` ++ x ++ `")"`
onhaakt x = `" "` ++ x ++ `" "`

Opgaven

12. Definieer een functie *showE* die een expressie toont — met haakjes bij elke operatie, zodat ambiguïteiten uitgesloten zijn.

Het is niet nodig, maar wel een uitdaging, om een expressie met zo min mogelijk haakjes te tonen:

(a) In $(3/4)+\ldots$ en $\ldots + (3/4)$ zijn de haakjes overbodig, maar niet in $(3/4)\times\ldots$. Dit berust op de *prioriteitsregels* (hier: *prio maal* < *prio deel* < *prio plus*).

(b) De haakjes in $(3 + 4)+\ldots$ en $\ldots + (3 + 4)$ zijn overbodig. Dit berust op de associativiteit van *plus* (hier: *assoc plus* = *True*).

(c) De haakjes in $(2 + 3) - 4$ zijn overbodig. Dit berust op de afspraak dat bij operatoren van gelijke prioriteit de ontbrekende haakjes naar links toe genest toegevoegd moeten worden (hier: *prio plus* = *prio minus*).

In alle drie de gevallen is voor het tonen van een operand kennis nodig over de bijhorende (omvattende) operator. Functies *showExL* en *showExR*, hieronder, hebben daarom niet alleen de te tonen expressie als argument, maar ook de 'bijhorende, omvattende' operator:

$$\begin{array}{l}
\| \; showEx\ x \;\; toont\ x\ met\ weinig\ (ihb\ geen\ omvattende)\ haakjes :\\
showEx\ (NumEx\ x)\ =\ shownum\ x\\
showEx\ (OpEx\ x\ o\ y)\ =\ showExL\ o\ x\ +\!\!+\ showOp\ o\ +\!\!+\ showExR\ o\ y
\end{array}$$

$$\begin{array}{l}
\| \; showExL\ o\ x\ toont\ x\ (als\ een\ Linker\ operand\ van\ o,\ zo\ nodig\ met\ haakjes) :\\
showExL\ o'\ (NumEx\ x)\ =\ shownum\ x\\
showExL\ o'\ (OpEx\ x\ o\ y)\\
\ =\ onhaakt\ (showEx\ (OpEx\ x\ o\ y)),\ if\ ?????\\
\ =\ behaakt\ (showEx\ (OpEx\ x\ o\ y)),\ otherwise
\end{array}$$

Opgaven

13. Geef een voorwaarde die hierboven op de plaats van de vraagtekens kan staan. Voorwaarde *False* is correct, maar niet goed, want daarmee worden haakjes *nooit* weggelaten. Voorwaarde *True* is fout; dan worden haakjes *altijd* weggelaten. Bedenk dus een voorwaarde tussen deze extremen in.
14. Geef een definitie voor de functie met specificatie:

$$\| \; showExR\ o\ x\ toont\ x\ (als\ een\ Rechter\ operand\ van\ o,\ zo\ nodig\ met\ haakjes) :$$

Nu het omgekeerde van *show*...: de *read*... functies. Deze functies worden niet gebruikt in het uiteindelijke flippo-script. Ze zijn alleen maar handig om tijdens de programma-constructie het een en ander te testen. De definities volgen een standaard techniek; als je die eenmaal door hebt, zijn dergelijke functies in de toekomst een makkie.

Herinner je dat —volgens de bedoeling— $readEx'$ "$(2{\times}(3+4))xxx$" $=$ $(e,$ "xxx"$)$, waarbij e de expressie $(2{\times}(3+4))$ voorstelt. Het tweede lid van de uitkomst van $readEx'\ xs$ is dus het restant van xs dat overblijft als de expressie-notatie vóóraan in xs eraf gehaald wordt.

$$\begin{array}{l}
\| \; readEx\ vereist\ een\ volledig\ behaakte\ notatie\ als\ invoer :\\
readEx\ xs\ =\ e \quad where\ (e,\ zs)\ =\ readEx'\ xs\\
readEx'\ (x : xs)\\
\ =\ (NumEx\ ?????,\ xs),\ if\ digit\ x\\
\ =\ (OpEx\ e\ o\ e',\ zs),\ if\ x\ =\ '('\\
\quad where\ ?????
\end{array}$$

Opgaven

15. Voltooi bovenstaande definitie van $readEx'$.
16. Hoe moet je $readOp'$ en $readEx'$ aanpassen opdat er ook 'leidende' spaties toegestaan zijn? Bijvoorbeeld: $readEx'$ "$\ (2+\ \ (\ 3\ {\times}4)\ )xxx$" $=$ $(e,$ "xxx"$)$ voor geschikte expressie e.

17. Hoe moet je *readEx* aanpassen zodat een foutmelding wordt gegeven indien het argument méér is dan een expressie gevolgd door spaties? Bijvoorbeeld, *readEx* "$(2\times(3+4))5$" en *readEx* "$(2\times(3+4))$ 5" eindigen dan met een foutmelding, maar *readEx* "$(2\times(3+4))$ " niet.

Hiermee is het laag-nivo werk (met heel wat tierlantijne) afgehandeld. We komen nu toe aan de flippo-essentie: de functie *oplossingen* die bij gegeven flippo *xs* alle expressies met *xs* als getallen en de juiste uitkomst genereert, de functie *sv* die een expressie in een standaardvorm zet, en de functie *bags* die alle bags over een verzameling oplevert.

