Inhoudsopgave

FP

Geen Woorden Maar Daden

(versie: mei 1997)

Deze bundel opgaven is bedoeld als een **eerste kennismaking** met programmeren, aan de hand van een functionele programmertaal. De student krijgt 'gevoel' voor het nauwkeurig uitdrukken van gewenste resultaten, en leert de **basistechnieken** op **informele** wijze beheersen. Het gaat er in deze bundel niet om technische termen, theorie of technieken voor gevorderden aan te leren.

In een vervolgcursus kunnen pas technieken voor gevorderden, ingewikkelde algoritmen en formele aspecten aan bod komen.

Naar mijn overtuiging is er maar één manier om een vaardigheid zoals programmeren aan te leren: véél afkijken, véél nadoen en vooral véél oefenen. Dat is effectiever dan het lezen van lange uiteenzettingen en wijze lessen. Vanuit die overtuiging heb ik deze cursus opgezet. Het bekijken van antwoorden wordt aangemoedigd; er blijven voldoende veel opgaven over om zelf te proberen.

Vanuit de visie dat "even uitproberen" effectiever is dan precieze omschrijvingen, ben ik bewust onvolledig geweest in de syntax en semantiek van diverse taalconstructies.

De tekst die nu voor u ligt is de tweede versie. Ik houd me aanbevolen voor alle commentaar, suggesties en kritiek (zowel positieve als ook negatieve).

Maarten Fokkinga, mei 1996
e-mail: `fokkinga@cs.utwente.nl`

In de druk van 1997 zijn slechts heel kleine correcties aangebracht.

1 De Miranda omgeving

De programmeertaal Miranda[1] wordt in deze cursus gehanteerd. Er is een on-line, menu-gestuurde, handleiding, waar alles over de taal beknopt en helder uitgelegd wordt. Dit hoofdstuk dient slechts als naslagwerk. We raden je aan het zo nu en dan eens door te kijken.

§1.1 Intikken van Miranda teksten. Sommige tekens, zoals $\times$ en $+\!\!+$ zitten niet op het toetsenbord. Dergelijke tekens moeten als volgt in Miranda ingetikt worden:

<table>
<tr><td>$\times$</td><td>$+\!\!+$</td><td>$-\!\!-$</td><td>$\equiv$</td><td>$\leq$</td><td>$\geq$</td><td>$\neq$</td><td>$\neg$</td><td>$\rightarrow$</td><td>$\leftarrow$</td><td>$\wedge$</td><td>$\vee$</td><td>$\|$</td></tr>
<tr><td><code>*</code></td><td><code>++</code></td><td><code>--</code></td><td><code>==</code></td><td><code><=</code></td><td><code>>=</code></td><td><code>~=</code></td><td><code>~</code></td><td><code>-></code></td><td><code><-</code></td><td><code>&</code></td><td><code>\/</code></td><td><code>||</code></td></tr>
</table>

<table>
<tr><td>α</td><td>β</td><td>γ</td><td>δ</td><td>ε</td><td>ζ</td><td>$\underline{abc}$</td></tr>
<tr><td><code>*</code></td><td><code>**</code></td><td><code>***</code></td><td><code>****</code></td><td><code>*****</code></td><td><code>******</code></td><td><code>$abc</code></td></tr>
</table>

Als je een lange uitdrukking wil uittesten, *definieer* hem dan in het script met een korte naam, bijvoorbeeld *tst*. Dan hoef je bij (denk- of tik-)fouten niet steeds de hele uitdrukking opnieuw in te tikken, maar kun je hem met de tekstverwerker gemakkelijk wijzigen.

§1.2 Commando's. De interpretator kent onder andere de volgende commando's en afkortingen; met name $$ is erg handig!

$$	afkorting voor de laatst geëvalueerde expressie; als expr te gebruiken
expressie	toon de uitkomst van *expressie* op het scherm
expressie &> *file*	zet de uitkomst van *expressie* in *file*
expressie ::	toon het type van *expressie*
?	toon alle gedefinieerde identifiers
?*identifier*	toon informatie over *identifier*
??*identifier*	activeer de editor op de definitie van *identifier*, ook bij standaard fcts
/*m*, /*man*	start de menu-gestuurde handleiding (manual)
/*h*, /*help*	toon de mogelijke commando's

§1.3 Standaard functies en waarden. De volgende omschrijvingen dienen als geheugensteuntje; ze zijn niet altijd 100% volledig. Zie de Miranda handleiding voor meer informatie. Een alfabetisch gerangschikte lijst van alle bekende identifiers wordt getoond door het commando ? na de prompt. We gebruiken 'elt' als afkorting van 'element'; *xs* en *ys* duiden steeds lijsten aan.

$$
\begin{aligned}
&\mathit{fst}\ (x, y) &&= x \\
&\mathit{snd}\ (x, y) &&= y \\[1em]
&x\ :\ [y, \ldots, z] &&= [x, y, \ldots, z]
\end{aligned}
$$

[1] Miranda$^{\text{TM}}$ is een handelsmerk van Research Software Limited

$\mathit{postfix}\ z\ [x, \ldots, y]$	$=$	$[x, \ldots, y, z]$
$[u, v, \ldots, w] +\!\!+ [x, y, \ldots, z]$	$=$	$[u, v, \ldots, w, x, y, \ldots, z]$
$\mathit{concat}\ [\mathit{xs}, \mathit{ys}, \ldots, \mathit{zs}]$	$=$	$\mathit{xs}\ +\!\!+\ \mathit{ys}\ +\!\!+\ \ldots\ +\!\!+\ \mathit{zs}$
$\mathit{hd}\ [x, y, \ldots, z]$	$=$	x
$\mathit{tl}\ [x, y, \ldots, z]$	$=$	$[y, \ldots, z]$
$\mathit{last}\ [x, \ldots, y, z]$	$=$	z
$\mathit{init}\ [x, \ldots, y, z]$	$=$	$[x, \ldots, y]$
$\mathit{index}\ \mathit{xs}$	$=$	de rangnummers van de elten in xs: $[0\ ..\ \#\mathit{xs}{-}1]$
$\mathit{xs}\ !\ n$	$=$	het element van xs met rangnummer n
$\mathit{take}\ n\ \mathit{xs}$	$=$	het beginstuk van xs ter lengte $\mathit{min}[n, \#\mathit{xs}]$
$\mathit{drop}\ n\ \mathit{xs}$	$=$	zodanig dat: $\mathit{take}\ n\ \mathit{xs}\ +\!\!+\ \mathit{drop}\ n\ \mathit{xs}\ =\ \mathit{xs}$
$\mathit{takewhile}\ p\ \mathit{xs}$	$=$	langste beginstuk van xs waarvan elk elt aan p voldoet
$\mathit{dropwhile}\ p\ \mathit{xs}$	$=$	zodanig dat: $\mathit{takewhile}\ p\ \mathit{xs}\ +\!\!+\ \mathit{dropwhile}\ p\ \mathit{xs}\ =\ \mathit{xs}$
$\mathit{filter}\ p\ \mathit{xs}$	$=$	de grootste sublijst van xs waarvan elk elt aan p voldoet
$\mathit{mkset}\ \mathit{xs}$	$=$	een grootste sublijst van xs zonder duplicaten
$\mathit{xs}\ {-}{-}\ \mathit{ys}$	$=$	de lijst ontstaan uit xs door schrapping *vlnr* van ys
		vb: $[1,2,3,2,1]\ {-}{-}\ [4,2,1,2]\ =\ [3,1]$
$\mathit{map}\ f\ [x, y, \ldots, z]$	$=$	$[f\ x,\ f\ y,\ \ldots\ ,\ f\ z]$
$\mathit{reverse}\ [x, y, \ldots, z]$	$=$	de argumentlijst omgedraaid: $[z, \ldots, y, x]$
$\mathit{zip}\ \ ([u, v, \ldots],\ [x, y, \ldots])$	$=$	$[(u, x),\ (v, y),\ \ldots]$
$\mathit{zip}2\ \ [u, v, \ldots]\ \ [x, y, \ldots]$	$=$	$[(u, x),\ (v, y),\ \ldots]$
$\mathit{map}2\ f\ [u, v, \ldots]\ [x, y, \ldots]$	$=$	$[f\ u\ x,\ f\ v\ y,\ \ldots\ ]$, dus: $\mathit{zipwith}\ f\ (\mathit{xs}, \mathit{ys})\ =\ \mathit{map}2\ f\ \mathit{xs}\ \mathit{ys}$
$\mathit{zip}3,\ \mathit{zip}4,\ \mathit{zip}5,\ \mathit{zip}6$	$=$	net als $\mathit{zip}2$ maar met 3, 4, 5, 6 losse argumentlijsten
$\mathit{transpose}\ [[a, b, \ldots], [d, e, \ldots], [x, y, \ldots]]$	$=$	$[[a, d, x], [b, e, y], \ldots]$, etcetera
$\mathit{rep}\ n\ x$	$=$	$[x, x, \ldots, x]$ (n maal een x)
$\mathit{repeat}\ x$	$=$	$[x, x, \ldots]$ (oneindig lange lijst)
$\mathit{iterate}\ f\ x$	$=$	$[x, f^1 x, f^2 x, f^3 x,\ \ldots\ ]$ waarbij $f^i x = f(f(\ldots f\ x))$
$\mathit{until}\ p\ f\ x$	$=$	de eerste $f^i x$ $(i = 0, 1, ..)$ die aan p voldoet
$\mathit{limit}\ \mathit{xs}$	$=$	het eerste element van xs dat gelijk is aan z'n buur
$\#\mathit{xs}$	$=$	het aantal elementen in xs
$\mathit{member}\ \mathit{xs}\ x$	$=$	'lijst xs bevat x' (dus *True* of *False*)
$\mathit{sum}\ [x, y, \ldots, z]$	$=$	$x + y +\ \ldots\ + z$
$\mathit{product}\ [x, y, \ldots, z]$	$=$	$x \times y \times\ \ldots\ \times z$
$\mathit{and}\ [x, y, \ldots, z]$	$=$	$x\ \wedge\ y\ \wedge\ \ldots\ \wedge\ z$
$\mathit{or}\ [x, y, \ldots, z]$	$=$	$x\ \vee\ y\ \vee\ \ldots\ \vee\ z$

$foldl\ f\ a\ [x, y, \ldots, z]$	=	$((a\ \underline{f}\ x)\ \underline{f}\ y)\ \ldots\ \underline{f}\ z$
$foldl1\ f\ [x, y, \ldots, z]$	=	$(x\ \underline{f}\ y)\ \ldots\ \underline{f}\ z$
$foldr\ f\ a\ [x, \ldots, y, z]$	=	$x\ \underline{f}\ \ldots\ (y\ \underline{f}\ (z\ \underline{f}\ a))$
$foldr1\ f\ [x, \ldots, y, z]$	=	$x\ \underline{f}\ \ldots\ (y\ \underline{f}\ z)$
$scan\ f\ a\ [x, y, \ldots, z]$	=	$[a,\ (a\ \underline{f}\ x),\ ((a\ \underline{f}\ x)\ \underline{f}\ y),\ \ldots\ ,\ (((a\ \underline{f}\ x)\ \underline{f}\ y) \ldots \underline{f}\ z)]$
$x \leq y$	=	x is hoogstens y in een onbekende, maar vaste, ordening
$max\ xs$	=	het grootste element van xs
$min\ xs$	=	het kleinste element van xs
$max2\ x\ y$	=	de grootste van x en y
$min2\ x\ y$	=	de kleinste van x en y
$sort\ xs$	=	de permutatie $[x, y, \ldots, z]$ van xs met $x \leq y \leq \ldots \leq z$
$merge\ xs\ ys$	=	$sort\ (xs \mathbin{+\!\!+} ys)$ mits xs en ys gesorteerd zijn
$sqrt\ x$	=	$\sqrt{x}$
$abs\ x$	=	de absolute waarde van x
$entier\ x$	=	grootste gehele getal hoogstens x
$integer\ x$	=	‘getal x is geheel’ (dus *True* of *False*)
$subtract\ x\ y$	=	$y - x$, nut: $(-x)$ is géén sectie, maar $(subtract\ x)$ wel
$neg\ x$	=	$-x$, nut: $(-)$ is níet ‘negatief maken’; *neg* is dat wel
$arctan\ x$	=	de arctangens van x; $-pi/2 \leq$ uitkomst $\leq pi/2$
$sin\ x$	=	de sinus van x radialen
$cos\ x$	=	de cosinus van x radialen
pi	=	3.14159..., de verhouding omtrek/middellijn van een cirkel
e	=	2.71828..., het grondtal van de nat. logaritme
$log\ x$	=	de natuurlijke logaritme van x; grondtal e
$log10\ x$	=	de logaritme van x, met grondtal 10
$hugenum$	=	grootste floating point getal — systeem afh., bijv. 1e308
$tinynum$	=	kleinste positieve getal — systeem afhankelijk, bijv. 1e–324
$show\ x$	=	de Miranda-notatie van x (voor willekeurige x)
$shownum\ x$	=	de standaard notatie van getal x; $shownum\ pi$ = "3.14159265359"
$showfloat\ p\ x$	=	floating point notatie (p decimalen); $showfloat\ 3\ pi$ = "3.142"
$showscaled\ p\ x$	=	wetenschappelijke notatie; $showscaled\ 3\ pi$ = "3.142e+00"
$lay\ [xs, ys, \ldots, zs]$	=	$xs \mathbin{+\!\!+}$ "\n" $\mathbin{+\!\!+}\ ys \mathbin{+\!\!+}$ "\n" $\mathbin{+\!\!+} \ldots \mathbin{+\!\!+}\ zs \mathbin{+\!\!+}$ "\n"
$layn$	=	net als *lay*, maar nu met genummerde regels
$spaces\ n$	=	de lijst bestaande uit n spaties

ljustify n xs	=	string ter lengte $max[n, \#xs]$ met *xs* links, verder spaties
rjustify n xs	=	string ter lengte $max[n, \#xs]$ met *xs* rechts, verder spaties
cjustify n xs	=	string ter lengte $max[n, \#xs]$ met *xs* centraal, verder spaties
lines	=	de inverse van *lay*: splitst een string op de regelovergangen
numval xs	=	het getal met Miranda-notatie *xs* (evt. met spaties ervoor)
read xs	=	de inhoud (als één string) van de file met padnaam *xs*
readvals xs	=	de inhoud (als een $[\alpha]$) van de file met padnaam *xs*: elke regel van de file wordt één element van de lijst. Dus na `lay(map show` *vs*`) &>tmp` geldt *readvals* "*tmp*" $=$ *vs*
code x	=	de ASCII code (een getal) van karakter x
decode n	=	het karakter met ASCII code n
digit x	=	'karakter x is een cijfer $('0', \ldots, '9')$'
letter x	=	'karakter x is een letter $('A', \ldots, 'Z', 'a', \ldots, 'z')$'
id	=	de identiteitsfunctie: $id\ x\ =\ x$
const x	=	de "constante" functie f met $f\ y = x$
converse f	=	de functie f' met $f'\ x\ y = f\ y\ x$
undef	=	waardeloos; uitrekenen ervan resulteert in een foutmelding
error xs	=	waardeloos; uitrekenen ervan resulteert in foutmelding *xs*

Opgaven

- Onder welk menu-nummer staat in de on-line handleiding het gebruik ervan beschreven? Lees dat.
- Zoek in de on-line handleiding op of, in Miranda, vermenigvuldigen voorrang heeft op delen.
- Zoek in de on-line handleiding op of in Miranda de deel-operator links-associatief is, zodat $a/b/c = (a/b)/c$, of rechts-associatief, zodat $a/b/c = a/(b/c)$.
- Welk commando moet je aan de interpretator geven opdat na het uitrekenen van een expressie het aantal rekenstapjes ('reductiestappen') getoond wordt?

2 Functies en definities

Functies vormen een belangrijk bestanddeel van programma's. Met uitzondering van de haakjes om de argumenten, worden ze net zo genoteerd als in de wiskunde.

§2.1 Voorbeeld. De twee oplossingen van een vierkantsvergelijking $ax^2+bx+c = 0$ worden *theoretisch* gegeven door :

$$wortel_{1,2} \quad = \quad \frac{-b \pm \sqrt{b^2-4ac}}{2a} \quad .$$

In feite zijn $wortel_{1,2}$ afhankelijk van a, b, c; met andere woorden, ze zijn een functie van a, b, c. In de programmeertaal Miranda kunnen we ze als volgt definiëren (*sqrt* is de Miranda naam voor de square root $\sqrt{\ }$):

$$\begin{array}{lcl} wortel1\ a\ b\ c &=& (-b - sqrt\ (b\hat{}2 - 4{\times}a{\times}c))\ /\ (2{\times}a) \\ wortel2\ a\ b\ c &=& (-b + sqrt\ (b\hat{}2 - 4{\times}a{\times}c))\ /\ (2{\times}a) \end{array}$$

Als voorbeeld zijn hier twee *aanroepen* ('gebruik') van functie *wortel*1; de argumenten bij de ene aanroep zijn gelijk aan die bij de andere aanroep, dus ook de uitkomsten zijn gelijk:

$wortel1\ 3\ (-10)\ 8$
$wortel1\ (1{\times}3)\ (-2{\times}3-1{\times}4)\ (2{\times}4)$

Anders dan in wiskunde-boeken zijn er in Miranda geen haakjes nodig om aan te geven dat *wortel*1 een functie is en wat-daarna-komt zijn parameters of argumenten zijn.

> **Haakjes**
> **worden uitsluitend gebruikt om aan te geven wat-bij-wat hoort**.

In Miranda gelden voor de operatoren ˆ, ×, /, +, − de volgende voorrangregels: éérst machtsverheffen ˆ, dan vermenigvuldigen en delen met gelijke voorrang, en tenslotte optellen en aftrekken met gelijke voorrang. Daarom kunnen er in de definities van *wortel*1 en *wortel*2 hierboven geen haakjes worden weggelaten zonder dat de betekenis verandert. In Miranda geldt tevens:

> **een functie–argument binding**
> **heeft altijd voorrang boven een operator–operand binding**.

Dus:

$wortel1\ 3\ (-10)\ 2{\times}4$	staat voor	$(wortel1\ 3\ (-10)\ 2)\ \times\ 4$
	en **niet** voor	$wortel1\ 3\ (-10)\ (2{\times}4)$
$sqrt\ \ b\hat{}2\ \ldots$	staat voor	$(sqrt\ b)\hat{}2\ \ldots$
	en **niet** voor	$sqrt\ (b\hat{}2\ \ldots)$.

Overbodige haakjes zijn geoorloofd, en kunnen soms misverstanden voorkomen; teveel haakjes verslechteren de leesbaarheid. Spaties hebben geen invloed op de bepaling van wat-bij-wat hoort.

§2.2 Variaties. Andere schrijfwijzen voor de definitie van *wortel*1, met dezelfde betekenis, zijn:

$$wortel1\ p\ q\ r\ =\ (-q - sqrt\ (q\hat{\ }2 - 4{\times}p{\times}r))\ /\ (2{\times}p)$$

$$\begin{aligned}&wortel1\ a\ b\ c\\ &=\ (-\ b + sqrt\ discr)\ /\ (2{\times}a)\\ &\qquad where\\ &\qquad discr\ =\ b\hat{\ }2 - 4{\times}a{\times}c\end{aligned}$$

$$\begin{aligned}&wortel1\ a\ b\ c\ =\ (-\ b + sqrt\ (discr\ a\ b\ c))\ /\ (2{\times}a)\\ &discr\ a\ b\ c\ =\ b\hat{\ }2 - 4{\times}a{\times}c\end{aligned}$$

In het eerste alternatief hebben de parameters van *wortel*1 een andere naam gekregen. De naamgeving van parameters in een functiedefinitie geldt alleen binnen de definitie, en is in gebruik van de functie niet te achterhalen. In het tweede alternatief wordt een 'locale' hulpdefinitie gegeven. De parameters van een functiedefinitie zijn overal in het rechterlid van de definitie bekend, dus ook binnen de 'locale' where-definities. Daarom moet, bij het derde alternatief, de globaal gedefinieerde *discr* die parameters meegegeven krijgen, terwijl de locaal gedefinieerde *disrc* die parameters a, b, c gewoon kent. Merk op dat we de laatste regel ook kunnen schrijven als: $discr\ x\ y\ z\ =\ y\hat{\ }2 - 4{\times}x{\times}z$; immers, de naamgeving van de parameters is alleen geldig binnen de definitie zelf.

Locale definities zijn vooral handig om een meermalen voorkomend onderdeel slechts eenmaal op te hoeven schrijven. Ook helpen locale definities (soms!) om de structuur van een uitdrukking beter aan te geven: in de laatste twee alternatieven wordt de rol van de discriminant $b^2{-}4ac$ benadrukt. In een where-part mogen verscheidene definities staan, ook van functies.

Opgaven

Als we zeggen 'Definieer' dan bedoelen we 'Definieer in Miranda'.

1. Definieer de functie *gemiddelde* die het gemiddelde van zijn twee argumenten oplevert. Wat is de uitkomst van *gemiddelde* 3 7, en van *gemiddelde* 3 2+5, en van *gemiddelde* (3, 7)?
2. Een geldbedrag staat weg tegen r procent rente per jaar; $0 \le r \le 100$. Definieer de functie *saldo* met:

$$\begin{aligned}saldo\ r\ n\ b\ \ =\ \ &\text{het eindbedrag na } n \text{ volle jaren}\\ &\text{bij rentepercentage } r \text{ en beginbedrag } b\end{aligned}$$

3. Definieer de functie *hypotenusa* die van een rechthoekige driehoek de lengte van de schuine zijde oplevert, wanneer de lengtes van de twee rechthoekszijden als argument gegeven worden.
4. Definieer de functie die het oppervlak van een blok (met hoogte h, diepte d, breedte b) oplevert. Geef de definitie de volgende vorm (je hoeft alleen het 'where-part' in te vullen):

$$\begin{aligned}&blokOpp\ h\ d\ b\\ &=\ 2{\times}voorkant + 2{\times}zijkant + 2{\times}onderkant\\ &\qquad where\end{aligned}$$

...