§5 Het genereren en testen van expressies. De cruciale functie is *oplossingen*:

```
|| oplossingen out ops xs = de verzameling van alle expressies met
|| operanden xs (in WILLEKEURIGE volgorde; xs fungeert als bag), haakjes
|| willekeurig, operaties uit ops en uitkomst out :
oplossingen out ops xs
 = (filter correct . genereer ops) xs
   where
   correct ys = eval ys = (True, out)
```

Hierin is *eval* de functie die een expressie evalueert; er wordt een tweetal opgeleverd dat aangeeft of de expressie een waarde heeft (dat wil zeggen, er wordt niet gedeeld door nul), en wat de waarde is:

```
|| eval e = (bool : e heeft een waarde, num : de waarde van e)
|| Bij deling door nul heeft een expressie geen waarde!
eval (NumEx x) = (True, x)
eval (OpEx x o y) = ???
```

Opgaven

18. Voltooi bovenstaande definitie van *eval*. Gebruik een where-part om de onderdelen van *eval x* en *eval y* te benoemen.

Het genereren van alle mogelijke expressies, gegeven een *bag xs* van getallen, is misschien nog wel het moeilijkste onderdeel van het hele flippo-script.

Mijn idee is als volgt. Eerst definieer ik een hulpfunctie *gen* die expressies vormt die *van links naar rechts* precies de getallen uit een *lijst xs* bevatten. Bijvoorbeeld, bij lijst $xs = [3,3,7,7]$ en $\#ops = 1$ levert *gen ops xs* een lijst van lengte 5: er zijn vijf manieren om haakjes te plaatsen in 3 $\underline{o}$ 3 $\underline{o}$ 7 $\underline{o}$ 7. (Algemener, bij diezelfde *xs* en $\#ops = n$ geldt $\#gen\ ops\ xs = 5\times n^3$.) Het is dan een apart probleem om uit een *bag xs* alle relevante *lijsten* xs' te vormen, en die aan *gen* te onderwerpen.

```
|| gen ops xs = de verzameling van alle expressies met operanden xs
|| (in deze VASTE volgorde; xs fungeert als lijst), haakjes willekeurig,
|| en operaties uit ops :
```

$gen\ ops\ [x]\ =\ [NumEx\ x]$
$gen\ ops\ xs\ =\ [OpEx\ e\ o\ e'\ |\ \ldots e{\leftarrow}\ldots e'{\leftarrow}\ldots o{\leftarrow}\ldots]$

$\|$ *genereer ops xs = de verzameling van alle expressies met*
$\|$ *operanden xs (in WILLEKEURIGE volgorde; xs fungeert als bag), haakjes*
$\|$ *willekeurig, operaties uit ops :*
genereer ops xs = ?????

Opgaven

19. Voltooi de definitie van *gen*. Bedenk dat *xs* in twee delen gesplitst moet worden: het ene deel wordt gebruikt om e te genereren, het andere deel om e' te genereren.
20. Voltooi de definitie van *genereer*. Hint: definieer een “standaard” functie *perms* die, gegeven een bag *xs*, alle permutaties van *xs* oplevert. Als de bag duplicaten bevat, komen sommige permutaties misschien meermalen voor; die kun je verwijderen met *mkset*. Het is ook goed mogelijk om *perms* direct, zonder *mkset*, te definiëren.

§**6 Standaardvorm.** We willen $(3+\ 3/7\ )\times 7$, $(\ 3/7\ +3)\times 7$, $7\times(3+\ 3/7\ )$ en $7\times(\ 3/7\ +3)$ niet als verschillende oplossingen beschouwen van de flippo $3,3,7,7$. Immers, die expressies zijn aan elkaar gelijk volgens de algemene wetten

$x+(y+z)=(x+y)+z$	associativiteit van $+$, en net zo voor $\times$;
$x+y=y+x$	commutativiteit van $+$, en net zo voor $\times$.

Dus hoewel flippo $3,3,7,7$ ogenschijnlijk vier oplossingen heeft, willen wij zeggen dat er in essentie maar één oplossing is.

We vermijden verschillende maar in essentie gelijke oplossingen, door de expressies eerst tot een standaardvorm te transformeren (die voor in essentie gelijke expressies identiek is!), en dan dubbele voorkomens te verwijderen met *mkset*. Dit is al geprogrammeerd in de definities van *oplnSv* en *flipposSv* in opgave 8 en 9. Wij gaan hier de functie *sv* construeren.

Hoe construeren we uit een expressie een standaardvorm? In een eerste poging om *sv* te definiëren, liet ik eerst alle mogelijke associativiteits-transformaties van de vorm $(x+(y+z)) \mapsto ((x+y)+z)$ uitvoeren, gevolgd door alle mogelijke transformaties van de vorm $(x+y) \mapsto (y+x)$ (indien $y<x$ voor een of andere, willekeurige maar vast gekozen ordening $<$). (Voor de ordening kiezen we Miranda’s standaard ordening; die werkt op alle typen.) Maar dit gaat fout: de twee expressies $(((1+2)+3)+4)$ en $(((3+4)+2)+1)$ veranderen door deze aanpak niet, en hun ‘standaardvormen’ zijn dan verschillend.

Een andere aanpak is als volgt. Beschouw als voorbeeld een expressie van de vorm $(((a+b)+(c+d))+(d+e))$, waarbij a,b,c,d,e niet van de vorm $x+y$ zijn. De standaardvorm hiervoor is dan $((((a'+b')+c')+d')+e')$ waarbij $[a',b',c',d',e'] = sort\ [a,b,c,d,e]$. De standaardvorm is gelijk aan de oorspronkelijke expressie op grond van assocoiativiteit én commutativiteit van optelling. Is de operatie alleen associatief, en niet commutatief, dan moet de ‘*sort*’ achterwege gelaten worden. Is de operatie alleen commutatief, en niet associatief, dan moet steeds het ‘verzamelen van alle operanden’ beperkt worden tot ‘het tweetal operanden’. Dus we definiëren:

$sv\ (OpEx\ e\ o\ e')$
$=\ hervormAC\ o\ (OpEx\ e\ o\ e'),\ if\ assoc\ o\ \wedge\ comm\ o$
$=\ hervormA\ \ o\ (OpEx\ e\ o\ e'),\ if\ assoc\ o\ \wedge\ \neg comm\ o$ $\quad\|\ overbodig\ bij\ flippos$
$=\ hervormC\ \ o\ (OpEx\ e\ o\ e'),\ if\ \neg assoc\ o\ \wedge\ comm\ o$ $\quad\|\ overbodig\ bij\ flippos$
$sv\ (OpEx\ e\ o\ e')\ =\ OpEx\ (sv\ e)\ o\ (sv\ e')$
$sv\ (NumEx\ n)\ =\ (NumEx\ n)$

$\|\ hervormA\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ e's\ operator\ o\ accoc\ is)$
$\|\ hervormC\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ e's\ operator\ o\ comm\ is)$
$\|\ hervormAC\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ o\ accoc\ en\ comm\ is)$

Geen van de flippo-operatoren *maal*, *deel*, *plus*, *minus* is alleen associatief of alleen commutatief. We hebben de betreffende clausules toegevoegd voor de algemeenheid (en om daardoor tot een beter begrip van de zaak te komen). De laatste twee clausules doen niets anders dan *sv* doorschuiven naar de onderdelen van de expressie. Waren dit de enige twee clausules geweest, dan was *sv* de identiteit, die z'n argument onveranderd als resultaat oplevert.

Opgaven

21. Definieer de hulpfuncties:

$\|\ destruct\ o\ e\ =\ een\ bag\ [e0, e1, \ldots]\ van\ expressies\ zo\ dat:$
$\|\quad e\ =\ e0\ \underline{o}\ e1\ \underline{o}\ \ldots\ (met\ haakjes\ op\ een\ of\ andere\ manier\ geplaatst),\ en$
$\|\quad geen\ van\ e0,\ e1, \ldots\ is\ een\ expressie\ met\ operator\ o.$
$\|\ construct\ o\ [e0, e1, e2 \ldots]\ =\ de\ expressie\ ((e0\ \underline{o}\ e1)\ \underline{o}\ e2) \ldots$

De definitie van construct kan gemakkelijk in één regel (met een *fold*.)

22. Definieer *hervormAC*, *hervormA* en *hervormC* met behulp van *destruct* en *construct* uit de vorige opgave. Probeer zo veel mogelijk overeenkomst te krijgen in de drie definities. Let er op dat onderdelen van het argument van deze functies op hun beurt nog in standaardvorm gebracht moeten worden.

§**7 Tenslotte.** Er zijn nu nog twee functies niet gedefinieerd; *bags* en *chain*. Hun specificaties luiden als volgt:

$\|\ bags\ n\ xs\ =\ de\ verzameling\ van\ alle\ bags\ ter\ grootte\ n\ over\ verzameling\ xs$
$\|\ chain\ xs\ [ys0, ys1, \ldots, ysn]\ =\ ys0\ +\!\!+\ xs\ +\!\!+\ ys1\ +\!\!+\ xs\ +\!\!+\ \ldots\ +\!\!+\ ysn$

Opgaven

23. Definieer de functie *bags*.
24. Definieer de functie *chain*.

§**8 Nog meer tierlantijne.** Er zijn nog diverse mogelijkheden tot efficiëntieverbetering:

Opgaven

25. In de definitie van *flippos* en *flipposSv* staat de test $\#opln = k$. Geef een efficiënte test voor $\#opln = 0$, voor $\#opln > 0$, voor $\#opln = 1$.
26. Stel dat je niet in de flippos zelf geïnteresseerd bent, maar wel in álle verschillende expressies met uitkomst 24. Dat kan met de volgende definitie:

```
alle24expressies
  = "Alle 24expressies : \n\n" ++ layn (map showEx (filter correct zs))
    where
    zs = (concat . map (gen flippoOperaties) . permbags 4) [1 .. 9]
    correct e = eval e = (True, 24)
```

waarbij *permbags* de functie is met:

```
|| permbags n xs = de verz van permutaties van bags ter grootte n over xs
permbags n xs = (concat . map (mkset . perms) . bags n) xs
```

Geef een efficiënte definitie van *permbags*, waarbij het genereren van de tussentijdse lijsten en het gebruik van *mkset* achterwege blijft. (Pas op: zelfs met de efficiënte *permbags* duurt het nog ongeveer 3.5 uur cpu-tijd, namelijk 405879852 reducties, om de uitkomst volledig te tonen.)