5. Definieer een functie *cirkelOpp* die, gegeven een straal r, de oppervlakte oplevert van een cirkel met straal r. De constante *pi* is al gedefinieerd (probeer maar!). Gebruik de functie een paar keer, ook met een samengesteld argument zoals 3+4 of 8/2.
6. Net als in de vorige opgave, maar nu voor de inhoud (het volume) van een bol.
7. De trigonometrische functies *arctan*, *cos*, *sin* zijn standaard functies in Miranda, dat wil zeggen: ze bestaan al, ze zijn al gedefinieerd. Probeer maar (geef de hoeken in radialen mee, en niet in graden). Definieer zelf een functie *tan* die de tangens oplevert; het argument is gegeven in radialen, dus *tan* (*pi*/4) levert 1.
8. Rijen zoals 1, 34, 67, ..., 1717 heten *rekenkundige rijen*: zij hebben de vorm a, $a+v$, $a+2v$, ..., $a+(n-1)v$, waarbij a de startwaarde is, v het verschil tussen opeenvolgende termen, en n het aantal termen. Van zo'n rekenkundige rij kan de som eenvoudig bepaald worden: het gemiddelde van de eerste en de laatste term is $\frac{1}{2}(a+(n-1)v+a)$, en dat geldt ook voor het gemiddelde van de tweede en vóórlaatste term, enzovoorts. Dus de som is n maal dat gemiddelde.
   Definieer de functie *somRR* (som Rekenkundige Rij) met:

   $$\mathit{somRR}\ a\ v\ n \quad = \quad \text{som van de rekenkundige rij } a,\ a+v,\ \dots\ ,\ a+(n-1)v$$

   Reken daarmee 23+24+...+43 uit, en ook 1+34+...+1717.
9. Definieer in Miranda de functie *lengteRR* die, gegeven de eerste, tweede en laatste term van een rekenkundige rij, het aantal termen in die rij oplevert:

   $$\mathit{lengteRR}\ a\ b\ z \quad = \quad \text{het aantal termen in de rekenkundige rij } a, b, \dots, z$$

   Bijvoorbeeld, *lengteRR* 1 34 1717 levert 53. Ga na dat 1+34+...+1717 de uitkomst is van: *somRR* 1 (34−1) (*lengteRR* 1 34 1717).
   Definieer nu een functie *somRR′* met de volgende eigenschap. Voor iedere *rekenkundige* rij $a, b, \dots, z$ geldt:

   $$\mathit{somRR}'\ a\ b\ z \quad = \quad \text{de som van de rekenkundige rij } a, b, \dots, z$$

   Gebruik de functies *somRR* en *lengteRR*.

Om te onthouden

• Haakjes geven aan wat-bij-wat hoort; een functie–argument binding heeft voorrang boven een operator–operand binding.

• Het kan niet vaak genoeg gezegd worden:
Haakjes worden uitsluitend gebruikt om aan te geven wat-bij-wat hoort; een functie–argument binding heeft voorrang boven een operator–operand binding.

• Spaties hebben geen invloed op de bepaling van wat-bij-wat hoort. (Ze zijn wel medebepalend voor de lexicale eenheden.)

• Bij elke definitie (en nergens anders) mag een where-part staan. In where-parts staan hulpdefinities. Wat locaal in het where-part gedefinieerd wordt, is alleen in het rechterlid van de definitie bekend. Is de definitie een functiedefinitie, dan zijn de parameters van de functie binnen het where-part bekend.

3 Elementaire waarden

De elementaire ('ondeelbare') waarden waarmee gerekend wordt zijn: getallen, karakters en waarheidswaarden. Voor elk van deze soort waarden zijn er specifieke operaties en notaties.

§3.1 Getallen. We hebben al gezien dat je in Miranda met getallen kunt rekenen: willekeurig gehele getallen en sommige rationale getallen. De notatie is precies zoals je gewend bent: 0 voor het getal nul, 1 voor het getal één, enzovoorts. Ook de operaties op getallen zien er bekend uit: + voor optelling, × voor vermenigvuldiging, / voor deling. Minder bekende operaties en functies zijn deze:

$$\begin{array}{lcl}
\textit{entier } x & = & \text{grootste gehele getal dat hoogstens } x \text{ is} \\
 & & \quad (\text{bijv: } \textit{entier } 3.7 \ = \ 3) \\
m \textit{ div } n & = & \text{gehele deling van gehele } m \text{ door gehele } n\text{: } \textit{entier } (m/n) \\
 & & \quad (\text{bijv: } 7 \textit{ div } 3 \ = \ 2) \\
m \textit{ mod } n & = & \text{rest na gehele deling van gehele } m \text{ door gehele } n \\
 & & \quad (\text{bijv: } 7 \textit{ mod } 3 \ = \ 1)\text{; er geldt:} \\
 & & (m \textit{ div } n) \times n + (m \textit{ mod } n) \ = \ m
\end{array}$$

In Miranda is het teken (negatief, positief) van m *mod* n dat van n, en hebben *div* en *mod* dezelfde voorrang als vermenigvuldigen en delen.

Naast getallen bestaan er nog andere soorten gegevens. Miranda kent met name de karakters en waarheidswaarden.

§3.2 Karakters. De *karakters* zijn de honderd en achtentwintig symbolen waaruit teksten zijn opgebouwd; de *zichtbare* karakters zijn: `!"#$%&'()*+,-...012...9ABC...Za...xyz...` (inclusief de spatie). Daarnaast zijn er nog heel wat *onzichtbare* karakters, zoals het *newline* karakter. Een karakter in Miranda wordt genoteerd door er een quote vóór en achter te zetten: bijvoorbeeld '!', '"',..., '0',..., 'a', enzovoorts. Er zijn een paar karakters die een bijzondere notatie hebben, onder andere de quote en de onzichtbare karakters.

Vergis je niet tussen het *cijfer* '0' en het *getal* 0; cijfers zijn karakters, getallen worden met cijfers genoteerd. Let ook op het verschil tussen het karakter 'x' en de variable x.

Met karakters zijn maar een paar bewerkingen mogelijk. De belangrijkste zijn *code* en *decode*. Functie *code* levert bij ieder karakter een getal op (de "code" voor dat karakter), en decode doet het omgekeerde. Met name:

$x =$	...	'A'	'B'	...	'a'	'b'	...	$=$ *decode* n
code $x =$	...	65	66	...	97	98	...	$= n$

Van de functie *code* hoef je eigenlijk niet méér te weten dan dat de code van opeenvolgende hoofdletters opeenvolgende getallen zijn; net zo voor kleine letters en voor cijfers. Op alle waarden in Miranda zijn de vergelijkingsoperatoren $<, \leq, =, \neq, \geq, >$ gedefinieerd. De ordening van karakters ziet er zo uit:

$$\ldots < {}'A' < {}'B' < \ldots < {}'Z' < \ldots < {}'a' < {}'b' < \ldots < {}'z' < \ldots$$

Voor karakters x en y geldt $x < y$ precies wanneer $code\ x \ < \ code\ y$.

Een klein voorbeeld; straks volgen er meer. Laat $kLetter$ de functie zijn die, gegeven een hoofdletter, de corresponderende kleine letter oplevert. Bijvoorbeeld, $kLetter\ {}'C' = {}'c'$. Die kunnen we als volgt definiëren:

$$kLetter\ x \ = \ decode\ (code\ x + 32)$$

Mooier is het om de definitie onafhankelijk te maken van de precieze codes:

$$kLetter\ x \ = \ decode\ (code\ x + (code\ {}'a' - code\ {}'A'))$$

§3.3 Waarheidswaarden. Er zijn twee waarheidswaarden, namelijk 'waar' en 'onwaar'. Ze worden genoteerd met *True* en *False* (let op de hoofdletters!). Deze waarden worden zelden expliciet genoteerd; meestal "ontstaan" ze uit vergelijkingen '$\ldots \leq \ldots$' en worden ze gecombineerd door middel van de operatoren $\wedge$ ("én"), $\vee$ ("of"), $\neg$ ("niet"):

$$\begin{array}{lcl}
\neg x & = & \mathit{True} \text{ precies wanneer } x = \mathit{False} \\
x \wedge y & = & \mathit{True} \text{ precies wanneer } x \text{ én } y \text{ beide } \mathit{True} \text{ zijn} \\
x \vee y & = & \mathit{True} \text{ precies wanneer } x \text{ of } y \text{ of beide } \mathit{True} \text{ zijn}
\end{array}$$

In alle andere gevallen zijn de uitkomsten *False*.

§3.4 Voorbeelden. Waarheidswaarden worden veel gebruikt in gevalsonderscheid. De functie *abs* die de absolute waarde van zijn argument oplevert, kunnen we in Miranda als volgt definiëren:

$$\begin{array}{l}
abs\ x \\
= \ x, \ \mathit{if}\ 0 \leq x \\
= \ -x, \ \mathit{if}\ x \leq 0
\end{array}$$

Deze definitie heeft twee *alternatieven*; ieder alternatief heeft een *bewaking* *if* .... Bij een aanroep van de functie kiest Miranda het éérste (bovenste) alternatief waarvan de bewakingsvoorwaarde *True* oplevert. De definitie van *abs* verandert niet van betekenis, als we de bewaking van het tweede alternatief vervangen door $\mathit{if}\ x < 0$.

De standaard functie $max2$ levert de grootste van zijn twee argumenten:

$$\begin{array}{l}
max2\ x\ y \\
= \ x, \ \mathit{if}\ y \ \leq \ x \\
= \ y, \ \mathit{if}\ x \ \leq \ y
\end{array}$$

Met $max2$ kunnen we *abs* eenvoudiger definiëren:

$$abs\ x \ = \ max2\ x\ (-x)$$

Dit kunnen we ook noteren met

$$abs\ x \ = \ x \ \underline{max2} \ (-x)$$

Door $max2$ te noteren als $\underline{max2}$ (zie §1.1) wordt die naam een operatorsymbool dat, net als de symbolen + en × etc, *tussen* de argumenten in moet staan. Dit geldt ook in de linkerkant van definities. (Operatoren *div* en *mod* zijn de enige twee namen die zélf al als operatie-*symbolen* beschouwd worden; zij hoeven dus niet als $\underline{div}$ en $\underline{mod}$ genoteerd te worden.)

Beschouw nu eens de functies met de volgende specificatie:

$$\begin{array}{lcl} isLetter\ x & = & x \text{ is een letter} \\ iskLetter\ x & = & x \text{ is een kleine letter} \\ ishLetter\ x & = & x \text{ is een hoofdletter} \\ kLetter\ x & = & \text{de kleine-letter versie van letter } x \end{array}$$

De eerste drie functies leveren een waarheidswaarde op; de uitkomst is *True* of *False*. Een functie die een waarheidswaarde oplevert, heet *predicaat.* Deze functies kunnen we in Miranda als volgt definiëren:

$$\begin{array}{l} isLetter\ x \ = \ iskLetter\ x \ \vee \ ishLetter\ x \\ iskLetter\ x \ = \ 'a' \ \le \ x \ \le \ 'z' \\ ishLetter\ x \ = \ 'A' \ \le \ x \ \le \ 'Z' \\ kLetter\ x \\ \ = \ x,\ if\ iskLetter\ x \\ \ = \ decode\ (code\ x + code\ 'a' - code\ 'A'),\ if\ ishLetter\ x \end{array}$$

Merk op dat *kLetter* geen uitkomst heeft in geval zijn argument geen letter is. Als *kLetter* '0' wordt uitgerekend, volgt er een foutstop. Om de foutstop van een zinvolle melding te voorzien, kunnen we een extra alternatief toevoegen:

$$\begin{array}{l} kLetter\ x \\ \ = \ x,\ if\ iskLetter\ x \\ \ = \ decode\ (code\ x + code\ 'a' - code\ 'A'),\ if\ ishLetter\ x \\ \ = \ error\ \texttt{"}kLetter:\ argument\ geen\ letter\texttt{"},\ otherwise \end{array}$$

De tekst `kLetter: argument geen letter` wordt nu afgedrukt bij een niet-letter argument. In deze definitie kunnen we in plaats van *otherwise* ook een van de volgende bewakingen schrijven:

$$\begin{array}{l} if\ \neg(iskLetter\ x \ \vee \ ishLetter\ x) \\ if\ \neg iskLetter\ x \ \wedge \ \neg ishLetter\ x \end{array}$$

De bewaking *otherwise* staat altijd voor de ontkenning van alle voorgaande bewakingen, en mag alleen als laatste voorkomen.

Opgaven

10. Definieer de functie $max3$ die de grootste van zijn drie argumenten oplevert. Doe dit een keer mét gebruik van $max2$ en een keer zónder $max2$ te gebruiken.

Definieer ook $max4$ (die het maximum van vier argumenten oplevert), eenmaal met behulp van $max2$, en eenmaal zonder $max2$ en $max3$ te gebruiken.

11. Definieer de functie *sign* die -1, 0, $+1$ oplevert al naar gelang het argument negatief, nul, positief is. Doe dit tweemaal: eenmaal een definitie met drie alternatieven, en eenmaal met twee alternatieven (en met gebruik van *abs* en *entier*).
12. Definieer de functie *eenheid* die het laatste cijfer (als getal) van zijn argument oplevert. Bijvoorbeeld, $\mathit{eenheid}\ 3628 = 8$.
13. Definieer de functie *vrnl* ('van rechts naar links') die bij argument n ($n \leq 999$) het getal oplevert waarvan de decimale notatie vrnl gelezen hetzelfde is als die van n vlnr gelezen. Bijvoorbeeld, $\mathit{vrnl}\ 572 = 275$.
    Wenk. Definieer hulpfuncties vrnl', vrnl'', ... die het zelfde doen voor argumenten in een geschikt gekozen deel-interval van $0 \mathrel{..} 999$.
    Gebruik *div* en *mod* en bedenk dat $n = (n\ \mathit{div}\ 10) \times 10 + (n\ \mathit{mod}\ 10)$.
14. Definieer functies *isCijfer* en *hLetter* analoog aan *isLetter* en *kLetter*.
15. Definieer een functie *cijferNgetal* ("cijfer naar getal") die de cijferkarakters omzet naar hun getalwaarde. Bijvoorbeeld, $\mathit{cijferNgetal}\ '3'$ levert het getal drie. Bij niet-cijfer argumenten moet er een foutstop met passende melding resulteren.
16. Net zo als in de vorige opgave, maar nu van getal naar cijfer. Dus $\mathit{getalNcijfer}\ 3$ levert $'3'$. Bij andere argumenten dan $0, 1 \mathrel{..} 9$ moet er een foutstop met passende melding resulteren.
17. Definieer de functie *letterNa* die voor iedere letter de volgende letter oplevert. Bijvoorbeeld, $\mathit{letterNa}\ 'c'$ levert $'d'$. Spreek af dat na $'A'$ de $'Z'$ komt, en na $'a'$ de $'z'$.
18. Een bekend gezelschapsspel. De personen tellen hardop, om de beurt in een kring: 0, 1, ... Wanneer het getal een zevenvoud is of in de decimale notatie een zeven bevat, moet de persoon die aan de beurt is, niet het getal zeggen maar 'piep'; anders krijgt-ie met een beroete kurk een dikke stip op z'n voorhoofd gewreven. Definieer het predicaat *piep* (een functie met waarheidswaarden als uitkomst) voor argumenten $0 \leq n \leq 999$:

    $$\mathit{piep}\ n \quad = \quad \text{`}n \text{ is een zevenvoud of bevat het cijfer 7'}$$

    Dus als de bewering over n waar is, is de uitkomst *True*; anders *False*.

Om te onthouden

• Gevalsonderscheid wordt in definities weer gegeven met "bewaakte alternatieven". De *if* en *otherwise* kunnen nergens anders voorkomen dan in deze "bewakingen".

• Een definitie van de vorm

$$\begin{aligned} & f\ x \\ & = \mathit{True},\ \mathit{if}\ \mathit{cond} \\ & = \mathit{False},\ \mathit{otherwise} \end{aligned}$$

kun je eleganter formuleren als

$$f\ x \ = \ \mathit{cond}$$

- Door een functienaam *abc* te noteren als <u>*abc*</u> , wordt die naam een operatiesymbool dat tussen de argumenten in moet komen te staan. Als infix operatie-symbool heeft <u>*abc*</u> voorrang op alle andere infix operatie-symbolen. (*Functies blijven de hoogste voorrang houden!*)

4 Tupels en Lijsten

Naast de elementaire ('ondeelbare') waarden bestaan er ook 'samengestelde' waarden: *tupels* en *lijsten*.

§4.1 Tupels. Een *tupel* is een stel waarden, zoals (1, *True*, $'a'$) of (1, 2). Hier is een voorbeeld waarbij tupels een rol spelen:

$$wortel1\ (a, b, c)\ =\ (-b - sqrt\ (b\hat{}2 - 4{\times}a{\times}c))\ /\ (2{\times}a)$$

Deze functie *wortel*1 heeft één parameter, een drie-tupel (a, b, c), terwijl de eerder gedefinieerde functies *wortel*1 in §2.1 en 2.2 steeds drie parameters hadden. In het rechterlid van de definitie zijn de componenten van het argument (een tupel) bekend onder de namen a, b, c. Dat komt doordat ze in de parametervorm benoemd zijn. (Op het voordeel van drie losse parameters boven een vast drietal als parameter komen we terug in §11.1.)

Nog een voorbeeld:

$$\begin{aligned}
&wortel12\ a\ b\ c\\
&=\ ((-b - discr)/(2{\times}a),\ \ (-b + discr)/(2{\times}a))\\
&\quad where\\
&\quad discr\ =\ sqrt\ (b\hat{}2 - 4{\times}a{\times}c)
\end{aligned}$$

Functie *wortel*12 heeft als het ware twee resultaten, namelijk één tupel met twee componenten.

Voor tweetallen zijn er de standaard functies *fst* en *snd*, die het eerste en tweede lid van een tweetal opleveren. Voor drietallen kun je desgewenst zelf definiërem:

$$\begin{aligned}
&fst3\ (x, y, z)\ =\ x\\
&snd3\ (x, y, x)\ =\ y\\
&thd3\ (x, y, z)\ =\ z
\end{aligned}$$

§4.2 Lijsten. Rijen van gelijksoortige dingen heten in Miranda *lijsten*. Ze kunnen genoteerd worden door hun *elementen* op te sommen tussen vierkante haken:

$$\begin{aligned}
&[2,\ 3,\ 5,\ 7,\ 11]\\
&[True,\ False,\ True]\\
&[\,]\\
&[\ \ (1, True),\ (2, False),\ (3, False)\ ]\\
&[\ \ [1,2],\ \ [3,4,5],\ \ [\,],\ \ [6,7,8,9]\ \ ]\\
&['b',\ 'e',\ 'e',\ 'r']
\end{aligned}$$

De lijst [ ] heet *de lege lijst*. Er is geen beperking op het soort van dingen die in een lijst zitten. Let op: [1, *True*, $'a'$] is niet geoorloofd, want in een lijst moeten alle elementen van eenzelfde soort zijn.

In een lijst met n elementen heeft het eerste element rangnummer 0 en het laatste element rangnummer $n-1$. Schematisch geven we een lijst met n elementen ($n \geq 0$) vaak aan met $[x_0, x_1, \ldots, x_{n-1}]$. Er zijn ook nog wat handige afkortingen voor getallijsten met een constant verschil tussen opeenvolgende elementen:

$$\begin{array}{lcl} [2\,..\,11] & = & [2,3,4,5,6,7,8,9,11] \\ [2,5\,..\,18] & = & [2,5,8,11,14,17] \\ [10,9\,..\,3] & = & [10,9,8,7,6,5,4,3] \end{array}$$

Lijsten kunnen ook oneindig lang zijn:

$$\begin{array}{lcl} [2..] & = & [2,3,4,5,6,7,8,9,11,\ldots \\ [2,5..] & = & [2,5,8,11,14,17,\ldots \\ [10,9..] & = & [10,9,8,7,6,5,4,3,\ldots \end{array}$$

Met 'Control-C' kun je een Miranda-berekening onderbreken, dus ook het tonen van een oneindige lijst.

§4.3 Lijstoperaties. Naast de notatie van lijsten, zoals hierboven behandeld, zijn er ook operaties op lijsten. Zij hebben een lijst als argument en/of een lijst als resultaat. Hier zijn een paar voorbeelden (met een toelichting erna):

$$\begin{array}{lcl} [a,b,c] +\!\!+ [d,e] & = & [a,b,c,d,e] \\ \#[a,b,c,d,e] & = & 5 \\ [a,b,c,d,e] \mathbin{!} 3 & = & d \\ \mathit{take}\ 2\ [a,b,c,d,e] & = & [a,b] \\ \mathit{drop}\ 2\ [a,b,c,d,e] & = & [c,d,e] \\ \mathit{max}\ [a,b,c,d,e] & = & \text{de grootste van } a,b,c,d,e \\ \mathit{min}\ [a,b,c,d,e] & = & \text{de kleinste van } a,b,c,d,e \\ \mathit{sum}\ [a,b,c,d,e] & = & a+b+c+d+e \\ \mathit{product}\ [a,b,c,d,e] & = & a\times b\times c\times d\times e \end{array}$$

Operatie $+\!\!+$ voegt twee lijsten aaneen; de uitdrukkingen $1 +\!\!+ [2,3]$ en $[1,2] +\!\!+ 3$ zijn onzin, want het linker en rechter argument van $+\!\!+$ moet een lijst zijn. Operatie $\#$ geeft de lengte van een lijst: $\#xs$ is het aantal elementen in xs. Operatie ! heet *indicering*; $xs!3$ is het element van lijst xs met rangnummer 3 (dus het element op de vierde plaats). En $\mathit{take}\ m\ xs$ levert het beginstuk van xs ter lengte m; $\mathit{drop}\ m\ xs$ levert de rest op. Wanneer $m > \#xs$, levert $\mathit{take}\ m\ xs$ de gehele lijst xs op en $\mathit{drop}\ m\ xs$ de lege lijst. Er geldt dus *altijd* dat $\mathit{take}\ m\ xs\ +\!\!+\ \mathit{drop}\ m\ xs$ gelijk is aan xs. Herinner je dat we '$\mathit{take}\ m\ xs$' ook mogen schrijven als '$m\ \underline{\mathit{take}}\ xs$'. We zullen verderop zien hoe je al deze functies, en nog veel meer, zelf kunt definiëren.