A Het script

```
|| flippo : − voor het genereren van flippos en oplossingen
|| Maarten Fokkinga, maart 1996.

|| Een 'flippo' is een bag ter grootte 4 over de getallen 1 . . 9;
|| de opgave is met deze getallen een expressie te vormen met uitkomst 24,
|| waarbij ieder getal precies een keer gebruikt wordt.
||
|| Verzamelingen worden gerepresenteerd als lijsten zonder duplicaten.
|| Bags worden gerepresenteerd als lijsten met zonodig duplicaten.

|| ------------------------------------------------------------------------
|| Script parameters en constanten :

flippoLengte = 4
flippoGetallen = [1 . . 9]
flippoUitkomst = 24
flippoOperaties = [maal, deel, plus, minus]

flippoOpln xs = oplossingen flippoUitkomst flippoOperaties xs
flippoVerz = bags flippoLengte flippoGetallen || = de verz van alle flippos

opln xs
 = "Alle oplossingen voor flippo " ⧺ show xs ⧺ " : \n" ⧺
   (layn . map showEx . flippoOpln) xs
flippos k
 = "Alle flippos met precies " ⧺ show k ⧺ " oplossingen : \n\n" ⧺
   layn
   [show xs ⧺ " : " ⧺ concat (map showEx opln) |
   xs ← flippoVerz; opln ← [flippoOpln xs]; #opln = k]
|| oplnSv xs = de verz van alle sv oplossingen voor flippo xs
|| flipposSv k = de verz van alle flippos met precies k sv oplossingen

|| ------------------------------------------------------------------------
|| Datastructuren : operator, expressie.

|| Bij elke datastructuur data definieren we (voor testdoeleinden) de functies
||   showData :: data → [char]
||   readData :: [char] → data
|| met hulpfunctie
||   readData' :: [char] → (data, [char])
||
|| Voorbeeld :
|| showEx e = "(1 + (2×3))"
|| readEx "(1 + (2×3))" = e
```

```
|| readEx' "(1 + (2×3))blahblahblah" = (e, "blahblahblah")

|| .................... operator .......................................
operator ::= Op num
maal = Op 0
deel = Op 1
plus = Op 2
minus = Op 3
|| exp, log, div, mod, ...

showOp (Op o) = ["×", "/", " + ", " − "] ! o
readOp' (x : xs) = (hd [o | o←[maal, deel, plus, minus]; showOp o = [x]], xs)

|| .................... expressie .......................................
expressie ::= NumEx num | OpEx expressie operator expressie
behaakt x = "(" ++ x ++ ")"
onhaakt x = " " ++ x ++ " "

|| showE x  toont x volledig behaakt :

|| showEx x  toont x met weinig (ihb geen omvattende) haakjes :
showEx (NumEx x) = shownum x
showEx (OpEx x o y) = showExL o x ++ showOp o ++ showExR o y

|| showExL o x toont x (als een Linker operand van o, zonodig met haakjes) :
showExL o' (NumEx x) = shownum x
showExL o' (OpEx x o y)
 = onhaakt (showEx (OpEx x o y)), if ?????
 = behaakt (showEx (OpEx x o y)), otherwise

|| showExR o x toont x (als een Rechter operand van o, zonodig met haakjes) :

|| readEx vereist een volledig behaakte notatie als invoer :
readEx xs = e    where (e, zs) = readEx' xs
readEx' (x : xs)
 = (NumEx ?????, xs), if digit x
 = (OpEx e o e', zs), if x = '('
     where ?????

|| --------------------------------------------------------------------------
|| Het genereren van oplossingen

|| oplossingen out ops xs = de verzameling van alle expressies met
|| operanden xs (in WILLEKEURIGE volgorde; xs fungeert als bag), haakjes
|| willekeurig, operaties uit ops en uitkomst out :
oplossingen out ops xs
 = (filter correct . genereer ops) xs
```

$where$
$correct\ ys\ =\ eval\ ys\ =\ (True,\ out)$

$\|\ eval\ e\ =\ (bool:\ e\ heeft\ een\ waarde,\ num:\ de\ waarde\ van\ e)$
$\|\ Bij\ deling\ door\ nul\ heeft\ een\ expressie\ geen\ waarde!$
$eval\ (NumEx\ x)\ =\ (True,\ x)$
$eval\ (OpEx\ x\ o\ y)\ = ???$

$\|\ gen\ ops\ xs\ =\ de\ verzameling\ van\ alle\ expressies\ met\ operanden\ xs$
$\|\ (in\ deze\ VASTE\ volgorde;\ xs\ fungeert\ als\ lijst),\ haakjes\ willekeurig,$
$\|\ en\ operaties\ uit\ ops:$
$gen\ ops\ [x]\ =\ [NumEx\ x]$
$gen\ ops\ xs\ =\ [OpEx\ e\ o\ e'\ |\ \ldots e{\leftarrow}\ldots e'{\leftarrow}\ldots o{\leftarrow}\ldots]$

$\|\ genereer\ ops\ xs\ =\ de\ verzameling\ van\ alle\ expressies\ met$
$\|\ operanden\ xs\ (in\ WILLEKEURIGE\ volgorde;\ xs\ fungeert\ als\ bag),\ haakjes$
$\|\ willekeurig,\ operaties\ uit\ ops:$
$genereer\ ops\ xs\ = ?????$

$\|$ ----
$\|\ Standaardvorm$

$sv\ (OpEx\ e\ o\ e')$
$=\ hervormAC\ o\ (OpEx\ e\ o\ e'),\ if\ assoc\ o\ \wedge\ comm\ o$
$=\ hervormA\ \ o\ (OpEx\ e\ o\ e'),\ if\ assoc\ o\ \wedge\ \neg comm\ o\qquad \|\ overbodig\ bij\ flippos$
$=\ hervormC\ \ o\ (OpEx\ e\ o\ e'),\ if\ \neg assoc\ o\ \wedge\ comm\ o\qquad \|\ overbodig\ bij\ flippos$
$sv\ (OpEx\ e\ o\ e')\ =\ OpEx\ (sv\ e)\ o\ (sv\ e')$
$sv\ (NumEx\ n)\ =\ (NumEx\ n)$

$\|\ hervormA\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ e's\ operator\ o\ accoc\ is)$
$\|\ hervormC\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ e's\ operator\ o\ comm\ is)$
$\|\ hervormAC\ o\ e\ =\ de\ standaardvorm\ van\ e\ (gegeven\ dat\ o\ accoc\ en\ comm\ is)$

$\|\ destruct\ o\ e\ =\ een\ bag\ [e0, e1, \ldots]\ van\ expressies\ zo\ dat:$
$\|\qquad e\ =\ e0\ \underline{o}\ e1\ \underline{o}\ \ldots\ (met\ haakjes\ op\ een\ of\ andere\ manier\ geplaatst),\ en$
$\|\qquad geen\ van\ e0,\ e1, \ldots\ is\ een\ o\ expressie.$
$\|\ construct\ o\ [e0, e1, e2\ldots]\ =\ de\ expressie\ ((e0\ \underline{o}\ e1)\ \underline{o}\ e2)\ldots$

$\|$ ----
$\|\ Miscellania$

$\|\ bags\ n\ xs\ =\ de\ verzameling\ van\ alle\ bags\ ter\ grootte\ n\ over\ verzameling\ xs:$
$\|\ chain\ xs\ [ys0, ys1, \ldots, ysn]\ =\ ys0\ +\!\!+\ xs\ +\!\!+\ ys1\ +\!\!+\ xs\ +\!\!+\ \ldots\ +\!\!+\ ysn$

B Antwoorden

1. $\#[(o1, o2, o3) \mid o1, o2, o3 \leftarrow [\times, /, +, -];\ ((3\ \underline{o1}\ 3)\ \underline{o2}\ 7)\ \underline{o3}\ 7\ =\ 24]$.

2. Bijvoorbeeld, definieer eerst hoe een drietal operaties getoond moet worden:

$$
\begin{array}{l}
f\ (o1, o2, o3) \\
=\ \texttt{"("}\ +\!\!+\ showO\ o1\ +\!\!+\ \texttt{", "}\ +\!\!+\ showO\ o2\ +\!\!+\ \texttt{", "}\ +\!\!+\ showO\ o3\ +\!\!+\ \texttt{")"} \\
showO\ o \\
=\ \texttt{"}\times\texttt{"},\ \mathit{if}\ 6\ \underline{o}\ 2\ =\ 12 \\
=\ \texttt{"/"},\ \mathit{if}\ 6\ \underline{o}\ 2\ =\ 3 \\
\ldots
\end{array}
$$

De uitdrukking is dan:
$[f\ (o1, o2, o3) \mid o1, o2, o3 \leftarrow [\times, /, +, -];\ ((3\ \underline{o1}\ 3)\ \underline{o2}\ 7)\ \underline{o3}\ 7\ =\ 24]$.

Een andere mogelijkheid is operaties te representeren met getallen (het tonen daarvan levert geen problemen). De interpretatie van een getal als een operatie wordt vastgelegd door deze functie f:

$$
\begin{array}{l}
f\ x\ 1\ y\ =\ x\times y \\
f\ x\ 2\ y\ =\ x/y \\
\ldots
\end{array}
$$

Dan is de uitdrukking:
$[(o1, o2, o3) \mid o1, o2, o3 \leftarrow [1, 2, 3, 4];\ f\ (f\ (f\ 3\ o1\ 3)\ o2\ 7)\ o3\ 7\ =\ 24]$.