Opgaven

19. Reken uit:
    a. $\#[]$ b. $\#[[1,2],[3],[4,5,6]]$ c. $\#[\ []\ ]$ d. $\#[\ [\ []\ ]\ ]$.
20. Definieer de functie $somRR'$ van opgave 9 op een eenvoudiger manier.

21. Definieer *somRR* van opgave 8 op een eenvoudiger manier. Gebruik *take* en een oneindige lijst (om “moeilijke uitdrukkingen” zoals $a+(n-1)\times v$ te vermijden).
22. Definieer de faculteitsfunctie (Engels: factorial, niet faculty): $\mathit{fac}\ n = n\times(n-1)\times\ \ldots\ \times 1$.
23. Definieer de standaard functie *index* met $\mathit{index}\ \mathit{xs} = [0\mathinner{..}\#\mathit{xs}-1]$ met behulp van *take* en $[0\mathinner{..}]$.
24. Geef een definitie voor de standaard functies *hd* (‘head’), *tl* (‘tail’), *init* (‘initiële deel’) en *last*:

$$\begin{aligned}
\mathit{hd}\ [x_0,\ldots,x_n] &= x_0\\
\mathit{tl}\ [x_0,\ldots,x_n] &= [x_1,\ldots,x_n]\\
\mathit{init}\ [x_0,\ldots,x_n] &= [x_0,\ldots,x_{n-1}]\\
\mathit{last}\ [x_0,\ldots,x_n] &= x_n
\end{aligned}$$

Hierbij is $n \geq 0$, dus de argument lijst is niet leeg. Voor een lege lijst resulteert er een foutstop. Gebruik operatie ! en functies *take* en *drop*.
25. Een *segment* van $[x_0, x_1, \ldots, x_{n-1}]$ is een aaneengesloten deel van de vorm: $[x_i, \ldots, x_{j-1}]$ met $0 \leq i \leq j \leq n$. Dus ook de lege lijst $(i = j)$ en de lijst zelf $((i,j) = (0,n))$ zijn een segment. Definieer de functie *segment* met:

$$\mathit{segment}\ [x_0, x_1, \ldots, x_{n-1}]\ i\ j \quad = \quad [x_i, \ldots, x_{j-1}]$$

26. Definieer de volgende functie:

swap xs i j = de lijst ontstaan uit *xs* door de elementen met rangnummers *i* en *j* van positie te wisselen

Doe dit eenmaal met gebruik van *segment* (opgave 25), en eenmaal zonder *segment*, *take* en *drop* te gebruiken.
27. Definieer de functie $\mathit{rotatieL}\ [x_0, x_1, \ldots, x_{n-1}] = [x_1, \ldots, x_{n-1}, x_0]$.
Definieer *rotatieR* analoog.
Wat is het resultaat van je functie op de lege lijst?

Om te onthouden

- De elementen in een lijst moeten alle van eenzelfde soort zijn; dat hoeft bij tupels niet.
- De elementen in een lijst mogen willekeurige zijn, bijvoorbeeld getallen of tupels, maar ook lijsten of wat-dan-ook.
- De lijsten [ ] en [[ ]] verschillen: de eerste heeft lengte 0, terwijl de tweede lengte 1 heeft.
- De rangnummers in een lijst beginnen bij 0, dus het grootste rangnummer is één minder dan de lengte.
- De volgende functies en operaties op lijsten zijn standaard aanwezig: $+\!\!+$, !, #, *take*, *drop*, *max*, *min*, *sum*, *product*, *index*, *hd*, *tl*, *init*, *last*.

5 Lijstcomprehensie

Lijstcomprehensie is een notatie voor lijsten. Het is een handig en krachtig uitdrukkingsmiddel.

§5.1 Heel handig en belangrijk is de volgende soort uitdrukking, die *lijstcomprehensie* heet:

$$\begin{array}{lcl}
[2{\times}x \mid x \leftarrow [3,4,6,1]] & = & [2{\times}3,\ 2{\times}4,\ 2{\times}6,\ 2{\times}1] \\
[x{+}y \mid x \leftarrow [a,b];\ y \leftarrow [c,d,e]] & = & [a{+}c, a{+}d, a{+}e, b{+}c, b{+}d, b{+}e] \\
[y \mid x \leftarrow [3\mathinner{..}5];\ y \leftarrow [x{+}1\mathinner{..}10];\ y\ \mathit{mod}\ x\ =\ 0] & = & [6,\ 9,\ 8,\ 10]
\end{array}$$

We lezen de laatste uitdrukking hierboven als:

> de lijst van alle waarden y waarbij x komt uit $[3\mathinner{..}5]$, en (bij iedere x) y komt uit $[x{+}1\mathinner{..}10]$, en x, y voldoen aan $y\ \mathit{mod}\ x\ =\ 0$ (dat wil zeggen, y is een veelvoud van x).

Een vrij algemene vorm van lijstcomprehensie is:

$$[\mathit{expr} \mid x_1{\leftarrow}\mathit{lijst}_1;\ \mathit{cond}_1;\ \ldots;\ x_n{\leftarrow}\mathit{lijst}_n;\ \mathit{cond}_n]$$

Hierin is *expr* een uitdrukking die de elementen in de resultaatlijst beschrijft. Elk deel $x \leftarrow \mathit{lijst}$ heet *generator*, en geeft de waarden die x achtereenvolgens aanneemt. Elke *cond* is een conditie; de waarden die niet aan de conditie voldoen, doen niet mee in de vorming van de resultaatlijst. De variabele x van een generator $x{\leftarrow}\mathit{lijst}$ mag gebruikt worden in de expressie *expr* helemaal links en verder *uitsluitend rechts* van de generator. De lijsten lijst_i hoeven geen getallijsten te zijn; ze mogen willekeurige lijsten zijn. We zullen in de loop van de tijd nog een paar variaties op deze algemene vorm tegenkomen. Met name:

$$[\mathit{expr} \mid\ \ldots(x,y){\leftarrow}\mathit{lijst};\ \ldots]$$

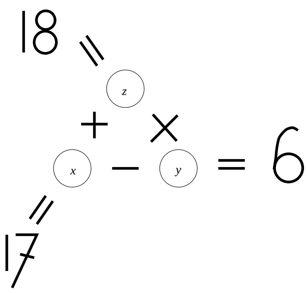
Hier zijn de elementen van *lijst* kennelijk tupels; de componenten van zo'n tupel zijn benoemd met x en y. Dus wanneer *lijst* tweetallen bevat, mag een generator ook de vorm $(x,y){\leftarrow}\mathit{lijst}$ hebben; net zo bij drietallen, enzovoorts.

Let op: het gebruik van de indiceringsoperatie ! in een lijstcomprehensie is vaak onelegant (en inefficiënt). Vergelijk bijvoorbeeld:

$$\mathit{product}\ [1 + \mathit{xs}!i \mid i \leftarrow [0\mathinner{..}\#\mathit{xs}{-}1]]$$
$$\mathit{product}\ [1 + x \mid x \leftarrow \mathit{xs}]$$

In een lijstcomprehensie kun je de individuele elementen benoemen met een generator in plaats van ze te selecteren middels indicering.

§5.2 Voorbeeld. Beschouw de volgende puzzel:



In woorden: vind drie getallen x, y, z waarvoor geldt dat $x{-}y = 6$ en $z{+}x = 17$, en $y{\times}z = 18$ en $1 \le x, y, z \le 32$. Dit soort puzzels vind je vaak in puzzelblaadjes; dit is een heel eenvoudige versie. Eén oplossing voor (x, y, z) is $(8, 2, 9)$. De lijst van álle oplossingen wordt gegeven door:

$$[(x, y, z) \mid x, y, z \leftarrow [1 \mathinner{..} 32];\ x{-}y = 6;\ z{+}x = 17;\ y{\times}z = 18]$$

dat wil zeggen:

$$[(x, y, z) \mid x \leftarrow [1 \mathinner{..} 32];\ y \leftarrow [1 \mathinner{..} 32];\ z \leftarrow [1 \mathinner{..} 32];\ x{-}y = 6;\ z{+}x = 17;\ y{\times}z = 18]$$

Maar er bestaat een véél efficiëntere uitdrukking voor dezelfde lijst. Immers, als x al gekozen is, is er wegens de conditie $x{-}y = 6$ geen vrijheid meer voor y en ligt ook z vast wegens de conditie $z{+}x = 17$. Dus bovenstaande lijst van alle oplossingen is gelijk aan:

$$[(x, y, z) \mid x \leftarrow [1 \mathinner{..} 32];\ y \leftarrow [x{-}6];\ 1 \le y \le 32;\ z \leftarrow [17{-}x];\ 1 \le z \le 32;\ y{\times}z = 18]$$

De condities $1 \le y \le 32$ en $1 \le z \le 32$ komen voort uit de probleemstelling. In deze versie worden er maar $32{\times}1{\times}1$ drietallen “geprobeerd”; in de vorige versies werden er $32{\times}32{\times}32$ drietallen geprobeerd — da’s ruim duizend maal zoveel! Overigens, $x{-}6$ en $17{-}x$ liggen in het interval $1 \mathinner{..} 32$ uitsluitend als $7 \le x$ en $x \le 16$. Dus de lijst van alle oplossingen is gelijk aan:

$$[(x, y, z) \mid x \leftarrow [7 \mathinner{..} 16];\ y \leftarrow [x{-}6];\ z \leftarrow [17{-}x];\ y{\times}z = 18]$$

We hebben hiermee het aantal te onderzoeken drietallen nogmaals verminderd. De meest efficiënte uitdrukking voor de lijst van alle oplossingen is deze:

$$[(8, 2, 9),\ (15, 9, 2)]$$

Maar… ten eerste vereist het wat meer denkwerk om in te zien dat dit inderdaad gelijk is aan voorgaande uitdrukkingen, en ten tweede leent deze uitdrukking zich niet zo goed voor veralgemeningen van de puzzel (met andere waarden voor 32, 6, 17 en 18).

De “eerste de beste” oplossing van de puzzel wordt gegeven door:

$$\mathit{take}\ 1\ [(x, y, z) \mid x \leftarrow [7 \mathinner{..} 16];\ y \leftarrow [x{-}6];\ z \leftarrow [17{-}x];\ y{\times}z = 18]$$

De uitkomst hiervan is een lijst met één drietal (x, y, z), of de lege lijst wanneer er geen oplossingen zijn. Er worden niet meer drietallen (x, y, z) berekend dan nodig is om het eerste element van de lijst van alle oplossings-drietallen te bepalen.

Opgaven

Opgaven over de puzzel.

28. Wat is er fout in de volgende uitdrukkingen voor de lijst van alle oplossingen van bovengenoemde puzzel?

$$[(x, y, z) \mid x \leftarrow [1\,..\,32];\ x-y = 6;\ y \leftarrow [1\,..\,32];\ z \leftarrow [1\,..\,32];\ z+x = 17;\ y\times z = 18]$$
$$[(x, y, z) \mid x \leftarrow [1\,..\,32];\ y \leftarrow x-6;\ z \leftarrow [17-x];\ y\times z = 18]$$
$$[(x, y, z) \mid x \leftarrow [1\,..\,32];\ y\ =\ x-6;\ z \leftarrow [17-x];\ y\times z = 18]$$

29. Is er enig verschil in uitkomst te bemerken tussen de volgende uitdrukkingen:

$$[(x, y, z) \mid x, y, z \leftarrow [1\,..\,32];\ x-y = 6;\ z+x = 17;\ y\times z = 18]$$
$$[(x, y, z) \mid z, y, x \leftarrow [1\,..\,32];\ x-y = 6;\ z+x = 17;\ y\times z = 18]$$

30. Geef een uitdrukking voor het aantal oplossingen van de puzzel.
31. We gaan de puzzel veralgemenen door de specifieke constanten 32, 6, 17, 18 tot parameter te maken. Definieer in Miranda een functie *opln* die voldoet aan:

$$\begin{aligned} \mathit{opln}\ n\ (a, b, c) \quad = \quad & \text{een lijst van alle drietallen } (x, y, z) \text{ met:} \\ & x, y, z \in 1\,..\,n,\ x-y = a,\ z+x = b,\ y\times z = c \end{aligned}$$

Gebruik deze functie om alle oplossingen voor bovenstaande puzzel uit te rekenen. Experimenteer ook eens met andere waarden voor n, a, b, c, maar kies n niet te groot!

32. We gaan nu geen puzzels oplossen, maar de getallen voor bijzondere puzzels construeren. Definieer een functie *puzzels* die voldoet aan:

$$\begin{aligned} \mathit{puzzels}\ k \quad = \quad & \text{een lijst van combinaties } n, (a, b, c) \text{ (met } n \in 0\,..\,15) \\ & \text{waarvoor functie } \mathit{opln} \text{ precies } k \text{ oplossingen levert.} \end{aligned}$$

Veralgemeen de functie *puzzels* door de specifieke constante 15 tot parameter te maken.

Opgaven

33. Definieer de functie *aantal* met: $x\ \underline{\mathit{aantal}}\ xs$ = het aantal keren dat x voorkomt in xs.
34. Voor de lidmaatschapstest is er een standaard functie (predicaat) *member*:

$$\mathit{member}\ [x_0, \ldots, x_{n-1}]\ y \quad = \quad y = x_0 \vee \ldots \vee y = x_{n-1}$$

Bijvoorbeeld, $\mathit{member}\ [1, 7, 3, 1]\ 4 = \mathit{False}$, en $[1, 7, 3, 1]\ \underline{\mathit{member}}\ 3 = \mathit{True}$. We schrijven informeel vaak $x \in xs$ voor $\mathit{member}\ xs\ x$. Definieer de functie *member*.

35. Definieer een functie *delers* met: $\mathit{delers}\ n$ = een lijst van alle delers van n die groter zijn dan 1 en kleiner dan n. Gebruik deze functie om een uitdrukking te geven voor de lijst van alle positieve delers van n, inclusief 1 en n zelf. (Herinner je dat $n\ \mathit{mod}\ d\ =\ 0$ precies wanneer d een deler is van n.)

36. Definieer de functie *priem* met:

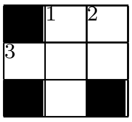
$$priem\ n \quad = \quad \text{‘}n \text{ is een priemgetal’}$$

Dat wil zeggen, de uitkomst van *priem* n is *True* precies wanneer $n \geq 2$ en n heeft geen andere delers dan 1 en n. De functie is dus een predicaat. Wenk: gebruik *delers* van opgave 35.

37. Definieer de functie *ggd* die de grootste gemeenschappelijke deler van zijn twee argumenten oplevert. Bijvoorbeeld, 42 $\underline{ggd}$ 60 = 6.

38. Definieer een functie *pythagoras* met *pythagoras* n = een lijst van alle drietallen natuurlijke getallen (x, y, z) met $1 \leq x \leq y \leq n$ en $x^2+y^2 = z^2$. Wenk: laat y variëren over een lijst van getallen vanaf x, en net zo iets voor z.

39. Los het volgende kruisgetalraadsel op: per vakje één cijfer uit 1 . . 9 (dus geen 0).



Hor 1: kleiner dan twintig.
Hor 3: de som der cijfers hiervan is Hor 1.
Vert 1: $\frac{1}{6}$-de van Hor 3.
Vert 2: dit getal plus dit-getal-achterste-voren is gelijk aan Vert 1.

40. Definieer de oneindige lijst *dias*:

$$[\underbrace{(0,0)}, \underbrace{(0,1),(1,0)}, \underbrace{(0,2),(1,1),(2,0)}, \ldots, \underbrace{(0,n)\cdots(n,0)}, \ldots]$$

Deze lijst bestaat uit delen van de vorm $(0, n), \ldots, (i, j), \ldots, (n, 0)$ met opklimmende waarden van i en dalende j, steeds met $i+j = n$. (De naam *dia* komt van ‘diagonaal’; lijst *dias* bevat de getallen van $N \times N$ in volgorde van de opeenvolgende, steeds verder van (0,0) verwijderde diagonalen.)
Geef ook een definitie voor de eerste honderd getallen uit die lijst.

41. Definieer een functie *inMunten* die gegeven een bedrag x, uitgedrukt in centen, alle mogelijkheden opsomt om x uit te betalen in stuivers, dubbeltjes, kwartjes en guldens. (Wenk: de uitkomst van *inMunten* x is een lijst van viertallen (s, d, k, g); wat moet er voor zo'n viertal gelden? Wat zijn de mogelijke waarden voor s? en voor d enzovoorts.)
Definieer ook een functie $inMunten'$ die de uitbetaling geeft in zo min mogelijk munten.

42. Het getallenpaar $(12, 42)$ heeft de bijzondere eigenschap dat het product van die twee gelijk is aan dat van $(21, 24)$, de omgedraaiden van die twee: $12\times42 = 21\times24$. Definieer de lijst van alle getalparen (i, j) (met $0 \leq i, j < 100$) waarvoor geldt dat $i\times j = i'\times j'$ waarbij de decimale notatie van i' de omgedraaide is van de decimale notatie van i, en net zo voor j'. Wenk: gebruik zo nodig de functie *vrnl* van opgave 13. Kun je het ook zonder de functie *vrnl*?

43. Gegeven is een eenvoudig intern telefoonboek van de faculteit:

$$telboek \ = \ [(\texttt{"}Peter\texttt{"}, 3683),\ (\texttt{"}Herman\texttt{"},\ 3772),\ (\texttt{"}Klaas\texttt{"}, 3783),\ \ldots]$$

De persoonsnamen zoals "*Peter*" zijn geoorloofde Miranda waarden; als x en y zulke namen zijn, is de test $x = y$ geoorloofd. Definieer de volgende lijsten en functies:

tels	=	een lijst van alle *nr*'s met $(..., nr) \in telboek$
tel nm	=	een lijst van alle *nr*'s met $(nm, nr) \in telboek$
abos	=	een lijst van alle *nm*'n met $(nm, ...) \in telboek$
abo nr	=	een lijst van alle *nm*'n met $(nm, nr) \in telboek$

44. De *standaard deviatie* van $n+1$ getallen $x_0, \ldots, x_n$ is:

$$\sqrt{\frac{\sum_{i=0}^{n}(x_i - \overline{x})^2}{n-1}}$$

waarbij $\overline{x}$ het gemiddelde is van de getallen. Definieer de functie *stdDeviatie* die, gegeven een lijst $[x_0, \ldots, x_n]$, de standaard deviatie ervan oplevert.

En nu nog wat theorie-vraagjes. . .

Opgaven

45. Reken uit: a. $[\#xs \mid xs \leftarrow []]$
b. $[\#xs \mid xs \leftarrow [\,[],\ [[]]\,]\,]$
c. $[xs +\!\!+\ zs \mid xs \leftarrow [\,[],\ [[]]\,];\ ys \leftarrow [];\ zs \leftarrow [[]]\,]$

46. Wat is de lengte van $[x{\times}y \mid x \leftarrow [1\,..\,4];\ y \leftarrow [3\,..\,8]]$?
Algemener, druk $\#[expr \mid x \leftarrow xs;\ y \leftarrow ys]$ uit in termen van $\#xs$ en $\#ys$.
Wat is de uitkomst van $[expr \mid x \leftarrow xs;\ y \leftarrow ys]$ wanneer $xs\ =\ []$, en wanneer $ys\ =\ []$?

Om te onthouden

- De lijst $[expr \mid \ldots\ x \leftarrow xs\ \ldots]$ is leeg wanneer xs leeg is.
- Bij $[expr \mid \ldots\ x \leftarrow [a]\ \ldots]$ neemt x precies één waarde aan: a. Dit is de enige manier om binnen een lijstcomprehensie een "locale definitie $x = a$" te geven.
- We benoemen lijsten met een "meervoud-s", bijvoorbeeld: xs voor een lijst van dingen (die we x zullen noemen), en xss voor een lijst van lijsten (die we stuk voor stuk xs zullen noemen — en de elementen daarvan dus x).
- Probeer altijd uitdrukkingen te veralgemenen door een constante in de probleemstelling tot parameter te maken.

6 Gegevensbestanden en combinatoriek

Veel bewerkingen op gegevensbestanden zijn eenvoudig met lijstcomprehensie uit te drukken. Ook verscheidene combinatorische functies op lijsten kunnen met lijstcomprehensie worden uitgedrukt.

Het Huis van Oranje

§6.1 We zijn geïnteresseerd in familie-relaties binnen het koninklijk huis, zoals: wie is de vader van Beatrix, wie zijn de zonen van Margriet, wie zijn de grootouders van Irene, enzovoort. Daartoe definiëren we twee lijsten, één van alle mannen en één van alle vrouwen, en definiëren we voorts één functie *kinderen* die bij iedere persoon de kinderen ervan oplevert (in een lijst).