3. Laat $xs\ =\ [3, 3, 7, 7]$, dan is zo'n uitdrukking, bijvoorbeeld:

$$
\begin{array}{l}
\#[(o1, o2, o3) \mid o1, o2, o3 \leftarrow [\times, /, +, -]; \\
\quad or\ [((a\ \underline{o1}\ b)\ \underline{o2}\ c)\ \underline{o3}\ d\ =\ 24 \mid \\
\qquad a\leftarrow xs;\ b\leftarrow xs -\!\!-\ [a];\ c\leftarrow xs -\!\!-\ [a, b];\ d\leftarrow xs -\!\!-\ [a, b, c]]\ ]
\end{array}
$$

4. Laat '*hd*' weg om alle zeventallen te krijgen:

$$
\begin{array}{l}
hd\ [(a, o1, b, o2, c, o3, d) \mid \\
\quad o1, o2, o3 \leftarrow [\times, /, +, -]; \\
\quad a\leftarrow xs;\ b\leftarrow xs -\!\!-\ [a];\ c\leftarrow xs -\!\!-\ [a, b];\ d\leftarrow xs -\!\!-\ [a, b, c]; \\
\quad ((a\ \underline{o1}\ b)\ \underline{o2}\ c)\ \underline{o3}\ d\ =\ 24\ ]
\end{array}
$$

5. Op vijf manieren. Als $o1$ of $o3$ de hoofd-operator is, zijn er twee manieren om het restant (met twee operaties) te behaken. En als $o2$ de hoof-operator is, ligt alles vast.

6. Er zijn 9^4 keuzen voor de vier operanden, 4^3 keuzen voor de operaties, en 5 keuzen voor de haakjes-plaatsing. Die keuzen zijn onafhankelijk van elkaar, en geven tesamen dus $9^4\times4^3\times5\ =\ 2099520$ mogelijkheden.

7. Er zijn $\binom{9}{4} = \frac{9\cdot8\cdot7\cdot6}{4\cdot3\cdot2\cdot1}$ manieren om vier verschillende getallen uit $1\,..\,9$ te kiezen. Er zijn $\binom{9}{3}$ manieren om drie getallen te kiezen; en bij ieder van die mogelijkheden zijn er 3 manieren om één ervan te verdubbelen zodat je een viertal krijg. Er zijn $\binom{9}{2}$ manieren om twee getallen te kiezen, en steeds 3 manieren om uit zo'n tweetal een viertal te maken. Er zijn $\binom{9}{1}$ manieren om één getal te kiezen, en steeds 1 manier om uit zo'n eental een viertal te maken. Totaal: $1\times\binom{9}{4} + 3\times\binom{9}{3} + 3\times\binom{9}{2} + 1\times\binom{9}{1} = 495$ manieren om vier getallen (niet noodzakelijk verschillend) te kiezen uit $1\,..\,9$.

8. Wijzig het deel '*flippoOpln*' in de definitie van *opln* door '*mkset . map sv . flippoOpln*'; dan worden alleen oplossingen in standaardvorm gegenereerd, en zonder duplicaten:

$$\begin{aligned}&\mathit{oplnSv}\ \mathit{xs}\\ &= \texttt{"}\mathit{Alle\ sv\ oplossingen\ voor\ flippo}\ \texttt{"} +\!\!+\ \mathit{show}\ \mathit{xs}\ +\!\!+\ \texttt{" : \textbackslash n"} +\!\!+\\ &\quad (\mathit{layn}\ .\ \mathit{map}\ \mathit{showEx}\ .\ \mathit{mkset}\ .\ \mathit{map}\ \mathit{sv}\ .\ \mathit{flippoOpln})\ \mathit{xs}\end{aligned}$$

9. Precies als bij de vorige opgave:

$$\begin{aligned}&\mathit{flipposSv}\ k\\ &= \texttt{"}\mathit{Alle\ flippos\ met\ precies}\ \texttt{"} +\!\!+ \mathit{show}\ k +\!\!+ \texttt{"}\ \mathit{sv\ oplossingen} : \texttt{\textbackslash n\textbackslash n"} +\!\!+\\ &\quad \mathit{layn}\\ &\quad [\mathit{show}\ \mathit{xs}\ +\!\!+\ \texttt{" : "}\ +\!\!+\ \mathit{chain}\ (\texttt{"\textbackslash n}\qquad\qquad\qquad\texttt{"})\ (\mathit{map}\ \mathit{showEx}\ \mathit{opln})\ |\\ &\quad \mathit{xs} \leftarrow \mathit{flippoVerz};\ \mathit{opln} \leftarrow [\mathit{mkset}\ (\mathit{map}\ \mathit{sv}\ (\mathit{flippoOpln}\ \mathit{xs}))];\ \#\mathit{opln}\ =\ k]\end{aligned}$$

10. 1. *flippos* 0
    2. Voeg aan de definitie van *flippos* toe (in de lijstcomprehensie): $4\ =\ \#\mathit{mkset}\ \mathit{xs}$.
    3. Voeg nu toe: $\mathit{sort}\ \mathit{xs}\ =\ \mathit{take}\ 4\ [\mathit{xs}!0\,..\,]$.
    4. Dat lukt niet of nauwelijks; daarvoor moeten we *oplossingen* aanpassen.
    5. Niet mooi, maar 't lukt wel; herdefinieer *flippoOpln* door:
    $\mathit{oplossingen}\ 24\ [\mathit{maal}]\ +\!\!+\ \ldots\ +\!\!+\ \mathit{oplossingen}\ 24\ [\mathit{minus}]$.