Hier zijn onze definities:

$$
\begin{array}{l}
mannen \;=\\
[\texttt{"}Willem\ III\texttt{"},\ \texttt{"}Hendrik\texttt{"},\ \texttt{"}Bernhard\texttt{"},\ \texttt{"}Claus\texttt{"},\ \texttt{"}Pieter\texttt{"},\\
\ \texttt{"}Carel{-}Hugo\texttt{"},\ \texttt{"}Jorge\texttt{"},\ \texttt{"}Willem{-}Alexander\texttt{"},\ \texttt{"}Johan{-}Friso\texttt{"},\\
\ \texttt{"}Constantijn\texttt{"},\ \texttt{"}Maurits\texttt{"},\ \texttt{"}Bernhard\ jr.\texttt{"},\ \texttt{"}Pieter{-}Christiaan\texttt{"},\\
\ \texttt{"}Floris\texttt{"},\ \texttt{"}Carlos\texttt{"},\ \texttt{"}Jaime\texttt{"},\ \texttt{"}Bernardo\texttt{"},\ \texttt{"}Nicolas\texttt{"}]
\end{array}
$$

$$
\begin{array}{l}
vrouwen \;=\\
[\texttt{"}Emma\texttt{"},\ \texttt{"}Wilhelmina\texttt{"},\ \texttt{"}Juliana\texttt{"},\ \texttt{"}Beatrix\texttt{"},\ \texttt{"}Irene\texttt{"},\\
\ \texttt{"}Margriet\texttt{"},\ \texttt{"}Christina\texttt{"},\ \texttt{"}Margarita\texttt{"},\ \texttt{"}Maria\texttt{"},\ \texttt{"}Juliana\ jr.\texttt{"}]
\end{array}
$$

$$
\begin{array}{l}
kinderen\ x\\
\ = [\texttt{"}Wilhelmina\texttt{"}],\\
\qquad if\ x = \texttt{"}Willem\ III\texttt{"} \lor x = \texttt{"}Emma\texttt{"}\\
\ = [\texttt{"}Juliana\texttt{"}],\\
\qquad if\ x = \texttt{"}Hendrik\texttt{"} \lor x = \texttt{"}Wilhelmina\texttt{"}\\
\ = [\texttt{"}Beatrix\texttt{"},\ \texttt{"}Irene\texttt{"},\ \texttt{"}Margriet\texttt{"},\ \texttt{"}Christina\texttt{"}],\\
\qquad if\ x = \texttt{"}Bernhard\texttt{"} \lor x = \texttt{"}Juliana\texttt{"}\\
\ = [\texttt{"}Willem{-}Alexander\texttt{"},\ \texttt{"}Johan{-}Friso\texttt{"},\ \texttt{"}Constantijn\texttt{"}],\\
\qquad if\ x = \texttt{"}Claus\texttt{"} \lor x = \texttt{"}Beatrix\texttt{"}\\
\ = [\texttt{"}Maurits\texttt{"},\ \texttt{"}Bernhard\ jr.\texttt{"},\ \texttt{"}Pieter{-}Christiaan\texttt{"},\ \texttt{"}Floris\texttt{"}],\\
\qquad if\ x = \texttt{"}Pieter\texttt{"} \lor x = \texttt{"}Margriet\texttt{"}\\
\ = [\texttt{"}Carlos\texttt{"},\ \texttt{"}Margarita\texttt{"},\ \texttt{"}Jaime\texttt{"},\ \texttt{"}Maria\texttt{"}],\\
\qquad if\ x = \texttt{"}Carel{-}Hugo\texttt{"} \lor x = \texttt{"}Irene\texttt{"}\\
\ = [\texttt{"}Bernardo\texttt{"},\ \texttt{"}Nicolas\texttt{"},\ \texttt{"}Juliana\ jr.\texttt{"}],\\
\qquad if\ x = \texttt{"}Jorge\texttt{"} \lor x = \texttt{"}Christina\texttt{"}\\
\ = [\,],\\
\qquad otherwise
\end{array}
$$

Een uitdrukking voor de lijst van alle vaders van Beatrix (een lijst die precies één element zal bevatten) luidt:

$$[m \mid m \leftarrow mannen;\ kinderen\ m\ \underline{member}\ \texttt{"}Beatrix\texttt{"}]$$

Hierbij is *member* een standaard functie (een predicaat); de specificatie luidt:

$$xs\ \underline{member}\ x \;=\; \text{‘de lijst } xs \text{ bevat } x\text{’}$$

We maken "*Beatrix*" tot parameter en krijgen dan een functie, *vaders*:

$$vaders\ x \;=\; [m \mid m \leftarrow mannen;\ kinderen\ m\ \underline{member}\ x]$$

Dus *vaders* "*Beatrix*" levert de lijst van vaders van Beatrix op. Een nette afdruk van een lijst van persoonsnamen krijg je met *lay* en *layn*; bijvoorbeeld:

$$layn\ (vaders\ \texttt{"}Beatrix\texttt{"})$$

De functies *lay* en *layn* komen verderop aan bod.

Opgaven

47. Geef een uitdrukking voor het aantal mannen, en voor het aantal vrouwen en het aantal personen (die door bovenstaande definities bekend zijn).
48. Definieer de volgende functies, en test jouw (en onze!) kennis over het Huis van Oranje. Bijvoorbeeld, is *Jaime* vader van *Jorge*?
Begin steeds —zonodig— met een uitdrukking voor Beatrix of zo, en maak daarvan dan een functie door Beatrix tot parameter x te maken. Net zo als wij hierboven gedaan hebben bij de functie *vaders*.

$moeders\ x$	=	een lijst van alle moeders van x
$ouders\ x$	=	een lijst van alle ouders van x
$oudertal\ x$	=	het aantal ouders van x
$dochters\ x$	=	een lijst van alle dochters van x
$zonen\ x$	=	een lijst van alle zonen van x
$zussen\ x$	=	een lijst van alle zussen van x
$brussen\ x$	=	een lijst van elke brus (= broer of zus) van x
$tantes\ x$	=	een lijst van alle tantes van x
$kleinkinderen\ x$	=	een lijst van alle kleinkinderen van x
$grootouders\ x$	=	een lijst van alle grootouders van x
$x\ \underline{isVaderVan}\ y$	=	x is vader van y (uitkomst *True* of *False*)
$x\ \underline{isBrusVan}\ y$	=	x is brus (= broer of zus) van y

Geef ook een uitdrukking met als uitkomst (de waarheidswaarde van):

Jorge is een vader van Jaime

Combinatoriek

§6.2 In probleemformuleringen en in "eerste schetsen" van een oplossing komen vaak functies voor die delen van een lijst opleveren of combineren. Sommige van die functies zijn standaard al in Miranda aanwezig. Veel ervan komen in de opgaven hieronder aan bod.

De opgaven hebben tot doel te oefenen met lijstcomprehensie; bovendien leer je zo de functies kennen. De definities met behulp van lijstcomprehensie zijn niet altijd efficiënt. We zullen later de 'echte' definities behandelen.

Opgaven

49. De standaard functie *concat* (van *concatenate*, samen-ketenen) is gespecificeerd door:

$$\mathit{concat}\ [xs_0, xs_1, \ldots, xs_{n-1}] \quad = \quad xs_0 +\!\!+\ xs_1 +\!\!+ \cdots +\!\!+\ xs_{n-1}$$

Definieer in Miranda de functie *concat*. Wenk: $\mathit{concat}\ xss = [\ldots \mid xs \leftarrow xss;\ \ldots\ldots]$.

50. De *initiële delen* van $[x_0, \ldots, x_{n-1}]$ zijn de beginstukken $[]$, $[x_0]$, $[x_0, x_1]$, $\ldots$, $[x_0, \ldots, x_{n-1}]$. Dus *xs* zelf is altijd één van de initiële delen van *xs*, zelfs voor $xs = []$. Definieer de functie *inits* die de initiële delen van zijn argument oplevert. (Dat kunnen we later efficiënter: opgave 90.) Druk de lengte van *inits xs* uit in de lengte van *xs*.

51. De *tails* van $[x_0, \ldots, x_{n-1}]$ zijn de staartstukken $[x_0, \ldots, x_{n-1}]$, $[x_1, \ldots, x_{n-1}]$, $\ldots$, $[x_{n-1}]$, $[]$. Dus *xs* zelf is altijd één van de tails van *xs*, zelfs voor $xs = []$. Definieer de functie *tails* die de tails van zijn argument oplevert. (Dat kunnen we later efficiënter: opgave 90.)

52. Definieer de functies *group* en *ggroup* die een lijst in groepen (respectievelijk groepen van groepen) verdelen van gelijke lengtes. Bijvoorbeeld:

$$\begin{array}{lcl} \mathit{group}\ 3\ [0\,..\,12] & = & [[0,1,2],\ \ [3,4,5],\ \ [6,7,8],\ \ [9,10,11],\ \ [12]] \\ \mathit{ggroup}\ 2\ 3\ [0\,..\,12] & = & [[[0,1],[2,3],[4,5]],\ \ [[6,7],[8,9],[10,11]],\ \ [[12]]] \end{array}$$

Alleen het laatste element van een groep is misschien te kort.
Welke van deze twee beweringen is altijd waar:

$$\begin{array}{l} \mathit{concat}\ (\mathit{group}\ 3\ xs)\ =\ xs \\ \mathit{group}\ 3\ (\mathit{concat}\ xss)\ =\ xss \end{array}$$

53. Een *segment* van $[x_0, x_1, \ldots, x_{n-1}]$ is een aaneengesloten deel van de vorm: $[x_i, \ldots, x_{j-1}]$ met $0 \le i \le j \le n$. Definieer de functie *segs* met:

$$\mathit{segs}\ xs \quad = \quad \text{een lijst met precies alle segmenten van } xs$$

Wenk: gebruik de functies *inits*, *tails* en/of *segment* van opgaven 50, 51 en 25.
Toepassing: geef een uitdrukking voor het aantal voorkomens van *xs* in *ys*. Bijvoorbeeld, $[1,2,1]$ komt driemaal voor in $[3,1,2,1,2,1,7,1,2,1]$.

54. Definieer de functie segs' met $\mathit{segs}'\ n\ xs$ = een lijst van alle segmenten ter lengte *n* van *xs*.
Toepassing: geef net als in de vorige opgave een uitdrukking voor het aantal voorkomens van *xs* in *ys*.

55. De standaard functie *reverse* levert het argument omgedraaid op:

$$\mathit{reverse}\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_{n-1}, \ldots, x_1, x_0]$$

Definieer *reverse*.
Toepassing. Definieer *tl* en *tails* met behulp van *init*, *inits* en *reverse* (en zo nodig lijstcomprehensie).

56. Definieer de standaard functie *or* met *or* $[x_0, \ldots, x_{n-1}] = x_0 \vee \ldots \vee x_{n-1}$. Bij de lege lijst is de uitkomst *False*. Wenk: gebruik lijstcomprehensie en de test $\ldots = []$.
Toepassing: druk *member* uit met behulp van *or* (vergelijk met opgave 34).

57. Definieer de standaard functie *and* met *and* $[x_0, \ldots, x_{n-1}] = x_0 \wedge \ldots \wedge x_{n-1}$. Bij de lege lijst is de uitkomst *True*. Wenk: gebruik lijstcomprehensie en de test $\ldots = xs$, of beter: gebruik *or* en het feit dat $\neg(x \wedge y \wedge \ldots) = \neg x \vee \neg y \vee \ldots$.
Toepassing. Laat *allPos* de functie zijn met: *allPos xs* = 'alle elementen van *xs* zijn positief'. Definieer *allPos* met behulp van *and*.

58. De standaard functie *zip* maakt van een paar van lijsten een lijst van paren ('zip' is engels voor 'ritssluiting'):

$$\mathit{zip}\ ([x_0, \ldots, x_{m-1}],\ [y_0, \ldots, y_{n-1}]) \quad = \quad [(x_0, y_0),\ \ldots,\ (x_{k-1}, y_{k-1})]$$

waarbij k het minimum is van m en n. Definieer *zip*.

59. Functie *unzip* maakt van een lijst van paren een paar van lijsten:

$$\mathit{unzip}\ [(x_0, y_0),\ \ldots,\ (x_{n-1}, y_{n-1})] \quad = \quad ([x_0, \ldots, x_{n-1}],\ [y_0, \ldots, y_{n-1}])$$

Er geldt dus altijd *zip* (*unzip xys*) = *xys*, en voor evenlange lijsten *xs* en *ys* ook: *unzip* (*zip* (*xs*, *ys*)) = (*xs*, *ys*).

60. Definieer de functie *isStijgend* met:

$$\mathit{isStijgend}\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad x_0 \leq x_1 \leq \ldots \leq x_{n-1}$$

Probeer indicering ! te vermijden.

61. Een *palindroom* is een tekst die van links naar rechts dezelfde letterrij heeft als van rechts naar links. Bijvoorbeeld:

    Madam, I'm Adam
    'Mooie zeden in Ede', zei oom
    Nora bedroog, o zo goor, de baron
    Soms loopt Roos door goor, droog rood soort Pools mos

Definieer de functie *isPalindroom* die *True* of *False* oplevert al naar gelang het argument een palindroom is. Ga er in eerste instantie van uit dat het argument uitsluitend uit kleine letters bestaat; en probeer later rekening te houden met hoofdletters en leestekens, zoals in de voorbeelden hierboven.

62. Definieer de functies f en g met:

$$\begin{aligned} f\ xs &= \text{het aantal verschillende elt'en in } xs\text{, voor } \textit{stijgende } xs \\ g\ xs &= \text{het aantal verschillende elt'en in } xs\text{, voor willekeurige } xs \end{aligned}$$

Er worden geen eisen aan $f\ xs$ gesteld wanneer xs niet stijgend is. Probeer indicering ! te vermijden.

Wenk bij f: voor $\#xs \neq 1$ is $f\ xs$ gelijk aan het aantal verschillende buur-paren in de lijst; gebruik *zip* en *tl* om buur-paren te genereren.
Wenk bij g: gebruik *tails* (opgave 51), en kijk bij iedere tail van xs of de kop ervan voorkomt in de staart ervan.

Om te onthouden

- De volgende functies op lijsten zijn standaard aanwezig:

$$
\begin{array}{lll}
\#\ [x_0, x_1, \ldots, x_{n-1}] & = & n \\
take\ m\ [x_0, x_1, \ldots, x_{n-1}] & = & [x_0, x_1, \ldots, x_{k-1}] \qquad \text{met: } k = m\ \underline{min2}\ n \\
drop\ m\ [x_0, x_1, \ldots, x_{n-1}] & = & [x_k, x_{k+1}, \ldots, x_{n-1}] \qquad \text{met: } k = m\ \underline{min2}\ n \\
[x_0, \ldots, x_{n-1}]\ !\ i & = & x_i \qquad {\scriptstyle (mits\ \ 0 \leq i < n)} \\
member\ [x_0, \ldots, x_{n-1}]\ y & = & x_0 = y \vee \ldots \vee x_{n-1} = y \\
or\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0 \vee \ldots \vee x_{n-1} \\
and\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0 \wedge \ldots \wedge x_{n-1} \\
max\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0\ \underline{max2}\ \ldots\ \underline{max2}\ x_{n-1} \\
min\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0\ \underline{min2}\ \ldots\ \underline{min2}\ x_{n-1} \\
 & & \\
concat\ [xs_0, xs_1, \ldots, xs_{n-1}] & = & xs_0 ++ xs_1 ++ \cdots ++ xs_{n-1} \\
hd\ [x_0, \ldots, x_n] & = & x_0 \\
tl\ [x_0, \ldots, x_n] & = & [x_1, \ldots, x_{n-1}] \\
init\ [x_0, \ldots, x_n] & = & [x_0, \ldots, x_{n-1}] \\
last\ [x_0, \ldots, x_n] & = & x_{n-1} \\
reverse\ [x_0, x_1, \ldots, x_{n-1}] & = & [x_{n-1}, \ldots, x_1, x_0] \\
zip\ \ ([x_0, \ldots, x_{m-1}],\ [y_0, \ldots, y_{n-1}]) & & \\
 & = & [(x_0, y_0),\ \ldots,\ (x_{k-1}, y_{k-1})] \qquad \text{met: } k = m\ \underline{min2}\ n
\end{array}
$$

- De volgende functies zijn niet standaard aanwezig:

$$
\begin{array}{lll}
inits\ [x_0, \ldots, x_{n-1}] & = & [[],\ [x_0],\ [x_0, x_1],\ \ldots\ , [x_0, \ldots, x_{n-1}]] \\
tails\ [x_0, \ldots, x_{n-1}] & = & [[x_0, \ldots, x_{n-1}],\ [x_1, \ldots, x_{n-1}],\ \ldots\ , [x_{n-1}],\ []] \\
segment\ k\ l\ [x_0, \ldots, x_{n-1}] & = & [x_k, \ldots, x_{l-1}], \quad if\ 0 \leq k \leq l \leq n \\
segs\ [x_0, \ldots, x_{n-1}] & = & \text{een lijst met alle } [x_i, \ldots, x_{j-1}] \quad (0 \leq i \leq j \leq n) \\
segs'\ k\ [x_0, x_1, \ldots, x_{n-1}] & = & \text{een lijst met alle } [x_i, \ldots, x_{i+k-1}] \quad (0 \leq i \leq i{+}k \leq n) \\
unzip\ \ [(x_0, y_0),\ \ldots,\ (x_{k-1}, y_{k-1})] & & \\
 & = & ([x_0, \ldots, x_{k-1}],\ [y_0, \ldots, y_{k-1}])
\end{array}
$$

7 Strings en opmaak van uitvoer

*String*s zijn lijsten van karakters. Door uitkomsten eerst naar strings om te zetten en dan pas te tonen, is een nette opmaak van de afdruk eenvoudig te regelen.

§7.1 Strings. Een string is een lijst van karakters, bijvoorbeeld $['a', 'a', 'p']$. Voor dit soort lijsten is er in Miranda een verkorte notatie: "*aap*" is een notatie voor exact dezelfde karakterlijst als $['a', 'a', 'p']$. Dus "*aap*" ++ "*beer*" levert "*aapbeer*" $(= ['a', 'a', 'p', 'b', 'e', 'e', 'r'])$. Let op, vergis je niet:

a	is een variabele
$'a'$	is een karakter (de eerste letter van het alfabet)
"a"	is een string bestaande uit één karakter: "a" $=$ $['a']$

en

' '	is het spatie-karakter
" "	is de string bestaande uit één spatie: " " $=$ [' ']
""	is de lege string: "" $=$ []

Een paar karakters worden afwijkend genoteerd binnen karakter- en stringnotaties:

'\n'	"...\n..."	voor het nieuwe-regel karakter
'\b'	"...\b..."	voor het backspace karakter
'\''	"...\'..."	voor het enkele quote karakter
'\"'	"...\"..."	voor het dubbele quote karakter
'\\'	"...\\..."	voor het backslash karakter

§7.2 Opmaak. Strings spelen een grote rol in het leesbaar presenteren van invoer en uitvoer. Bij de "afdruk" van een string op het beeldscherm (of op file) worden de individuele karakters getoond; het spatiekarakter verschijnt als een spatie, en het newline karakter wordt zichtbaar als een regelovergang. Elke waarde die geen string is, wordt als een lange brei van karakters getoond: de Miranda notatie voor die waarde zonder enige spatie of regelovergang. Bijvoorbeeld (let ook op de spaties):

waarde	getoond op beeldscherm
$[[1, 2,\quad 3],\ [3, 4],\ [5, 6, 7]]$	`[[1,2,3],[3,4],[5,6,7]]`

"$[1, 2,\quad 3]\backslash n[3, 4]\backslash n[5, 6, 7]$"	`[1,2,    3]`
	`[3,4]`
	`[5,6,7]`

Hieronder staan een paar functies die handig zijn om een nette afdruk te krijgen. De functie *show* zet een waarde om in een string: de Miranda-notatie voor die waarde. Functie *lay* zet een lijst van strings om in één lange lijst met een regelovergang na elke string:

$$\begin{array}{ll} \mathit{show}\ 45 & = \texttt{"}45\texttt{"} \\ \mathit{show}\ \mathit{True} & = \texttt{"}\mathit{True}\texttt{"} \\ \mathit{show}\ [[1,\quad 2],\ [\,]\ \ ] & = \texttt{"}[[1,2],[\,]]\texttt{"} \\ & \\ \mathit{lay}\ [\texttt{"}abc\texttt{"},\texttt{"}de\texttt{"},\ldots] & = \texttt{"}abc\backslash nde\backslash n\ldots\texttt{"} \\ \mathit{layn}\ [\texttt{"}abc\texttt{"},\texttt{"}de\texttt{"},\ldots] & = \texttt{"}\ 1)\ abc\backslash n\ 2)\ de\backslash n\ 3)\ \ldots\texttt{"} \end{array}$$

Functie *layn* verschilt alleen van *lay* doordat-ie de regels van een rangnummer voorziet. Functie *show* is geen echte functie; er geldt een beperking die we later zullen bespreken (in §8.5). Bij moeilijkheden biedt het gebruik van *shownum* soms uitkomst; *shownum* werkt uitsluitend op getallen.

Terzijde. De inverse van *shownum* is *numval*:

$$\begin{array}{lcl} \mathit{shownum}\ 123 & = & \texttt{"}123\texttt{"} \\ \mathit{numval}\ \texttt{"}123\texttt{"} & = & 123 \end{array}$$

§7.3 Voorbeeld. Al eerder hebben we de functie *opln* besproken:

$$\mathit{opln}\ n\ (a,b,c)\ =\ [(x,y,z)\mid x,y,z\leftarrow[1\mathinner{..}n];\ x{-}y=a;\ z{+}x=b;\ y{\times}z=c]$$

De “afdruk” van *opln* $32\ (6,17,18)$ op het beeldscherm ziet er weinig apetijtelijk uit: een lange brei van karakters. De uitkomst wordt mooier getoond door:

$$\mathit{layn}\ [\mathit{show}\ \mathit{opl}\mid \mathit{opl}\leftarrow \mathit{opln}\ 32\ (6,17,18)]$$

Hier wordt van iedere oplossing *opl* een string gemaakt door *show*, en de lijst van strings wordt door *layn* tot een lange lijst gemaakt (met newline karakters tussen ieder tweetal oplossingen). We kunnen deze tabel van een kop voorzien:

$$\texttt{"}\mathit{Alle\ oplossingen}:\backslash n\backslash n\texttt{"}\ +\!\!+\ \mathit{layn}\ [\mathit{show}\ \mathit{opl}\mid \mathit{opl}\leftarrow \mathit{opln}\ 32\ (6,17,18)]$$

Nog mooier is het om de argumenten 32 en (6,17,18) ook in de kop op te nemen:

$$\begin{array}{l} \texttt{"}\mathit{Alle\ oplossingen\ bij}\ \texttt{"}\ +\!\!+\ \mathit{show}\ (32,(6,17,18))\ +\!\!+\ \texttt{"}:\backslash n\backslash n\texttt{"}\ +\!\!+ \\ \mathit{layn}\ [\mathit{show}\ \mathit{opl}\mid \mathit{opl}\leftarrow \mathit{opln}\ 32\ (6,17,18)] \end{array}$$

Het ligt voor de hand om dit tot een functie te veralgemenen:

$$\begin{array}{l} \mathit{showOpln}\ n\ (a,b,c) \\ =\ \texttt{"}\mathit{Alle\ oplossingen\ bij}\ \texttt{"}\ +\!\!+\ \mathit{show}\ (n,(a,b,c))\ +\!\!+\ \texttt{"}:\backslash n\backslash n\texttt{"}\ +\!\!+ \\ \quad \mathit{layn}\ [\mathit{show}\ \mathit{opl}\mid \mathit{opl}\leftarrow \mathit{opln}\ n\ (a,b,c)] \end{array}$$

We kunnen deze definitie ook noteren met één variabele *abc* voor het drietal (a,b,c):

$$\mathit{showOpln}\ n\ \mathit{abc}$$

$= \texttt{"}\textit{Alle oplossingen bij }\texttt{"} +\!\!+ \; \textit{show}\ (n, abc) +\!\!+ \texttt{"} : \backslash n \backslash n\texttt{"} +\!\!+$
$\quad \textit{layn}\ [\textit{show opl} \mid \textit{opl} \leftarrow \textit{opln}\ n\ abc]$

Een aanroep van deze functie is bijvoorbeeld: *showOpln* 32 (6, 17, 18).

§7.4 Uitlijnen. Om een collectie van waarden netjes te tabelleren, moeten we strings aanvullen met spaties tot een opgegeven lengte. Daarvoor zijn er de volgende standaard functies:

ljustify n s	=	string *s* rechts aangevuld met spaties tot totale lengte *n*
rjustify n s	=	string *s* links aangevuld met spaties tot totale lengte *n*
cjustify n s	=	string *s* links en rechts aangevuld tot totale lengte *n*
spaces n	=	een string van precies *n* spaties

In Miranda kunnen we ze als volgt definiëren:

$$\begin{array}{l} \textit{spaces}\ n \ = \ [\text{' '} \mid i \leftarrow [0\,..\,n{-}1]] \\ \textit{ljustify}\ n\ s \\ \ = \ s +\!\!+ \textit{spaces}\ (n{-}\#s), \ \textit{if}\ n \ \geq \ \#s \\ \ = \ s, \ \textit{otherwise} \end{array}$$

En net zo voor *rjustify* en *cjustify*. Met de standaard functie *max*, die het maximum van een lijst oplevert, kan het nog eenvoudiger:

$$\begin{array}{l} \textit{ljustify}\ n\ s \ = \ s +\!\!+ \textit{spaces}\ (\textit{max}\ [0,\ n{-}\#s]) \\ \textit{rjustify}\ n\ s \ = \ \textit{spaces}\ (\textit{max}\ [0,\ n{-}\#s]) +\!\!+ s \\ \textit{cjustify}\ n\ s \ = \ \textit{ljustify}\ n\ (\textit{rjustify}\ k\ s)\ \textit{where}\ k \ = \ n - (n{-}\#s)\ \textit{div}\ 2 \end{array}$$

Bij *cjustify n s* is het misschien niet direct duidelijk hoeveel spaties er exact staan aan weerszijden van *s*, maar het is wel duidelijk dat de totale lengte tot *n* is aangevuld, en dat *s* er in staat.

Met deze hulpmiddelen kunnen we alle oplossingen van de puzzel netjes tabelleren. Voor ieder getal in een oplossing (x, y, z) reserveren we vier posities:

$$\begin{array}{l} \textit{showOpl}\ (x, y, z) \ = \ \textit{show}'\ x +\!\!+ \textit{show}'\ y +\!\!+ \textit{show}'\ z \\ \textit{show}'\ n \ = \ \textit{rjustify}\ 4\ (\textit{shownum}\ n) \end{array}$$

In functie *showOpln* wijzigen we het deel '*show opl*' in '*showOpl opl*':

$$\textit{showOpln}\ n\ abc \ = \ \ldots \ \textit{layn}\ [\textit{showOpl opl} \mid \textit{opl} \leftarrow \textit{opln}\ n\ abc]$$

Opgaven

63. Definieer functies waarvan de uitkomst op het beeldscherm er als volgt uit ziet:

verti xs	:	de karakters van *xs* verticaal onder elkaar
sqHor xs	:	#*xs* maal *xs* horizontaal onder elkaar: een vierkant

sqVer xs	:	*#xs* maal *xs* verticaal naast elkaar: een vierkant
diaNW xs	:	de karakters van *xs* schuin onder elkaar: van NW naar ZO
diaNO xs	:	de karakters van *xs* schuin onder elkaar: van NO naar ZW
triaNW xs	:	de NW boven-driehoek van *sqHor xs*
triaNO xs	:	de NO boven-driehoek van *sqHor xs*
triaZW xs	:	de ZW onder-driehoek van *sqHor xs*
triaZO xs	:	de ZO onder-driehoek van *sqHor xs*

64. Geef een uitdrukking die op het beeldscherm het volgende toont:

```
8 *  1 =  8
8 *  2 = 16
...
8 * 10 = 80
```

Maak het nog mooier door er een kopje boven te zetten (waarin de 8 genoemd wordt). Veralgemeen de uitdrukking tot een functie door 8 tot een parameter te maken.

65. Definieer de functie *histogram* met als uitkomst op $[x_0, \ldots, x_{n-1}]$:

n regels met op regel i een visuele representatie van x_i

Bijvoorbeeld, de uitkomst bij argument $[3, 17, 2, 5]$ ziet er uit als:

```
0:  3 | ###
1: 17 | #################
2:  2 | ##
3:  5 | #####
```

Doe het zo nodig eerst zonder de regelnummers.

66. Een lijst van voetbaluitslagen ziet er als volgt uit:

[... (("*Ajax*", "*Feyenoord*"), 0, 5), (("*FC Twente*", "*Roda*"), 3, 11) ...]

Geef een functie *showUitslagen* die, toegepast op zo'n lijst, een nette tabel toont:

```
------------------------------------
| ...                     |         |
| Ajax       - Feyenoord |  0 -  5 |
| FC Twente  - Roda      |  3 - 11 |
| ...                     |         |
| ...                     |         |
------------------------------------
```

Zorg ervoor dat de eenheden van de doelpunten recht onder elkaar staan.

67. Net als in de vorige opgave, maar nu krijgt de functie een extra parameter, zoals "*zondag* 21 *mei*", en komt er de volgende kop boven de tabel:

```
Uitslagen van zondag 21 mei:
------------------------------------
| ...                     |         |
```

68. Definieer de functies *chain* en *train* met:

$$\begin{array}{lcl} \mathit{chain}\ \mathit{sep}\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0 \mathbin{+\!\!+} \mathit{sep} \mathbin{+\!\!+} x_1 \mathbin{+\!\!+} \mathit{sep} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} x_{n-1} \\ \mathit{train}\ \mathit{sep}\ [x_0, x_1, \ldots, x_{n-1}] & = & x_0 \mathbin{+\!\!+} \mathit{sep} \mathbin{+\!\!+} x_1 \mathbin{+\!\!+} \mathit{sep} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} x_{n-1} \mathbin{+\!\!+} \mathit{sep} \end{array}$$

Wenk: laat de uitkomst van *chain* in eerste instantie één *sep* te veel hebben, en laat die dan weg met behulp van *drop* of zo. (De naamgeving '*sep*' komt van separator, scheider.)

69. Definieer met behulp van *chain* en *train* uit opgave 68 de drie functies *zin*, *zinnen* en *zinnen′* die een woordenlijst, respectievelijk een lijst van woordenlijsten, aaneen rijgen op de volgende manier. Laat:

$$\begin{array}{lcl} \mathit{xs} & = & [\texttt{"}\mathit{Daar}\texttt{"},\ \texttt{"}\mathit{loopt}\texttt{"},\ \texttt{"}\mathit{Jan}\texttt{"}] \\ \mathit{ys} & = & [\texttt{"}\mathit{Jan}\texttt{"},\ \texttt{"}\mathit{gaat}\texttt{"},\ \texttt{"}\mathit{naar}\texttt{"},\ \texttt{"}\mathit{huis}\texttt{"}] \\ \mathit{zss} & = & [\mathit{xs},\ \mathit{ys}] \end{array}$$

Dan wordt getoond op het beeldscherm:

```
zin xs       :  Daar loopt Jan
zinnen zss   :  Daar loopt Jan.
                Jan gaat naar huis.

zinnen' zss  :  Daar loopt Jan; Jan gaat naar huis.
```

Wat is de uitkomst van *zinnen xs*? (Eerst nadenken, dan pas proberen.)

70. Herinner je het Oranje-bestand van §6.1: *mannen*, *vrouwen*, *kinderen*. Geef een uitdrukking die het hele bestand netjes afdrukt: steeds de volgende persoon op een nieuwe regel, direct gevolgd door al zijn kinderen op één regel, elk daarvan op precies 20 posities.
Doe het ook eens zó: steeds de volgende persoon op een nieuwe regel, direct gevolgd door al zijn kinderen in een kolom onder elkaar (rechts aangelijnd).

71. De gegevens voor een kalender zijn per maand als volgt beschikbaar:

$$\begin{array}{lcl} \mathit{jan} & = & (6,\ 31) \\ \mathit{feb} & = & (2,\ 28) \\ \ldots & & \\ \mathit{week} & = & [\texttt{"}\mathit{ma}\texttt{"},\ \texttt{"}\mathit{di}\texttt{"},\ \texttt{"}\mathit{wo}\texttt{"},\ \texttt{"}\mathit{do}\texttt{"},\ \texttt{"}\mathit{vr}\texttt{"},\ \texttt{"}\mathit{za}\texttt{"},\ \texttt{"}\mathit{zo}\texttt{"}] \end{array}$$