11. Er zijn vele varianten mogelijk; hier is er een:

$$\begin{aligned}&\mathit{evalOp}\ (\mathit{Op}\ o)\ =\ [\times,\ /,\ +,\ -]\ !\ o\\ &\mathit{prio}\ (\mathit{Op}\ o)\ =\ [1,2,3,3]!o\\ &\mathit{assoc}\ o\ =\ \mathit{member}\ [\mathit{maal},\ \mathit{plus}]\ o\\ &\mathit{comm}\ o\ =\ \mathit{member}\ [\mathit{maal},\ \mathit{plus}]\ o\end{aligned}$$

12.

$$\begin{aligned}&\|\ \mathit{showE}\ x\ \ \mathit{toont}\ x\ \mathit{volledig\ behaakt} :\\ &\mathit{showE}\ (\mathit{NumEx}\ x)\ =\ \mathit{shownum}\ x\\ &\mathit{showE}\ (\mathit{OpEx}\ x\ o\ y)\ =\ \mathit{behaakt}\ (\mathit{showE}\ o\ x\ +\!\!+\ \mathit{showOp}\ o\ +\!\!+\ \mathit{showE}\ o\ y)\end{aligned}$$

13. De voorwaarde luidt: $\mathit{prio}\ o\ \leq\ \mathit{prio}\ o'$.
Deel $<$ van deze voorwaarde is correct volgens regel (a) uit de begeleidende tekst; en

deel $=$ volgens regel (c).

14. Een 'onhaakte' expressie zou fout kunnen zijn en moet daarom goed bewaakt worden; een behaakte expressie is sowieso correct.

$$
\begin{array}{l}
\| \ showExR\ o\ x\ toont\ x\ (als\ een\ Rechter\ operand\ van\ o,\ zo\ nodig\ met\ haakjes) :\\
showExR\ o'\ (NumEx\ x) \ = \ shownum\ x\\
showExR\ o'\ (OpEx\ x\ o\ y)\\
\ = \ onhaakt\ (showEx\ (OpEx\ x\ o\ y)),\ if\ voorwaarde\\
\ = \ behaakt\ (showEx\ (OpEx\ x\ o\ y)),\ otherwise\\
\quad where\ voorwaarde\\
\quad\ = \ prio\ o' \ > \ prio\ o \vee\\
\qquad o' = o \ \wedge\ assoc\ o \vee\\
\qquad o' = plus \ \wedge\ o = minus \vee\\
\qquad o' = maal \ \wedge\ o = deel
\end{array}
$$

De *voorwaarde* mag eventueel sterker gemaakt worden (zodat hij in minder gevallen vervuld zal zijn). De eerste twee disjuncten zijn correct wegens regel (a) en regel (b) uit de begeleidende tekst. De laatste twee disjuncten zijn correct volgens de rekenregels $x + (y - z) = (x + y) - z$ en $x{\times}(y/z) = (x{\times}y)/z$.

15.

$$
\begin{array}{l}
readEx'\ (x : xs)\\
\ = \ (NumEx\ (numval\ [x]),\ xs),\ if\ digit\ x\\
\ = \ (OpEx\ e\ o\ e',\ zs),\ if\ x \ = \ '('\\
\quad where\\
\quad (e,\ rest1) \ = \ readEx'\ xs\\
\quad (o,\ rest2) \ = \ readOp'\ rest1\\
\quad (e', rest3) \ = \ readEx'\ rest2\\
\quad ')' : zs \ = \ rest3
\end{array}
$$

16. Voeg aan het begin van de definities een clausule toe die spaties overslaat:

$$
\begin{array}{l}
readOp'\ ('\ ' : xs) \ = \ readOp'\ xs\\
readEx'\ ('\ ' : xs) \ = \ readEx'\ xs
\end{array}
$$

17. Bewaak de gegeven clausule voor *readEx xs* met de voorwaarde dat het restant *zs* louter uit spaties bestaat:

$$
readEx\ xs \ = \ e,\ if\ dropwhile\ (='\ ')\ zs \ = \ \texttt{""} \qquad where\ (e,\ zs) \ = \ readEx'\ xs
$$

18. De definitie volgt een standaard techniek (vergelijk met die van *readEx'*):

$$
\begin{array}{l}
eval\ (NumEx\ x) \ = \ (True,\ x)\\
eval\ (OpEx\ x\ o\ y)
\end{array}
$$

$$= (xOK \;\wedge\; yOK \;\wedge\; (yVal \neq 0 \;\vee\; o \neq deel),\;\; evalOp\ o\ xVal\ yVal)$$
$$\begin{aligned}&where\\&(xOK,\ xVal) \;=\; eval\ x\\&(yOK,\ yVal) \;=\; eval\ y\end{aligned}$$

Merk op dat als het eerste lid van de uitkomst *False* is, de waarde van de expressie niet bestaat, en ook niet gebruikt mag worden.

19. We splitsen xs in twee stukken, waarvan het eerste deel i lang is. Dus er moet gelden $1 \leq i \leq \#xs - 1$.

$$\begin{aligned}&gen\ ops\ [x] \;=\; [NumEx\ x]\\&gen\ ops\ xs\\&= [OpEx\ e\ o\ e' \mid\\&\quad i \leftarrow [1\,..\,\#xs-1];\ e \leftarrow gen\ ops\ (take\ i\ xs);\ e' \leftarrow gen\ ops\ (drop\ i\ xs);\\&\quad o \leftarrow ops]\end{aligned}$$

Merk op dat als xs niet-leeg is (en dus tenminste twee getallen bevat), ook *take* i xs en *drop* i xs beide niet-leeg zijn.