De getallen 6 en 31 bij *jan* geven aan dat januari begint op de zevende dag van *week* (6 dagen overslaan, dus) en 31 dagen heeft. Dus maand *feb* begint op woensdag en heeft 28 dagen. Net zo bij andere maanden. Definieer een functie *showMaand* die, gegeven zo'n maand, een nette afdruk ervan oplevert; bijvoorbeeld, voor *jan*:

```
ma di wo do vr za zo
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Wenk. Voor de maand $jan = (6, 31)$ bestaat de tabel in essentie uit:

$$take\ 42\ (take\ 6\ nullen\ +\!\!+\ [1\,..\,31]\ +\!\!+\ nullen)$$

waarbij $nullen\ =\ [0, 0\,..]$. Dit geeft precies bovenstaande 7×6 getallen (waarbij een nul als spatie wordt geïnterpreteerd) in één lijst. Gebruik zo nodig functies *group* en *ggroup* van opgave 52.

Om te onthouden

• Een lijst van strings kan mooi worden afgedrukt met *lay* en *layn*; zij verwachten een lijst van strings als argument. Functie *show* maakt van willekeurige waarde een string: de Miranda-notatie ervan. Functies *ljustify* etc kunnen gebruikt worden voor uitlijnen.

• Neem lay-out voorzieningen niet op in functies waarvan je de resultaten nog in verdere berekeningen wil gebruiken.

• 'Lelijk' gevalsonderscheid kan soms voorkomen worden door gebruik te maken van *take* en *max*; daar zit al een gevalsonderscheid in ingebouwd.

8 Typering

Met *typering* wordt van een uitdrukking (of naam) het soort waarde vastgelegd. Hiermee zijn sommige tik- en denkfouten al te ontdekken nog voordat het programma in gebruik genomen wordt.

§8.1 Nut van typen. Een grote klasse van fouten in uitdrukkingen kan Miranda al achterhalen aan de hand van de tekst, nog voordat het programma in gebruik wordt genomen. Bijvoorbeeld, $\ldots 2+(\mathit{True}\times 4)\ldots$ is fout: de waarde *True* en de operatie $\times$ passen niet bij elkaar. Miranda zal deze uitdrukking niet accepteren (en daarmee een tikfout of denkfout ontdekken).

§8.2 Voorbeelden. De manier waarop Miranda te werk gaat is eenvoudig: bij iedere definitie wordt het 'type' van de gedefinieerde naam bepaald, en bij ieder gebruik ervan wordt getest of het gebruik in overeenstemming is met het type. In feite probeert Miranda *iedere* deeluitdrukking een type toe te wijzen. In het type van een naam of uitdrukking wordt vastgelegd of de waarde ervan een getal zal zijn (type *num*) of een waarheidswaarde (type *bool*) of een karakter (type *char*), en hoe de structuur van de waarde is: een lijst $[\ldots]$ of tupel $(\ldots)$ of functie $\ldots \to \ldots$. De naam '*bool*' komt van de logicus George Boole; waarheidswaarden worden voortaan ook wel *boolse* of *boolean* waarden genoemd. Hier zijn wat voorbeelden van type-specificaties (de :: betekent 'is van type'):

3	::	num
$3+4$	::	num
$3<4$	::	bool
$[3+4,\ 5\times 7]$	::	$[\mathit{num}]$
$[3+4,\ \mathit{True}]$		niet typeerbaar: fout
$(3+4,\ \mathit{True})$	::	$(\mathit{num},\ \mathit{bool})$
$['a',\ 'b',\ 'c']$	::	$[\mathit{char}]$
sqrt	::	$\mathit{num} \to \mathit{num}$
$\mathit{sqrt}\ (3+4)$	::	num
$\mathit{sqrt}\ '3'$		niet typeerbaar: fout
code	::	$\mathit{char} \to \mathit{num}$
decode	::	$\mathit{num} \to \mathit{char}$
$\mathit{decode}\ (32+2)$	::	char
$\mathit{code}\ (\mathit{decode}\ (32+2))$	::	num
$\mathit{code}\ (\mathit{decode}\ 'a')$		niet typeerbaar: fout

Het volgende voorbeeld laat zien hoe het voor functies met verscheidene parameters gaat. Wanneer $\mathit{wortel}1$ en $\mathit{wortel}1'$ als volgt zijn gedefinieerd:

$$\mathit{wortel}1\ a\ b\ c\ =\ (b - \mathit{sqrt}\ (b\hat{\ }2 - 4\times a\times c))\ /\ (2\times a)$$
$$\mathit{wortel}1'\ (a,\ b,\ c)\ =\ (b - \mathit{sqrt}\ (b\hat{\ }2 - 4\times a\times c))\ /\ (2\times a)$$

dan geldt:

$$\begin{array}{lll} wortel1 & :: & num \to num \to num \to num \\ wortel1' & :: & (num,\ num,\ num) \to num \end{array}$$

en dus:

$$\begin{array}{lll} wortel1\ 10\ 20\ (10{+}20) & :: & num \\ wortel1'\ (10,\ 20,\ 10{+}20) & :: & num \end{array}$$

en ook:

$$\begin{array}{ll} wortel1\ (10,\ 20,\ 10{+}20) & \text{niet typeerbaar: fout} \\ wortel1\ 10\ (decode\ 32)\ (10{+}20) & \text{niet typeerbaar: fout} \end{array}$$

Om operatoren te typeren maken we er eerst een functie van (door er haakjes omheen te zetten):

$$\begin{array}{lll} (+),\ (\times),\ (/),\ (div) & :: & num \to num \to num \\ (\wedge),\ (\vee) & :: & bool \to bool \to bool \end{array}$$

In iedere type-uitdrukking staan α en β etc voor ‘een willekeurig type’:

$$\begin{array}{lll} [\,] & :: & [\alpha] \\ (\#) & :: & [\alpha] \to num \\ (=),\ (\neq) & :: & \alpha \to \alpha \to bool \\ fst & :: & (\alpha,\ \beta) \to \alpha \\ snd & :: & (\alpha,\ \beta) \to \beta \end{array}$$

Op grond van deze typering geldt nu onder andere:

$$\begin{array}{lll} [\,] & :: & [num] \\ [\,] & :: & [(char, \alpha, [\beta])] \end{array}$$

en

$$\begin{array}{lll} \#\ [1,2,3] & :: & num \\ \#\ [('a', [1,2,3]),\ ('b', [4,5,6])] & :: & num \\ fst\ (1,\ 'a') & :: & num \\ snd\ (1,\ 'a') & :: & char \end{array}$$

Als er in het type van een uitdrukking een α staat (of β etc), dan heet die uitdrukking én zijn waarde: *polymorf* (‘veel-vormig’), en anders *monomorf* (‘eenvormig’).

§8.3 Type-specificaties. In een script mag van iedere naam die gedefinieerd is op globaal niveau (dus niet: binnen een where-part) het beoogde type ergens worden gespecificeerd. Bijvoorbeeld:

$$\begin{array}{l} \mathit{wortel}1\ ::\ \mathit{num} \rightarrow \mathit{num} \rightarrow \mathit{num} \rightarrow \mathit{num} \\ \mathit{wortel}1\ a\ b\ c\ =\ (b - \mathit{sqrt}\ (b\hat{\ }2 - 4{\times}a{\times}c))\ /\ (2{\times}a) \end{array}$$

Het voordeel hiervan is tweevoudig: Miranda kan eventuele type-fouten (soms tikfouten, soms denkfouten) beter localiseren, en de lezer van de tekst heeft soms grote hulp van de type-specificaties bij het begrijpen van de tekst. Dit laatste is nauwelijks het geval bij de definitie van *wortel*1, maar is wel het geval bij grote, ingewikkelde definities. Goed gekozen commentaar maakt een programmatekst natuurlijk nog leesbaarder, maar van een type-specificatie weet je tenminste dat die door Miranda gecontroleerd en in orde bevonden is.

§8.4 Type-synoniemen. Om typen wat korter te noteren, en met wat suggestievere namen, mag je *type-synoniemen* gebruiken. Bijvoorbeeld, na:

$$\mathit{datum}\ \equiv\ (\mathit{num},\ \mathit{num},\ \mathit{num})$$

zijn *datum* en $(\mathit{num}, \mathit{num}, \mathit{num})$ volkomen uitwisselbaar, en zou je kunnen gaan specificeren:

$$\begin{array}{l} \mathit{maand}\ ::\ \mathit{datum} \rightarrow [\mathit{char}] \\ \mathit{weeknr}\ ::\ \mathit{datum} \rightarrow \mathit{num} \\ \mathit{jaarLater}\ ::\ \mathit{datum} \rightarrow \mathit{datum} \end{array}$$

Miranda zal bij type-controle altijd alle type-synoniemen helemaal uitschrijven.

Type-synoniemen mogen ook parameters α, β, ... hebben.

§8.5 Show. In §7.2 hebben we *show* besproken en een beperking op het gebruik ervan aangekondigd. Die beperking kunnen we nu precies maken. Weliswaar is de speciale functie *show* polymorf, met type $\alpha{\rightarrow}[\mathit{char}]$, maar elk voorkomen ervan moet monomorf gebruikt worden. Dus *show expr* is alleen maar geoorloofd als (uit de context of uit de expressie zelf) blijkt dat *expr* een monomorf type heeft. Door zelf voldoende veel type-specificaties toe te voegen verdwijnen de meeste problemen.

Opgaven

72. Geef het type van de waarden en functies die voldoen aan de volgende specificaties:

pi	=	π, de verhouding tussen omtrek en diameter van een cirkel
sqrt pi	=	$\sqrt{\pi}$
take	=	de functie met *take n xs* = de lijst van de eerste *n* elementen van *xs*
member	=	de functie met *member xs x* = de lijst *xs* bevat *x*
oplLijst	=	de lijst van drietallen getallen die voldoen aan ...
opln	=	de fct met *opln n abc* = een lijst van drietallen (x, y, z) waarvoor geldt ...
puzzels	=	de fct met *puzzels k* = een lijst van tupels (n, abc) met $\#\mathit{opln}\ n\ \mathit{abc}\ =\ 2$

Functies *opln* en *puzzels* zijn degenen die al eerder zijn besproken (in §5.2).

73. Geef de typen van functies en lijsten die bij het Huis van Oranje gedefinieerd zijn: *mannen*, *kinderen*, *vaders*, *broersEnZussen*, *isBroerOfZusVan*.

74. Geef de typen van de functies *show*, *shownum*, *lay*, *layn* en *error*.
75. Studentgegevens bestaan, per student, uit: de naam, het geboortejaar, de naam van de faculteit, en het aantal behaalde studiepunten (SP). Geef het type voor de studentgegevens van één student, en laat *student* een type-synoniem zijn hiervoor.
Een studentenbestand wordt gerepresenteerd door een lijst van studentgegevens. Geef het type van een studentenbestand, en definieer *bestand* als synoniem hiervoor.
Geef nu de typen (niet de definities) voor de functies met de volgende specificaties:

studenten b	=	een lijst van alle studentnamen in bestand *b*
gemSP b	=	het aantal SP dat de studenten uit *b* gemiddeld behaald hebben
faculteiten b	=	de faculteiten die voorkomen in *b*, op alfabetische volgorde

76. Een bestand is een lijst van patiëntengegevens. De gegevens van één patiënt zijn: de naam, het geslacht, het geboortejaar en de lijst van afdelingen-met-jaartallen van opnamen van de patiënt. Geef het type van zo'n bestand; gebruik eventueel zinvolle type-synoniemen.
Geef van de volgende functies de type-specificatie:

namen b	=	een lijst van alle patiëntnamen in bestand *b*
namenVr b	=	een lijst van alle namen in *b* van vrouwelijke patiënten
opnamesVoor b j	=	een lijst van alle (nm, jr) met (volgens bestand *b*):
		patiënt *nm* is geboren in jaar *jr* en opgenomen vóór jaar *j*

77. Geef de type-specificaties van de functies uit het overzicht in §6.2: $\#$, *take*, *drop*, !, *or*, *and*, *min*, *max*, *member*, *concat*, *hd*, *last*, *init*, *tl*, *inits*, *tails*, *segment*, *segs*, *segs'*, *reverse*, *zip*. Bedenk dat er in een specificatie verscheidene namen voor de :: mogen staan.
78. Een partitie van een lijst *xs* is een rij van delen van *xs* die, achter elkaargezet, tesamen geheel *xs* zijn. Bijvoorbeeld, $[[a, b, c], [d, e], [f, g, h]]$ is een partitie van $[a, b, c, d, e, f, g, h]$. Geef het type van een partitie van een lijst van getallen, een lijst van karakters, en een lijst van getallijsten. Geef het type van de functie *partitions*, gespecificeerd door:

$$\mathit{partitions}\ \mathit{xs} \quad = \quad \text{een lijst van alle partities van } \mathit{xs}$$

Laat t het type zijn van de elementen van *xs*; dus $\mathit{xs} :: [t]$. Onder welke voorwaarde op t is $[\mathit{layn}\ \mathit{xss} \mid \mathit{xss} \leftarrow \mathit{partitions}\ \mathit{xs}]$ typeerbaar, en wat is dan het type? Is $\mathit{layn}\ (\mathit{partitions}\ \mathit{xss})$ typeerbaar?

Om te onthouden

• Denk- en tikfouten manifesteren zich vaak als type-fouten. Door de aanwezigheid van type-specificaties kan Miranda eventuele type-fouten beter localiseren.

• Type-specificaties kunnen, net als goede uitleg, behulpzaam zijn bij het begrijpen van Miranda-definities.

• Op het commando '*expression* ::' toont de interpretator het type van *expression*.

9 Patronen

Een *patroon* is een gedeeltelijke beschrijving van een waarde, met name van de vorm en constante delen ervan. Patronen worden gebruikt daar waar nieuwe namen geïntroduceerd worden. Met patronen kunnen de onderdelen van een samengestelde waarde geselecteerd en benoemd worden.

§9.1 De prefix- of cons-operatie. Operatie $:$ is als volgt gedefinieerd:

$$x \,:\, xs \quad = \quad [x] \mathbin{+\!\!+} xs$$

Dus $1 : [2,3]$ levert $[1] \mathbin{+\!\!+} [2,3]$, dat is $[1,2,3]$. De voorrangregels zijn zó dat, onder andere:

$$\begin{array}{llll} x \,:\, y \,:\, zs & \text{staat voor} & x : \,(y : \,zs) & \text{en niet voor } (x : y) \,:\, zs \\ x + y \,:\, zs & \text{staat voor} & (x{+}y) : \,zs & \text{en niet voor } x + (y : zs) \end{array}$$

Operatie $:$ speelt een speciale rol, omdat Miranda intern iedere lijst $[x_0, x_1, \ldots, x_{n-1}]$ opslaat in de vorm $x_0 : x_1 : \ldots : x_{n-1} : []$, dat wil zeggen $x_0 : (x_1 : (\ldots : (x_{n-1} : [])))$. Operatie $:$ heet 'cons' (van: lijst*cons*tructor) en ook wel 'prefix'.

Let er op dat $[1,2] : 3$ niet zinvol is; operatie $:$ verwacht een element links en een lijst rechts. Er geldt wel dat $[1,2,3]$ te schrijven is als $[1,2] \mathbin{+\!\!+} [3]$.

§9.2 Patronen in functiedefinities. In Miranda mogen parameters van functiedefinities een *vorm* ofwel *patroon* voorschrijven, waarin het argument moet *passen*. Daarmee is gevalsonderscheid soms elegant uit te drukken, en zijn de onderdelen van een argument direct te benoemen. Bijvoorbeeld:

$$\begin{array}{l} \mathit{is0koppig}\ [] \ = \ \mathit{False} \\ \mathit{is0koppig}\ (x : xs) \ = \ x = 0 \end{array}$$

Dus $\mathit{is0koppig}\ [0,1,2]$ levert True, omdat het argument $[0,1,2]$ niet past in het eerste patroon $[]$ en wel past in het volgende patroon $x : xs$ (met $x = 0$ en $xs = [1,2]$) en de uitdrukking $x = 0$ dan True levert.

Een alternatieve definitie voor $\mathit{is0koppig}$ luidt:

$$\begin{array}{l} \mathit{is0koppig}\ (0 : xs) \ = \ \mathit{True} \\ \mathit{is0koppig}\ xs \ = \ \mathit{False} \end{array}$$

Dus, volgens de laatste clausule is $\mathit{is0koppig}\ xs$ altijd False, en de eerste clausule geeft daarop een uitzondering. De volgorde van de clausules is belangrijk: $[0,1,2]$ past in beide patronen; Miranda 'probeert' de alternatieven in de opgeschreven volgorde, en levert dus True. De uitkomst van $\mathit{is0koppig}\ [1,2,3]$ is False want $[1,2,3]$ past niet in het eerste patroon $0 : xs$ en wel in het tweede, xs.

Ook voor de natuurlijke getallen $0, 1, 2 \ldots$ zijn er patronen:

$$\mathit{voorganger}\ 0 \ = \ 0$$

$$\textit{voorganger}\ (n{+}1)\ =\ n$$

Dus *voorganger* $3 = 2$, want 3 is niet te schrijven als 0 en wel als $n{+}1$. En *voorganger* (-3) resulteert in een foutstop, want -3 is niet te schrijven als 0 of $n{+}1$ met *een natuurlijk getal* n.

Patronen voor tupels liggen voor de hand en zijn we al tegen gekomen, zoals:

$$\textit{fst3}\ (x, y, z)\ =\ x$$

§9.3 Algemener. Hier zijn nog meer voorbeelden van patronen:

$$n{+}2 \qquad x : y : zs \qquad (x : xs) : xss \qquad [x, y] \qquad ([x, 3, 5, y] : xss) : [[3]] : xsss$$

Algemener, patronen zijn opgebouwd uit variabelen en:

willekeurige constanten, zoals 0, 1, $'a'$, *True*
de vormen +1, +2, +3, etcetera
de vormen : en [, , ,]
de vormen (, , ,).

Operatie $+\!\!+$ is niet toegestaan in een patroon, omdat een argument in het algemeen op verschillende manieren met $+\!\!+$ te schrijven is, en je met zo'n patroon dus verschillende uitkomsten zou kunnen krijgen. Let er op dat '$f\ n{+}1$' staat voor '$(f\ n){+}1$'; net zo staat '$f\ x : xs$' voor '$(f\ x) : xs$', want, zoals altijd, *een functie–argument binding heeft voorrang boven een operator–operand binding.* Zet dus zo nodig haakjes om een patroon.

§9.4 Patronen elders. Niet alleen worden patronen gebruikt in functiedefinities, ze worden ook gebruikt in comprehensies:

$$[\ldots \mid \ldots\ \textit{patroon} \leftarrow \textit{lijst}\ \ldots]$$

Dit komt veel voor als de elementen van *lijst* paren zijn:

$$[\ldots x \ldots y \ldots \mid \ldots\ (x, y) \leftarrow \textit{zip}\ (\ldots\ ,\ \ldots)\ \ldots]$$

Patronen in een comprehensie werken als een filter; in de uitdrukking:

$$[\ldots x \ldots y \ldots zs \ldots \mid \ldots\ (x : y : zs) \leftarrow \textit{lijst}\ \ldots]$$

doen uitsluitend de *lijst*-elementen van de vorm $(x : y : zs)$ mee in de vorming van de resultaatlijst; dus *lijst*-elementen van de vorm [] en $[x]$ doen niet mee. Bijvoorbeeld,

$$\#\,[1 \mid (x : y : zs) \leftarrow [[],\ [a],\ [b, c],\ [d, e, f]]\,] \quad = \ 2$$

Patronen mogen ook gebruikt worden als linkerlid van definities. Bijvoorbeeld, het volgende is toegestaan als definitie van xs, x, y, z:

$$(x : xs) \ = \ \mathit{lijst}$$
$$(y,\ z) \ = \ \mathit{tupel}$$

In de context van deze definitie duidt x het eerste element aan van *lijst* en xs de rest; y en z duiden het linker en rechterlid van *tupel* aan.

Opgaven

79. Gebruik patronen om *letterNa* van opgave 17 te definiëren zonder '*if*' en '*otherwise*'.
80. Definieer zonder '*if*' en '*otherwise*' functie f met:

$$f\ n \quad = \quad 91 \text{ als } n \leq 100 \text{ en anders } n{-}10$$

81. Geef met behulp van patronen een definitie van de standaard functies *hd* (head) en *tl* (tail):

$$\mathit{hd}\ [x_0, \ldots, x_n] \quad = \quad x_0$$
$$\mathit{tl}\ [x_0, \ldots, x_n] \quad = \quad [x_1, \ldots x_n]$$

waarbij $n \geq 0$. Voor de lege lijst zijn *hd* en *tl* niet gedefinieerd.

82. Definieer de functie *xor* (exclusive or) met behulp van patronen. Er geldt: x $\underline{\mathit{xor}}$ y is *True* precies wanneer x óf y, en niet beide, *True* is.

Om te onthouden

- Met patronen kunnen de onderdelen van een argument benoemd worden.
- Met patronen kan gevalsonderscheid soms elegant opgeschreven worden.
- Een patroon in een generator van een lijstcomprehensie werkt als een filter.
- Schrijf zo nodig haakjes om een patroon: $f\ n{+}1$ betekent $(f\ n){+}1$ en dat is wat anders dan $f\ (n{+}1)$. Net zo: $f\ x{:}xs$ betekent $(f\ x){:}xs$ en niet $f\ (x{:}xs)$.

10 Recursie

Definities, met name functiedefinities, mogen *recursief* zijn: de gedefinieerde naam komt voor in de definiërende uitdrukking. Hiermee zijn in principe alle standaard functies te definiëren.

§10.1 Recursieve definities. In Miranda mag het rechterlid van een definitie de gedefinieerde naam bevatten. We spreken dan van een recursieve definitie; 'recursie' betekent 'terugkomen'. Vooral functies worden vaak met recursie gedefinieerd; recursief gedefinieerde lijsten komen later aan bod. Hier is een volstrekt nutteloos maar wel illustratief voorbeeld:

$$
\begin{array}{l}
f\ n \\
\ = \ f\ (f\ (n{+}11)),\ \ \mathit{if}\ n \leq 100 \\
\ = \ n{-}10,\ \ \mathit{otherwise}
\end{array}
$$

Aan de hand van deze gelijkheden kunnen we f 98 als volgt uitrekenen:

$$
\begin{array}{ll}
 & f\ 98 \\
= & \\
 & f\ (f\ 109) \\
= & \\
 & f\ 99 \\
= & \\
 & f\ (f\ 110) \\
= & \\
 & f\ 100 \\
= & \\
 & f\ (f\ 111) \\
= & \\
 & f\ 101 \\
= & \\
 & 91
\end{array}
$$

Als je op grond van de vergelijkingen in een definitie een uitkomst kúnt berekenen, dan zál Miranda diezelfde uitkomst ook produceren (maar misschien met een andere volgorde van de rekenstappen). Tussen haakjes, voor alle $n \leq 100$ levert $f\ n$ uiteindelijk 91 als uitkomst.

Door 'recursie' kan de berekening soms oneindig doorlopen, zonder dat er een antwoord wordt geproduceerd. Dat is het geval bij de volgende twee functies:

$$
\begin{array}{l}
f\ n \ = \ f\ (f\ (n{+}11)) \\
\mathit{droste}\ x \ = \ 2 \times \mathit{droste}\ (x/2)
\end{array}
$$

Meestal heeft een recursieve definitie daarom tenminste een clausule waarin geen 'recursie' optreedt.

§10.2 Inductie. We zeggen dat een functie f gedefinieerd is '*met inductie naar een opbouw van het argument*' als de uitkomst bij een "opgebouwd" argument wordt uitgedrukt in de uitkomsten bij de onderdelen van het argument. Dit is een eenvoudige vorm van recursie. We geven hiervan een paar voorbeelden.

De *faculteit* (engels: *factorial* en niet faculty) is de functie *fac* met:

$$fac\ n \ = \ n \times (n{-}1) \times \ \ldots \ \times 1 \qquad\qquad en: \qquad fac\ 0 \ = \ 1$$

Een recursieve definitie luidt als volgt:

$$\begin{aligned} &fac\ 0 \ = \ 1 \\ &fac\ (n{+}1) \ = \ (n{+}1) \times fac\ n \end{aligned}$$

Aangezien ieder natuurlijk getal is opgebouwd door middel van 0 en +1, en hier *fac* $(n{+}1)$ wordt uitgedrukt in *fac* n, noemen we deze definitie 'met inductie naar de opbouw van het argument'.

De uitkomst van *fac* is een getal; een lijst als uitkomst van een recursief gedefinieerde functie kan ook:

$$\begin{aligned} &terug\ 0 \ = \ [] \\ &terug\ (n{+}1) \ = \ n{+}1 : \ terug\ n \end{aligned}$$

Er geldt dan voor alle n dat *terug* $n \ = \ [n,\ n{-}1 \ ..\ 1]$. Functie *terug* is gedefinieerd met inductie naar de opbouw van het argument.

Nog een voorbeeld: de standaard functie *sum* met

$$sum\ [x_0, x_1, \ldots, x_{n-1}] = x_0{+}x_1{+}\cdots{+}x_{n-1} \qquad\qquad en: \ sum\ [\,] \ = \ 0$$

Hieronder volgen drie recursieve definities van *sum*. *Het zijn correcte definities, maar ze zijn niet toegestaan in Miranda* (omdat $+\!\!+$ niet is toegestaan in een patroon); we zullen verderop zeggen hoe we ze wél in Miranda kunnen opschrijven.

De eerste manier. Iedere lijst is (op verschillende manieren) te schrijven met $[]$, $[x]$ en $+\!\!+$. Bijvoorbeeld, $[a, b, c] = (([a] +\!\!+ []) +\!\!+ ([b] +\!\!+ [c])) +\!\!+ []$. We noemen dit de $+\!\!+$-opbouw. Hier is een definitie van *sum* met inductie naar de $+\!\!+$-opbouw van het argument:

$$\begin{aligned} &sum\ [] \ = \ 0 \\ &sum\ [x] \ = \ x \\ &sum\ (xs +\!\!+ ys) \ = \ sum\ xs + sum\ ys \end{aligned}$$

De tweede manier. Ieder lijst is (op één manier) te schrijven met $[]$ en $+\!\!+ [x]$. Bijvoorbeeld, $[a, b, c] = [] \ +\!\!+ [a] \ +\!\!+ [b] \ +\!\!+ [c]$. We noemen dit de postfix opbouw. Hier is een definitie van *sum* met inductie naar de postfix opbouw van het argument:

$$\begin{aligned} &sum\ [] \ = \ 0 \\ &sum\ (xs +\!\!+ [x]) \ = \ sum\ xs + x \end{aligned}$$

De derde manier. Iedere lijst is (op één manier) te schrijven met $[]$ en $[x]{+\!\!+}$. Bijvoorbeeld, $[a, b, c] = [a] +\!\!+ \ [b] +\!\!+ \ [c] +\!\!+ \ []$. We noemen dit de prefix opbouw. Hier is een definitie van *sum* met inductie naar de prefix opbouw van het argument:

$$sum\ [] \ = \ 0$$

$$sum\ ([x] +\!\!+\ xs)\ =\ x + sum\ xs$$

Geen van bovenstaande definities zijn in Miranda toegestaan, omdat $+\!\!+$ niet in patronen mag voorkomen. Geoorloofde Miranda-definities voor *sum* zien er, bijvoorbeeld, als volgt uit. Voor de inductie naar de $+\!\!+$-opbouw (we kiezen de splitsing in $xs +\!\!+ ys$ ongeveer in het midden):

$$\begin{array}{l}
sum\ []\ =\ 0 \\
sum\ [x]\ =\ x \\
sum\ zs \\
=\ sum\ xs + sum\ ys \\
\quad where \\
\quad (xs,\ ys)\ =\ (take\ (\#zs\ div\ 2)\ zs,\ \ drop\ (\#zs\ div\ 2)\ zs)
\end{array}$$

Voor de inductie naar de postfix opbouw:

$$\begin{array}{l}
sum\ []\ =\ 0 \\
sum\ zs\ =\ sum\ xs + x \quad where\ (xs,\ x)\ =\ (init\ zs,\ \ last\ zs)
\end{array}$$

Voor de inductie naar de prefix opbouw kunnen we het patroon $x : xs$ gebruiken:

$$\begin{array}{l}
sum\ []\ =\ 0 \\
sum\ (x : xs)\ =\ x + sum\ xs
\end{array}$$

Omdat lijsten door Miranda intern opgeslagen worden in de vorm $x_0 : x_1 : \ldots$, is het gebruik van een patroon met $:$ heel efficiënt. Omdat we vaak naar efficiëntie streven, zullen we patronen met $:$ dus veel tegenkomen. Maar laat je hierdoor niet misleiden: ook inductie naar de postfix opbouw kan heel efficiënt berekend worden (maar met een ietwat andere uitdrukking dan hierboven is gegeven).

Opgaven

83. Definieer op drie manieren de standaard functie *product* die het product van een getallijst oplevert. Druk hiermee de faculteitsfunctie *fac* uit.
84. Definieer op drie manieren de standaard functie *max* die het maximum van een niet-lege argumentlijst oplevert. Wenk: gebruik $max2$ om daarmee gevalsonderscheid te vermijden. (De definitie van *min* is natuurlijk analoog; te flauw om nog als opgave te geven.)
85. Geef een recursieve definitie van de standaard functies *last* en *init*:

$$\begin{array}{lcl}
last\ [x_0, \ldots, x_n] & = & x_n \\
init\ [x_0, \ldots, x_n] & = & [x_0, \ldots x_{n-1}]
\end{array}$$

waarbij $n \geq 0$. Voor de lege lijst zijn *last* en *init* niet gedefinieerd.

86. Geef een recursieve definitie van de standaard indiceringsoperatie ! :

$$[x_0, \ldots, x_{n-1}]\ !\ i \quad = \quad x_i$$

waarbij $0 \leq i < n$. Voor $n \leq i$ is de uitkomst niet gedefinieerd.

87. Geef een recursieve definitie, op drie manieren, van de standaard functie *reverse*:

$$reverse\ [x_0, \ldots, x_{n-1}] \quad = \quad [x_{n-1}, \ldots, x_0]$$

88. Geef van de standaard functie *member* een recursieve definitie:

$$member\ xs\ x \quad = \quad \text{lijst } xs \text{ bevat } x$$

89. Geef een recursieve definitie, op drie manieren, van de volgende standaard functies:

$$\begin{array}{lcl} and\ [x_0,\ldots,x_{n-1}] & = & x_0 \wedge \cdots \wedge x_{n-1} \\ or\ [x_0,\ldots,x_{n-1}] & = & x_0 \vee \cdots \vee x_{n-1} \\ concat\ [xs_0,\ldots,xs_{n-1}] & = & xs_0 + \!\!\! + \cdots + \!\!\! + xs_{n-1} \\ \#\ [x_0,\ldots,x_{n-1}] & = & 1+\cdots+1 \quad (n \text{ maal een `1'}) \end{array}$$

Waarom lukt het niet om ook de volgende functie op dezelfde drie manieren te definiëren:

$$f\ [x_0,\ldots,x_{n-1}] \quad = \quad x_0\hat{}\cdots\hat{}x_{n-1}$$

Operatie-symbool ˆ (voor: machtsverheffen) is rechts-associatief, dus $x\hat{}y\hat{}z$ staat voor $x\hat{}(y\hat{}z)$.

90. Geef, zo mogelijk op drie manieren, een recursieve definitie van de volgende functies:

$$\begin{array}{lcl} inits\ [x_0,\ldots,x_{n-1}] & = & [[],\ [x_0],\ [x_0,x_1],\ \ldots\ ,[x_0,\ldots,x_{n-1}]] \\ tails\ [x_0,\ldots,x_{n-1}] & = & [[x_0,\ldots,x_{n-1}],\ [x_1,\ldots,x_{n-1}],\ \ldots\ ,[x_{n-1}],\ []] \\ subs\ xs & = & \text{een lijst met alle deelrijen van } xs \end{array}$$

Een *deelrij* (*subrij*) van xs is een rij die uit xs ontstaat door sommige (nul of meer) elementen te schrappen.
Wenk. Voor een definitie van *inits* met inductie naar de prefix opbouw van het argument, probeer je $inits\ ([x] + \!\!\! + xs)$ uit te drukken in termen van $inits\ xs$. Dat gaat als volgt: schrijf $inits\ ([x] + \!\!\! + xs)$ in formules of in een plaatje uit en probeer die uitdrukking dan te herschrijven tot een vorm waarin $inits\ xs = \underline{[[],[x_1],[x_1,x_2],\ldots,[x_1,x_2,\cdots,x_{n-1}]]}$ als één subexpressie verschijnt. Dat lukt hieronder uiteindelijk in de één-na-laatste regel:

$$\begin{array}{ll} & inits([x] + \!\!\! + [x_1,\ldots,x_{n-1}]) \\ = & \\ & [\ [],\ [x],\ [x,x_1],\ [x,x_1,x_2],\ \ldots,\ [x,x_1,x_2,\cdots,x_{n-1}]\ ] \\ = & \\ & [[]] + \!\!\! + [\ x:[],\ x:[x_1],\ x:[x_1,x_2],\ \ldots,\ x:[x_1,x_2,\cdots,x_{n-1}]\ ] \\ = & \\ & [[]] + \!\!\! + [\ x:us \mid us \leftarrow \underline{[[],[x_1],[x_1,x_2],\ldots,[x_1,x_2,\cdots,x_{n-1}]]}\ ] \\ = & \\ & [[]] + \!\!\! + [\ x:us \mid us \leftarrow inits[x_1,x_2,\ldots,x_{n-1}]\ ] \end{array}$$

Dus $inits\ ([x] + \!\!\! + xs) \ = \ [[]] \ + \!\!\! + \ [x : us \mid us \leftarrow inits\ xs]$.
Zo ga je ook te werk bij inductie naar de postfix of $+ \!\!\! +$ opbouw van het argument, en bij de andere functies.

91. (Alleen voor de liefhebbers van “puzzels”.) Geef, zo mogelijk op drie manieren, een recursieve definitie van de volgende functie:

$$segs\ [x_0,\ldots,x_{n-1}] \quad = \quad \text{een lijst met alle segmenten } [x_i,\ldots,x_{j-1}] \quad (0 \le i \le j \le n)$$

Je hebt *inits* en *tails* van opgave 90 hierbij nodig.

Opgaven

92. De Fibonacci-getallen zijn 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...: ieder getal is de som van z'n twee voorgangers. Fibonacci getallen komen veel voor in de natuur. Bijvoorbeeld, ze duiden de grootte van een konijnenpopulatie aan in achtereenvolgende maanden, wanneer (1) elk volwassen konijn per maand één jong baart, (2) ieder konijn na één maand volwassen is, en (3) geen konijn ooit dood gaat. Immers, onder die voorwaarden is het aantal konijnen in maand $n+2$ het aantal konijnen in maand $n+1$ plus het aantal *volwassenen* in maand $n+1$ (dat is: het aantal *konijnen* in maand n).
Geef een recursieve definitie van *fib* n, Fibonacci-getal nummer n; dus *fib* $0 = 0$. Gebruik de patronen 0, 1 en $n+2$.
93. Het aantal deelverzamelingen van grootte k in een verzameling van grootte n wordt genoteerd met $\binom{n}{k}$, 'n boven k'. Dit is het aantal manieren om een greep van k objecten te doen uit n objecten. Deze getallen heten binomiaal coefficiënten. Bijvoorbeeld, $\binom{4}{3} = 4$ en $\binom{4}{2} = 6$ en $\binom{5}{3} = 10$. Definieer de functie *binom* met: *binom* $n\ k = \binom{n}{k}$.
Wenk. Hoe is een greep van $k+1$ uit $n+1$ gerelateerd aan een greep van $k+1$ uit n en een greep van k uit n? Druk dus *binom* $(n+1)\ (k+1)$ uit in *binom* $n\ (k+1)$ en *binom* $n\ k$, en behandel de gevallen $n = 0$ en/of $k = 0$ apart.
94. Definieer de functie *subs'* met:

$$subs'\ xs\ k = \text{een lijst van deellijsten-ter-grootte } k \text{ van } xs$$

Een *deelrij* (*subrij*) van xs is een rij die uit xs ontstaat door sommige (nul of meer) elementen te schrappen. Dus met $n = \#xs$ geldt: $\#subs'\ xs\ k = \binom{n}{k} = binom\ n\ k$; zie opgave 93.
95. De *ggd* (grootste gemeenschappelijke deler) kan als volgt gedefinieerd worden:

$$ggd\ m\ n \ = \ max\ [d \mid d \leftarrow [1\ ..\ m\ \underline{min2}\ n];\ m\ mod\ d \ = \ 0;\ n\ mod\ d \ = \ 0]$$

Er is ook een recursieve definitie waarin niet van *mod* gebruik wordt gemaakt, maar alleen van aftrekking (oh, wat mooi! wat elegant!). Ga na dat de volgende eigenschap geldt:

elke deler van $m+n$ en n is ook deler van m en n, en omgekeerd

Dus in het bijzonder is de grootste gemeenschappelijke deler van $m+n$ en n dezelfde als die van m en n. Ook geldt:

de grootste gemeenschappelijke deler van m en m is m

de grootste gemeenschappelijke deler van m en 0 is m

Definieer *ggd* nu recursief, gebaseerd op deze eigenschappen.
96. Definieer de standaard functie *numval* (de inverse van *shownum*):

$$numval\ xs \ = \ \text{het getal } n \text{ waarvan } xs \text{ de decimale notatie is}$$

Beperk je tot xs die uitsluitend cijfers bevatten; de echte *numval* is algemener. Probeer zowel inductie naar de prefix opbouw van het argument, als ook inductie naar de postfix opbouw.
97. Functie *binval* levert de getalwaarde van een binair genoteerd getal:

$$binval\ [x_{n-1}, \ldots, x_1, x_0] \ = \ x_{n-1}{\times}2^{n-1} + \cdots + x_1{\times}2^1 + x_0{\times}2^0$$

Neem *binval* $[] = 0$. Geef een recursieve definitie van *binval*. Doe dit drie keer: met inductie naar de prefix, postfix en $+\!\!+$-opbouw van het argument.

98. Volgens de 'negen-test' is een getal deelbaar door negen precies wanneer de som der cijfers (van de decimale notatie) deelbaar is door negen. Definieer de functies *cijfers* en *csom*:

$$\begin{array}{lcl} \mathit{cijfers}\ n & = & \text{een lijst met de cijfers (als getallen) van de decimale notatie van } n \\ \mathit{csom}\ n & = & \text{de cijfersom (van de cijfersom etc) van } n\text{; hoogstens 9} \end{array}$$

Dus n is deelbaar door negen precies wanneer $\mathit{csom}\ n = 9$.

99. Definieer het predicaat *piep* op alle natuurlijke getallen:

$$\mathit{piep}\ n \ = \ \text{`}n \text{ is een 7voud is of bevat het cijfer 7'}$$

Zie opgave 18 voor een toepassingmogelijkheid.

100. Een voorbeeld van een volstrekt nutteloze functie:

$$\begin{array}{l} f\ 1 \ = \ 1 \\ f\ n \\ \ = \ f\ (n\ \mathit{div}\ 2),\ \mathit{if}\ \ n\ \mathit{mod}\ 2 \ = \ 0 \\ \ = \ f\ (3{\times}n{+}1),\ \mathit{otherwise} \end{array}$$

Voor $f\ 5$ wordt de volgende berekening opgeroepen: $f\ 5 = f\ 16 = f\ 8 = f\ 4 = f\ 2 = 1$. Het eindigen van de berekening is mogelijk doordat één van de clausules géén recursieve aanroep bevat. Toch is het toch niet duidelijk of al de berekeningen stoppen. Sterker nog, de eindigheid van de berekeningen is niet alleen onduidelijk, maar een bekend onopgelost probleem in de wiskunde.
Definieer een functie f' die de berekening van f toont: de rij van argumenten van de recursieve aanroepen. Bijvoorbeeld, $f'\ 5 = [5, 16, 8, 4, 2, 1]$ en $f'\ 3 = [3, 10, 5, 16, 8, 4, 2, 1]$.
Geef voorts een uitdrukking die de berekening van $f\ n$ toont voor n = 1,2,... .

101. Gegeven een lijst *personen* en een functie *kinderen* (zoals bij de Oranje familie, in §6.1). Definieer de functies:

$$\begin{array}{lcl} \mathit{nakomelingen}\ n\ p & = & \text{de } n\text{-de graads nakomelingen van persoon } p \\ \mathit{voorouders}\ n\ p & = & \text{de } n\text{-de graads voorouders van persoon } p \end{array}$$

In beide gevallen moet bij graad 0 de persoon zelf opgeleverd worden. (In opgave 116 doen we dit zónder lijstcomprehensie.)

Om te onthouden

• Een veel voorkomende vorm van recursie is inductie: dan is het functie-resultaat bij een argument uitgedrukt in de resultaten van de functie op de onderdelen van het argument.

• Ga bij het opstellen van een inductieve definitie als volgt te werk: veronderstel dat je de functie-waarde bij (geschikt gekozen) onderdelen van het argument al kent, en probeer dan daarmee de functie-waarde van het geheel uit te drukken.

• Bij een recursieve definitie is er het gevaar dat een berekening niet stopt: "oneindige recursie". Daarom is er vaak tenminste één niet-recursieve clausule.

11 Functies als gewone waarden

Functies mogen in lijsten en tupels voorkomen, en als argument en resultaat optreden. Functies en operatoren hoeven bij gebruik niet álle argumenten te krijgen; bij "te weinig" argumenten is het resultaat een functie van de nog ontbrekende argumenten.

§11.1 Secties. In Miranda staat (/2) voor de functie die zijn argument deelt door 2. Dus (/2) 6 levert 3.0. Iets soortgelijks geldt voor (2/) en (/); een operator tussen haakjes is een gewone functie geworden. Dus:

$$(/)\;2\;6 \quad = \quad (2/)\;6 \quad = \quad (/6)\;2 \quad = \quad 2/6 \quad = \quad 0.333333333333$$
$$\begin{array}{lll} (2/) & :: & num \rightarrow num \\ (/6) & :: & num \rightarrow num \\ (/) & :: & num \rightarrow num \rightarrow num \end{array}$$

Een operator samen met één of géén argument, en door haakjes tot een geheel gemaakt, heet een *sectie*. Secties worden veel gebruikt. Hier nog wat meer voorbeelden van secties:

$$\begin{array}{l} (+2),\;\; (\neq 2),\;\; (\leq 10),\;\; (2\;:),\;\; ([1,2,3]{+\!\!+}),\;\; ({+\!\!+}[9,10]), \\ (+\;sqrt\;2),\;\; (=\;fac\;5), \\ (\#),\;\; (-),\;\; (:),\;\; ({+\!\!+}) \end{array}$$

Er is één uitzondering: (-2) is geen sectie maar hetzelfde als -2. Let op de secties in de tweede regel: '$(+\;sqrt\;2)$' staat voor '$(+\;(sqrt\;2))$' want, zoals altijd, een functie–argument binding heeft voorrang boven een operator–operand binding.

Ook functies hoeven niet van alle argumenten voorzien te worden. Bijvoorbeeld, na:

$$takeTwo \;=\; take\;2$$

is *takeTwo* de functie die, gegeven een lijst, de eerste twee elementen ervan oplevert:

$$takeTwo\;[a,b,c,d,e] \quad = \quad take\;2\;[a,b,c,d,e] \quad = \quad [a,b]$$

Je ziet hier het voordeel van 'losse parameters' boven één tupel als parameter: van 'losse parameters' kun je de eerste (of de eerste twee of eerste drie etc) vastpinnen en dan het resultaat net zo behandelen als iedere andere functie. In feite kun je ook alleen de tweede parameter gemakkelijk vastpinnen: ($\underline{take}\;[0,1,2,3,4]$) is zelf een functie van type $num \rightarrow [num]$.

§11.2 Functies op zichzelf. Getallen, karakters, getallijsten, lijsten van getallijsten, enzovoorts heten wel 'waarden'. Zij kunnen optreden als argument en resultaat van een operator of functie, en als component van een lijst of tupel. In functionele talen behoren ook de functies tot de waarden: ook zij kunnen optreden als argument en resultaat van een operatie of functie, en als component van een lijst of tupel. Definieer, bijvoorbeeld:

$$\begin{aligned}
&map\ f\ xs\ =\ [f\ x \mid x \leftarrow xs]\\
&fcts\ =\ [(\times 2),\ fac,\ (2/),\ (mod\ 10),\ sincos\ 1]\\
&sincos\ 1\ =\ sin\\
&sincos\ 2\ =\ cos
\end{aligned}$$

Schematisch: $map\ f\ [x_0, \ldots, x_{n-1}] = [f\ x_0, \ldots, f\ x_{n-1}]$. Functies treden hier op als parameter (bij *map*), als lijstelement (bij *fcts*), en als functieresultaat (bij *sincos*). Voorbeelden:

$$\begin{array}{lcl}
map\ (+10)\ [1,2,3] & = & [11,\ 12,\ 13]\\
map\ (<10)\ [1,12,3] & = & [True,\ False,\ True]\\
map\ (\#)\ [[1,2,3],\ [30,40,50,60],\ [7]] & = & [3,\ 4,\ 1]\\
map\ (take\ 2)\ [[1,2,3],\ [30,40,50,60],\ [7]] & = & [[1,2],\ [30,40],\ [7]]\\
[f\ 6 \mid f \leftarrow fcts] & = & [12,\ 720,\ 0.333\ldots,\ 6,\ -0.27941\ldots]\\
[f\ 6 \mid f \leftarrow take\ 3\ fcts] & = & [12,\ 720,\ 0.333\ldots]\\
[f\ 6 \mid f \leftarrow take\ 3\ fcts;\ \ f\ 4\ >\ 10] & = & [720]\\
(sincos\ 2)\ 0 & = & 1
\end{array}$$

De type-specificatie van *map*, *fcts*, en *sincos* luidt als volgt:

$$\begin{aligned}
&map\ ::\ (\alpha \to \beta) \to [\alpha] \to [\beta]\\
&fcts\ ::\ [num \to num]\\
&sincos\ ::\ num \to (num \to num)
\end{aligned}$$

Dus in '$map\ f\ xs$' moeten het argumenttype van f en het elementtype van xs gelijk zijn; alle elementen van *fcts* hebben eenzelfde type, namelijk $num \to num$; en functie *sincos* levert altijd een resultaat van type $num \to num$.

In Miranda zijn de haakjes in '$(sincos\ 2)\ 0$' en in '$sincos\ ::\ num \to (num \to num)$' niet nodig; we mogen ook schrijven '$sincos\ 2\ 0$' en '$sincos\ ::\ num \to num \to num$'. Dus *sincos* wordt precies zo behandeld als een functie met twee *num*-argumenten en een *num*-resultaat. Dit komt helemaal overeen met de mogelijkheid die we hierboven bij *take* al noemden: een functie hoef je niet in één keer van álle argumenten te voorzien. Bijvoorbeeld, uit deze haakjes-afspraak volgt voor $take\ ::\ num \to [\alpha] \to [\alpha]$ dat ook:

$$take\ \ ::\ \ num \to ([\alpha] \to [\alpha])$$

en dus is $take\ 2$ op zichzelf al een functie, namelijk van type $[\alpha] \to [\alpha]$.

§11.3 Functiecompositie. Stel we willen getallen achtereenvolgens met 2 vermeerderen, met 3 vermenigvuldigen, en met 15 vergelijken Deze samengestelde bewerking kunnen we visueel voorstellen door:

$$\longleftarrow \boxed{(15 <)} \longleftarrow \boxed{(3\times)} \longleftarrow \boxed{(2+)} \longleftarrow$$

In Miranda kunnen we die samenstelling noteren als:

$$(15 <)\ .\ (3\times)\ .\ (2+)$$

De secties $(15 <)$, $(3\times)$ en $(2+)$ ken je al. Operatie $.$ is functiecompositie; per definitie geldt:

$$(f\ .\ g)\ x\ =\ f\ (g\ x)$$

en algemener:

$$(f\ .\ g\ .\ h\ .\ j)\ x\ =\ f\ (g\ (h\ (j\ x)))$$

Operatie $.$ is een 'gewone' operatie, net als $+$ en $\times$; alleen werkt-ie met functies en niet met getallen. Spreek $f\ .\ g$ uit als 'f na g'. De samengestelde bewerking kunnen we een naam geven:

$$f\ =\ (15 <)\ .\ (3\times)\ .\ (2+)$$

Er geldt dan voor alle x dat $f\ x = 15 <\ (3\times\ (2+\ x))$.

Hier is een "nuttiger" voorbeeld. Beschouw de volgende functie definitie (uit §5.2):

$$\mathit{puzzels}\ n\ \mathit{abc}\ =\ \mathit{kop}\ +\!\!+\ \mathit{layn}\ [\mathit{showOpl}\ \mathit{opl} \mid \mathit{opl} \leftarrow \mathit{opln}\ n\ \mathit{abc}]$$

waarbij de betekenis van *kop*, *showOpl* en *opln* er nu niet toe doet. Uit de vorige paragrafen volgt dat *puzzels* n en $(\mathit{kop}+\!\!+)$ gewone functies zijn: secties. Met secties en *map* kunnen we de definitie ook schrijven als:

$$(\mathit{puzzels}\ n)\ \mathit{abc}\ =\ (\mathit{kop}+\!\!+)\ (\mathit{lay}\ (\mathit{map}\ \mathit{showOpl}\ ((\mathit{opln}\ n)\ \mathit{abc})))$$

Je ziet in de linkerkant dat *abc* wordt onderworpen aan de functie *puzzels* n, en in de rechterkant dat *abc* wordt onderworpen aan achtereenvolgens de functies *opln* n, *map showOpl*, *lay* en $(\mathit{kop}+\!\!+)$. Dus functie *puzzels* n is niets meer en minder dan de compositie van die vier andere functies. Dit kunnen we duidelijker noteren met behulp van *functiecompositie*:

$$\mathit{puzzels}\ n\ =\ (\mathit{kop}+\!\!+)\ .\ \mathit{lay}\ .\ \mathit{map}\ \mathit{showOpl}\ .\ \mathit{opln}\ n$$

Dus functie *puzzels* n is de volgende samengestelde bewerking:

$$\longleftarrow \boxed{(\mathit{kop}+\!\!+)} \longleftarrow \boxed{\mathit{lay}} \longleftarrow \boxed{\mathit{map}\ \mathit{showOpl}} \longleftarrow \boxed{\mathit{opln}\ n} \longleftarrow$$

Let op de haakjes: zoals altijd heeft een functie–argument binding voorrang boven een operator–operand binding. Dus in '$f\ .\ g\ .\ h\ .\ j\ x$' wordt éérst $j\ x$ bij elkaar genomen en wordt *het resultaat daarvan* samengesteld met de andere functies:

$$\begin{aligned}(f\ .\ g\ .\ h\ .\ j)\ x\ &=\ f\ (g\ (h\ (j\ x)))\\ (f\ .\ g\ .\ h\ .\ j\ x)\ y\ &=\ f\ (g\ (h\ (j\ x\ y)))\end{aligned}$$

Met verscheidene 'losse' argumenten komt de eerste bij de 'binnenste' functie, en de rest bij de 'buitenste':

$$(f\ .\ g\ .\ h\ .\ j)\ x_0\ x_1\ \ldots\ x_{n-1}\ =\ f\ (g\ (h\ (j\ x_0)))\ x_1\ \ldots x_{n-1}$$

In het rechterlid heeft f de argumenten: $g(h(j\ x_0))$ en x_1 en $\ldots$ en x_{n-1}.