20.

$$\begin{aligned}&genereer\ ops \;=\; concat\ .\ map\ (gen\ ops)\ .\ perms\\&\|\\&\|\ perms\ xs \;=\; de\ verz\ van\ alle\ permutaties\ van\ bag\ xs\\&\|\ perms'\ xs \;=\; de\ bag\ van\ alle\ permutaties\ van\ bag\ xs:\\&perms \;=\; mkset\ .\ perms'\\&perms'\ [] \;=\; [[]]\\&perms'\ (x:xs)\\&= [take\ i\ ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} drop\ i\ ys \mid\\&\quad ys \leftarrow perms'\ xs;\ i \leftarrow [0\,..\,\#xs]]\end{aligned}$$

Er zijn vele varianten van de definitie voor $perms'$ mogelijk. In de definitie hierboven doorloopt i de rangnummers van ys waarna x geplaatst wordt. Het is niet nodig om x nog te plaatsen in ys ná een positie j waarop al een x staat; want zo'n permutatie wordt ook al opgeleverd door x te plaatsen op j in een $ys' \leftarrow perms'\ xs$ met $ys!i \;=\; x$. Aldus kunnen we het gebruik van *mkset* vermijden, en verkrijgen we een veel efficiënter algoritme:

$$\begin{aligned}&perms\ [] \;=\; [[]]\\&perms\ (x:xs)\\&= [take\ i\ ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} drop\ i\ ys \mid\\&\quad ys \leftarrow perms\ xs;\ i \leftarrow [0\,..\,\#\ takewhile\ (\neq x)\ ys]]\end{aligned}$$

Merk op hoe weinig verschil er is tussen deze definitie en die van $perms'$; het verschil in tekst is heel klein, maar in semantiek heel groot.

21.

$$\begin{array}{l} destruct\ o\ (OpEx\ e\ o\ e') \ = \ destruct\ o\ e\ +\!\!+\ destruct\ o\ e' \\ destruct\ o\ e \ = \ [e] \\ construct\ o\ es \ = \ foldl1\ oper\ es \quad where\ oper\ e\ e' \ = \ OpEx\ e\ o\ e' \end{array}$$

22.

$$\begin{array}{l} hervormAC\ o \ = \ construct\ o\ .\ map\ nf\ .\ sort\ .\ destruct\ o \\ hervormA\ o \ = \ construct\ o\ .\ map\ nf\ .\ destruct\ o \\ hervormC\ o\ (OpEx\ e\ o\ e') \ = \ (construct\ o\ .\ map\ nf\ .\ sort)\ [e, e']; \end{array}$$

23.

$$\begin{array}{l} \|\ bags\ n\ xs \ = \ de\ verzameling\ van\ alle\ bags\ ter\ grootte\ n\ over\ verzameling\ xs : \\ bags\ 0\ xs \ = \ [[]] \\ bags\ (n+1)\ []\ = \ [] \\ bags\ (n+1)\ (x:xs) \\ \ = \ [x:ys \mid ys \leftarrow bags\ n\ (x:xs)]\ +\!\!+\ [ys \mid ys \leftarrow bags\ (n+1)\ xs] \end{array}$$

24.

$$\begin{array}{l} \|\ chain\ xs\ [ys0, ys1, \ldots, ysn] \ = \ ys0\ +\!\!+\ xs\ +\!\!+\ ys1\ +\!\!+\ xs\ +\!\!+\ \ldots\ +\!\!+\ ysn \\ chain\ xs\ yss \ = \ drop\ (\#xs)\ (concat\ (map\ (xs+\!\!+)\ yss)) \end{array}$$

25. 1. $\#opln = 0$ is gelijk aan $opln = []$.
2. $\#opln > 0$ is gelijk aan $opln \neq []$.
3. $\#opln = 1$ is gelijk aan $take\ 1\ opln\ \neq\ []\ \wedge\ drop\ 1\ opln\ =\ []$.
Een efficiënte test met dezelfde uitkomst als $\#opln = k$ is:
$\#take\ k\ opln\ =\ k\ \wedge\ drop\ k\ opln\ =\ []$.

26. Deze definitie combineert de definities van *bags* en *perms* uit de antwoorden bij opgaven 23 en 20:

$$\begin{array}{l} permbags\ 0\ xs \ = \ [[]] \\ permbags\ (n+1)\ []\ = \ [] \\ permbags\ (n+1)\ (x:xs) \\ \ = \ permbags\ (n+1)\ xs\ +\!\!+ \\ \quad [take\ i\ ys\ +\!\!+\ [x]\ +\!\!+\ drop\ i\ ys \mid \\ \quad\ s \leftarrow permbags\ n\ (x:xs);\ i \leftarrow [0\ ..\ \#takewhile\ (\neq x)\ ys]] \end{array}$$

Dit is efficiënter dan de definitie uit de opgave, omdat het genereren van de tussentijdse lijsten, met overbodige elementen die er door *mkset* uitgefilterd werden, nu achterwege blijft.