§11.4 Componeren met zip. Composities van *map* na *zip* komen veel voor. Bijvoorbeeld, de stijgendheid van een lijst *xs* kan als volgt worden uitgedrukt:

$$xs \text{ is stijgend} \;=\; (and\ .\ map\ f\ .\ zip)\ (xs,\ tl\ xs) \qquad where\ f\ (x,y) \;=\; x \le y$$

De benodigde hulpfunctie f is vrijwel identiek aan de sectie ($\le$); het enige verschil is dat f een tupel als argument krijgt (uit de gezipte lijst) terwijl ($\le$) de argumenten één voor één krijgt. Om zulke flauwe wijzigingen voor eens en altijd te voorkomen, definiëren we *zipwith* als de bedoelde samenstelling van *map* na *zip*. Daarmee kunnen we bovenstaande schrijven als:

$$xs \text{ is stijgend} \;=\; (and\ .\ zipwith\ (\le)\ )\ (xs,\ tl\ xs)$$

Functie *zipwith* is gedefinieerd door:

$$zipwith\ f \;=\; map\ f'\ .\ zip \quad where\ f'\ (x,y) \;=\; f\ x\ y$$

of met behulp van de standaard functie *map*2:

$$zipwith\ f\ (xs,\ ys) \;=\; map2\ f\ xs\ ys$$

Helaas is *zipwith* niet standaard aanwezig; *map*2 wel.

Opgaven

102. Een predicaat is een functie met boolean resultaat. Geef een *uitdrukking* (geen definitie!) met behulp van secties en functiecompositie voor de volgende predicaten: (1) ‘is even’, (2) ‘is deler van n’, (3) ’verschilt van karakter `a`’, (4) ‘heeft lengte 4’, (5) ‘heeft 5 op kop’, (6) ‘heeft een element dat gelijk is aan 6’. Geef ook voor ieder het type aan.
103. Geef een recursieve definitie van de standaard functie *filter*:

$$filter\ p\ xs \;=\; [x \mid x \leftarrow xs;\ p\ x]$$

Wat is het type van *filter*?
Geef nu, met behulp van de antwoorden uit de vorige opgave, een uitdrukking voor: (1) de lijst van even getallen, (2) de lijst van delers van n, (3) alle letters verschillend van $'a'$ uit een gegeven lijst *xs*, (4) alle elementen van een lijst-van-lijsten *xss* die lengte 4 hebben, (5) alle elementen van een lijst-van-lijsten *yss* die 5 op kop hebben, (6) alle elementen van een lijst-van-lijsten *zss* die een element 6 bevatten.
104. Geef een recursieve definitie van de standaard functie *map*.
105. Geef niet-recursieve definitie van *unzip*, die van een lijst van paren een paar van lijsten maakt (zie opgave 59). Gebruik geen lijstcomprehensie (maar wel *map*, *fst* en *snd*).
106. Predicaat *all* wordt gespecificeerd door: $all\ p\ xs =$ ‘alle elementen van *xs* voldoen aan predicaat p’. Definieer $all\ p$ met behulp van functiecompositie en de standaard functies *and* en *map*.
107. Geef een recursieve definitie van de standaard functie *takewhile*:

$takewhile\ p\ xs =$

het langste beginstuk van *xs* waarvan alle elementen voldoen aan p

Geef ook een niet-recursieve definitie: druk *takewhile* p uit met behulp van functiecompositie en de functies *all*, *inits*, *filter* en *last*.

108. Geef een niet-recursieve en een recursieve definitie van de functie *dropwhile* met:

$$\mathit{takewhile}\ p\ xs\ +\!\!+\ \mathit{dropwhile}\ p\ xs \;=\; xs$$

109. Geef een recursieve definitie van de standaard functie *mkset*. De specificatie luidt:

(1) $x \in \mathit{mkset}\ xs \quad\equiv\quad x \in xs$

(2) *mkset xs* heeft geen dubbele voorkomens

Gebruik *filter*.

110. Geef een recursieve definitie van de standaard functie *lay*:

$$\mathit{lay}\ [xs_0, xs_1, \ldots, xs_{n-1}] \;=\; xs_0 +\!\!+ \texttt{"}\backslash n\texttt{"} +\!\!+ xs_1 +\!\!+ \texttt{"}\backslash n\texttt{"} +\!\!+ \cdots +\!\!+ xs_{n-1} +\!\!+ \texttt{"}\backslash n\texttt{"}$$

Geef ook niet-recursieve definities, één met behulp van *concat* en lijstcomprehensie, en één met behulp van compositie en *concat* en *map* (en een sectie).

111. Geef een uitdrukking voor de lijst met als enige twee waarden: $(-b \pm \sqrt{b^2-4ac})/2a$. Gebruik een lijstcomprehensie met generator $pm \leftarrow [(+),(-)]$. Veronderstel dat a, b en c bekend zijn.

112. Geef een uitdrukking voor alle viertallen operaties uit $\{+, -, \times, /\}$ die het volgende sommetje kloppend maken: $((((2 ? 4) ? 5) ? 2) ? 9) = 0$.
Geef er ook een afdruk van.

113. De functie *vrnl* ('van rechts naar links') levert bij argument n het getal waarvan de decimale notatie vrnl gelezen hetzelfde is als die van n vlnr gelezen. Bijvoorbeeld, *vrnl* $572 = 275$. Definieer *vrnl* als de compositie van drie functies. (Vergelijk met opgave 13.)

114. Gegeven is een eenvoudig intern telefoonboek van de faculteit, precies als in opgave 43:

$$\mathit{telboek} \;=\; [(\texttt{"}\mathit{Peter}\texttt{"}, 3683),\ (\texttt{"}\mathit{Herman}\texttt{"},\ 3772),\ (\texttt{"}\mathit{Klaas}\texttt{"}, 3783),\ \ldots]$$

Definieer de volgende functies en lijsten, zonder lijstcomprehensie te gebruiken:

tel nm	=	een lijst van alle *nr*'s met $(nm, nr) \in \mathit{telboek}$	(telefoonnummer)
abb nr	=	een lijst van alle *nm*'n met $(nm, nr) \in \mathit{telboek}$	(abbonnee)
tels	=	een lijst van alle *nm*'n met $(nm, ...) \in \mathit{telboek}$	(telefoonnummers)
abbs	=	een lijst van alle *nr*'s met $(..., nr) \in \mathit{telboek}$	(abbonnees)

115. Een *tekst* is een lijst van karakters; hierin onderscheiden we *woorden* (begrensd door spaties en newlines), *regels* (begrensd door newlines), en *alinea's* (begrensd door lege regels; engels: paragraph). Om een tekst in deze delen te ontleden zijn er de functies:

$$\mathit{words},\ \mathit{lines},\ \mathit{paras}\ ::\ [\mathit{char}]\rightarrow[[\mathit{char}]]$$

Om uit woorden etc weer een tekst op te bouwen zijn er de volgende functies:

wordgrps n: partitioneert een lijst van woorden in woordgroepen zó dat van iedere woordgroep alle woorden samen op een regel ter lengte n passen;
$\mathit{wordgrps}\ ::\ \mathit{num} \rightarrow [[\mathit{char}]] \rightarrow [[[\mathit{char}]]]$.

line n: maakt van een woordgroep een regel ter lengte n plus de afsluitende newline (aangenomen dat de woorden samen op zo'n regel passen);
$line :: num \to [[char]] \to [char]$.

para: maakt van een lijst van regels een alinea plus de afsluitende lege regel;
$para :: [[char]] \to [char]$.

Definieer met behulp van compositie (en *map* etc) de functie *format n* die een tekst omvormt tot een tekst waarin alle regels n posities lang zijn; de verdeling in alinea's blijft ongewijzigd. *De toevoeging en/of verwijdering van spaties en newlines gebeurt uitsluitend door de gegeven functies.*

116. Net als in opgave 101 is er gegeven een lijst *personen* en een functie *kinderen*:

$$
\begin{array}{l}
persoon \equiv \ldots \\
personen :: [persoon] \\
kinderen :: persoon \to [persoon]
\end{array}
$$

Definieer nu zonder lijstcomprehensie de functies:

$$
\begin{array}{lcl}
nakomelingen\ n\ p & = & \text{de } n\text{-de graads nakomelingen van persoon } p \\
voorouders\ n\ p & = & \text{de } n\text{-de graads voorouders van persoon } p
\end{array}
$$

In beide gevallen moet bij graad 0 de persoon zelf opgeleverd worden.

117. Definieer de lijst $[(i, n-i) \mid n \leftarrow [0..];\ i \leftarrow [0..n]]$ zonder lijstcomprehensie. Gebruik uitsluitend *zip* om de paren $(i, n-1)$ te maken.

118. Functie f wordt als volgt gedefinieerd in termen van functies g en p:

$$
\begin{array}{l}
f\ n\ []\ =\ 0 \\
f\ n\ (x : xs) \\
\ =\ g\ x,\ \mathit{if}\ p\ n\ x \\
\ =\ f\ n\ xs,\ \mathit{otherwise}
\end{array}
$$

Deze definitie kan ook in één regel geformuleerd worden:

$$f\ n\ xs\ =\ hd\ ([g\ x \mid x \leftarrow xs;\ p\ n\ x]\ +\!\!+\ [0])$$

Schrijf deze definitie zonder lijstcomprehensie, en zoveel mogelijk met functie-compositie.

Opgaven

Dit is de inleiding voor de opgaven in deze serie.

Een *matrix* is een rechthoek van getallen (of algemener: van 'elementen'). Bijvoorbeeld, hier staan een 4-bij-3 en een 3-bij-4 getallen-matrix:

$$
\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}
\qquad\qquad
\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}
$$

We onderscheiden *rijen* en *kolommen*; een m-bij-n matrix heeft m rijen en n kolommen. Hierboven staan de rijen horizontaal getekend. Een m-bij-n matrix heet *vierkant* als $m = n$. We representeren een m-bij-n matrix als een m-lijst van n-lijsten, dus als een stel rijen (iedere rij heeft lengte n). De linker en rechter matrix hierboven worden gerepresenteerd als:

$$[[1,2,3],\ [4,5,6],\ [7,8,9],\ [10,11,12]] \qquad [[1,4,7,10],\ [2,5,8,11],\ [3,6,9,12]]$$

119. Definieer zonder lijstcomprehensie de functie die de diagonaal $[x_{0,0}, x_{1,1}, \ldots]$ van een vierkante matrix oplevert.
120. Functie *transpose* "spiegelt" een m-bij-n matrix langs een NW-ZO diagonaal tot een n-bij-m matrix. Bijvoorbeeld, *transpose* op de linker matrix hierboven geeft de rechter, en *transpose* op de rechter matrix geeft de linker.
Definieer de functie *transpose* met inductie naar de prefix opbouw van het argument, en ga er van uit dat het argument een m-bij-n matrix voorstelt met $m \geq 1$ en $n \geq 1$. Maak gebruik van *map*, *zip* en/of *zipwith*. (De standaard functie *transpose* is nog iets algemener.)
121. Een *scalar* is een getal, een *vector* is een rij van getallen, en een *matrix* is een rechthoek van getallen. De volgende bewerkingen zijn op dit soort grootheden gedefinieerd:

$$a \odot \begin{pmatrix} x_1 & \ldots & x_n \end{pmatrix} = \begin{pmatrix} a{\times}x_1 & \ldots & a{\times}x_n \end{pmatrix} \quad \text{scalair product}$$

$$\begin{pmatrix} x_1 & \ldots & x_n \end{pmatrix} \oplus \begin{pmatrix} y_1 & \ldots & y_n \end{pmatrix} = x_1{\times}y_1 + \cdots + x_n{\times}y_n \quad \text{inproduct}$$

$$\begin{pmatrix} x_{11} & \ldots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{k1} & \ldots & x_{km} \end{pmatrix} \otimes \begin{pmatrix} y_{11} & \ldots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \ldots & y_{mn} \end{pmatrix} = \begin{pmatrix} z_{11} & \ldots & z_{1n} \\ \vdots & \ddots & \vdots \\ z_{k1} & \ldots & z_{kn} \end{pmatrix} \quad \text{matrix product}$$

waarbij

$$\begin{aligned} z_{ij} &= x_{i1}{\times}y_{1j} + \cdots + x_{im}{\times}y_{mj} \\ &= (x_{i1}\ \ldots\ x_{im}) \oplus (y_{1j}\ \ldots\ y_{mj}) \\ &= \text{rij } i \text{ van } xss \oplus \text{kolom } j \text{ van } yss \end{aligned}$$

We representeren een vector door een getallijst, en een matrix door een lijst van vectoren (de rijen van de rechthoek). Definieer hierbij de operaties $\odot$ (nu *scalprod* genoemd), $\oplus$ (nu *inprod* genoemd), en $\otimes$ (nu *matprod* genoemd). Maak voor *scalprod* en *inprod* gebruik van *map*, *zip* en/of *zipwith*. Definieer *matprod* zonder lijstcomprehensie; maak desgewenst gebruik van *transpose* van opgave 120.
122. Een *greep* in een vierkante m-bij-m matrix is een m-tal elementen, uit iedere rij en iedere kolom één; dus een m-tal elementen $x_{i,j}$ die alle verschillende rij-index i hebben, en ook verschillende kolom-index j. Bijvoorbeeld, van een 3-bij-3 matrix met elementen x_{ij} zijn dit de grepen (de volgorde van de lijsten en binnen iedere lijst doet niet ter zake):

$$[x_{0,0}, x_{1,1}, x_{2,2}], [x_{0,0}, x_{1,2}, x_{2,1}], [x_{0,1}, x_{1,0}, x_{2,2}], [x_{0,1}, x_{1,2}, x_{2,0}], [x_{0,2}, x_{1,0}, x_{2,1}], [x_{0,2}, x_{1,1}, x_{2,0}].$$

Definieer een functie die alle grepen van zijn argument-matrix oplevert. Doe dit op twee manieren: één waarbij de voorste elementen in de grepen uit één rij komen, en één waarbij die uit één kolom komen.

123. (Vervolg op de vorige opgave.)
Het aantal grepen wordt gegeven door de functie $grepental = (\#) \,.\, grepen$. Leid hiervoor een efficiënte definitie af.

Om te onthouden

- Composities van *map f* en *filter p* vormen een leesbaar alternatief voor veel lijstcomprehensies. Hierbij zijn f en p vaak secties.
- Gebruik zo nodig *zipwith f* als op de resultaten van *zip* nog f moet worden toegepast. Door *zipwith* wordt de $f :: t0 \rightarrow t1 \rightarrow t$ automatisch omgezet tot een $f' :: (t0, t1) \rightarrow t$.
- Denk aan de regels van wat-bij-wat hoort:

$$f \;=\; g \,.\, h \,.\, j \qquad \equiv \qquad f\ x \;=\; g\ (h\ (j\ x))$$
$$f\ x \;=\; g \,.\, h \,.\, j\ x \qquad \equiv \qquad f\ x\ y \;=\; g\ (h\ (j\ x\ y))$$

en

$$(f \,.\, g \,.\, h \,.\, j)\ x\ y\ \ldots\ z \quad = \quad f \quad (g\ (h\ (j\ x)))\ \ y\ \ \ldots\ \ z$$

12 Voorbeeld: Gauss' eliminatie

We ontwerpen hier een definitie voor de functie *gauss* die van 'n vergelijkingen met n onbekenden' de oplossing oplevert. De methode staat bekend als Gauss' eliminatie-methode.

§12.1 Het voorbeeld. We construeren de definitie van functie *gauss* aan de hand van een voorbeeld. Beschouw een *homogeen* stelsel vergelijkingen zoals links hieronder; rechts ervan staat hoe wij het representeren:

$$\begin{array}{rcrcrcrcl} 6x & + & 21y & - & 6z & - & 30 & = & 0 \\ 2x & + & 10y & - & z & - & 19 & = & 0 \\ -4x & - & 11y & + & 7z & + & 5 & = & 0 \end{array} \qquad \begin{array}{r} [6,\ \ 21, -6, -30] \\ [2,\ \ 10, -1, -19] \\ [-4, -11,\ \ 7,\ \ \ 5] \end{array}$$

De oplossing voor x, y, z bepalen we door het stelsel te transformeren tot 'driehoeksvorm':

$$\begin{array}{rcrcrcrcl} \ldots x & + & \ldots y & + & \ldots z & + & \ldots & = & 0 \\ & & \ldots y & + & \ldots z & + & \ldots & = & 0 \\ & & & & \ldots z & + & \ldots & = & 0 \end{array} \qquad \begin{array}{r} [\ldots, \ldots, \ldots, \ldots] \\ [\ldots, \ldots, \ldots] \\ [\ldots, \ldots] \end{array}$$

Immers, hieruit is eerst z te bepalen, en dan ook y, en tenslotte ook x, indien de voorste coefficiënten verschillen van 0.

§12.2 Eliminatie. De transformatie tot driehoeksvorm verloopt als volgt. Elimineer x uit de tweede en derde vergelijking, door de eerste vergelijking $-\frac{2}{6}$de keer bij de tweede op tellen, en $\frac{4}{6}$de keer bij de derde. Dit levert:

$$\begin{array}{rcrcrcrcl} 6x & + & 21y & - & 6z & - & 30 & = & 0 \\ & & 3y & + & z & - & 9 & = & 0 \\ & & 3y & + & 3z & - & 15 & = & 0 \end{array} \qquad \begin{array}{r} [6, 21, -6, -30] \\ [3,\ \ 1,\ \ -9] \\ [3,\ \ 3, -15] \end{array}$$

Dit noemen we een eliminatie-stap; voor het elimineren van 'n onbekende uit één vergelijking definiëren we straks operatie *elim*. In de volgende eliminatie-stap elimineren we y uit de derde vergelijking door de tweede vergelijking $-\frac{3}{3}$ keer bij de derde op te tellen. Dit levert:

$$\begin{array}{rcrcrcrcl} 6x & + & 21y & - & 6z & - & 30 & = & 0 \\ & & 3y & + & z & - & 9 & = & 0 \\ & & & & 2z & - & 6 & = & 0 \end{array} \qquad \begin{array}{r} [6, 21, -6, -30] \\ [3,\ \ 1,\ \ -9] \\ [\ \ 2,\ \ -6] \end{array}$$

Hiermee is de driehoeksvorm bereikt.

§12.3 Substitutie. Nu de bepaling van de oplossing. Het blijkt straks handig te zijn om het stelsel iets uniformer te noteren, namelijk door óók de constanten als coefficiënt te noteren. Dat kan met een nieuwe "bekende onbekende" u, bijvoorbeeld $u = 1$:

$$\begin{array}{rcrcrcrcl} 6x & + & 21y & - & 6z & - & 30u & = & 0 \\ & & 3y & + & z & - & 9u & = & 0 \\ & & & & 2z & - & 6u & = & 0 \\ \hline & & & & & & u & = & 1 \end{array} \qquad \begin{array}{r} [6, 21, -6, -30] \\ [3, 1, -9] \\ [2, -6] \\ \hline 1 \end{array}$$

We hebben onder de streep het alreeds gevonden deel van de oplossing geschreven. Uit de al bekende oplossing 1 voor u en de voorlaatste vergelijking volgt de oplossing voor z:

$$\begin{array}{rrrrrrrrrcr} 6x & + & 21y & - & 6z & - & 30u & = & 0 \\ & & 3y & + & z & - & 9u & = & 0 \\ \hline & & & & & & z & = & 3 \\ & & & & & & u & = & 1 \end{array} \qquad \begin{array}{r} [6,21,-6,-30] \\ [3,1,-9] \\ \hline 3 \\ 1 \end{array}$$

Dit noemen we een substitutie-stap: een substitutie van [1] voor $[u]$ in de vergelijking voor z. Daarvoor definiëren we straks operatie *subst*. De oplossing voor y volgt door substitutie van $[3,1]$ voor $[z,u]$ in de vergelijking voor y:

$$\begin{array}{rrrrrrrrr} 6x & + & 21y & - & 6z & - & 30u & = & 0 \\ \hline & & & & & & y & = & 2 \\ & & & & & & z & = & 3 \\ & & & & & & u & = & 1 \end{array} \qquad \begin{array}{r} [6,21,-6,-30] \\ \hline 2 \\ 3 \\ 1 \end{array}$$

En tenslotte levert de substitutie van $[2,3,1]$ voor $[y,z,u]$ de oplossing voor x:

$$\begin{array}{rcr} \hline x & = & 1 \\ y & = & 2 \\ z & = & 3 \\ u & = & 1 \end{array} \qquad \begin{array}{r} \hline 1 \\ 2 \\ 3 \\ 1 \end{array}$$

De oplossing voor $[x,y,z]$ (zonder u) is de *init* van deze oplossing voor $[x,y,z,u]$.

§12.4 Implementatie. Eén van de eliminaties van onbekende x luidde als volgt:

$$\begin{array}{ll} \text{“}2x + 10y - z - 19 \;=\; 0\text{”} & [2,10,-1,-19] \\ \Big\Downarrow \;\; \begin{array}{l}\text{vermeerderen met } -\frac{2}{6} \text{ keer} \\ \text{“}6x + 21y - 6z - 30 \;=\; 0\text{”}\end{array} & \Big\Downarrow \;\; ([6,21,-6,-30] \;\underline{elim}) \\ \text{“}3y + z - 9 \;=\; 0\text{”} & [3,1,-9] \end{array}$$

Deze bewerking noemen we *elim*. Dus:

$$[a,b,c,d] \;\underline{elim}\; [a',b',c',d'] \quad = \quad [-\tfrac{a'}{a}\times b+b', -\tfrac{a'}{a}\times c+c', -\tfrac{a'}{a}\times d+d']$$

Het ligt voor de hand hoe we nu xs $\underline{elim}$ ys definiëren. De gebruikte hulpfuncties staan bekend als 'scalair product' en 'vectoriele optelling':

$$\begin{array}{l} (x : xs) \;\underline{elim}\; (y : ys) \;=\; ((-y/x) \;\underline{scalprod}\; xs) \;\underline{vecplus}\; ys \\ a \;\underline{scalprod}\; xs \;=\; map\ (a\times)\ xs \\ xs \;\underline{vecplus}\; ys \;=\; zipwith\ (+)\ (xs, ys) \end{array}$$

Let op..., *delen door nul is flauwekul* en leidt tot een foutstop: dit doet zich bij *elim* zich voor als $x = 0 \wedge y \neq 0$. In dit geval bestaat er geen unieke oplossing; desgewenst kan daarvoor een geschikt *error*-alternatief toegevoegd worden aan de definitie van *elim*.

Door herhaaldelijk toepassen van een eliminatie-stap op een steeds kleiner en verder getransformeerd deel ontstaat een driehoeksvorm:

$$\begin{array}{l} \mathit{driehoek}\ (xs : xss)\ =\ xs :\ \mathit{driehoek}\ (\mathit{map}\ (xs\ \underline{\mathit{elim}})\ xss) \\ \mathit{driehoek}\ []\ =\ [] \end{array}$$

De laatste clausule dekt het geval van nul vergelijkingen met nul onbekenden; die staat al in driehoeksvorm. Uit de definitie volgt tevens dat *driehoek* $[xs] = [xs]$; één vergelijking met één onbekende staat ook al in driehoeksvorm.

Nu de substitutie-bewerking. Een toepassing ervan luidde:

$$\begin{array}{lll} \text{“}6x + 21y + -6z\ -30u\ =\ 0\text{”} & \qquad & [6, 21, -6, -30] \\ \Bigg\Downarrow\quad \text{substitutie van } [2,3,1] \text{ voor } [y,z,u] & & \Bigg\Downarrow\quad (\ \underline{\mathit{subst}}\ \ [2,3,1]) \\ \text{“}x\ =\ -\ (21{\times}2 - 6{\times}3 - 30{\times}1)\ /\ 6\text{”} & & -\ (21{\times}2 + -6{\times}3 + -30{\times}1)\ /\ 6 \end{array}$$

De uitkomst is dus de oplossing voor één onbekende; in dit geval x. De algemene vorm is duidelijk:

$$(x : [x_0, \ldots, x_{n-1}])\ \underline{\mathit{subst}}\ [y_0, \ldots, y_{n-1}]\ \ =\ \ -(x_0{\times}y_0 + \cdots + x_{n-1}{\times}y_{n-1})/x$$

De deelbewerking met $[x_0, \ldots, x_{n-1}]$ en $[y_0, \ldots, y_{n-1}]$ staat bekend als het *inproduct*; daarmee is *subst* direct uit te drukken:

$$\begin{array}{l} xs\ \ \underline{\mathit{inprod}}\ \ ys\ =\ \mathit{sum}\ (\mathit{zipwith}\ (\times)\ (xs, ys)) \\ (x : xs)\ \ \underline{\mathit{subst}}\ \ ys\ =\ -\ (xs\ \ \underline{\mathit{inprod}}\ \ ys)\ /\ x \end{array}$$

Wat we nodig hebben is een operatie die de meegegeven oplossing uitbreidt:

$$xs\ \ \underline{\mathit{bij}}\ \ ys\ =\ xs\ \ \underline{\mathit{subst}}\ \ ys\ :\ ys$$

De oplossing van een stelsel $[xs, \ldots, ys, zs]$ *in driehoeksvorm* wordt dan verkregen door de oplossing $[1]$ voor $[u]$ steeds verder uit te breiden met behulp van de eraanvoorafgaande vergelijkingen:

$$\mathit{oplossing}\ [xs, \ldots, ys, zs]\ =\ xs\ \ \underline{\mathit{bij}}\ \ (\ldots\ (ys\ \ \underline{\mathit{bij}}\ \ (zs\ \ \underline{\mathit{bij}}\ \ [1])))$$

Het is niet moeilijk om *oplossing* met inductie te definiëren (met inductie naar de prefix opbouw van het argument). Met de *foldr* functie van §13.7 kan het nog eenvoudiger; tevens passen we *init* toe om de oplossing voor de ‘bekende onbekende’ $u = 1$ weg te halen:

$$\mathit{oplossing}\ =\ \mathit{init}\ .\ \mathit{foldr}\ \mathit{bij}\ [1]$$

Tenslotte de gehele functie *gauss*. Van een willekeurig stelsel vergelijkingen wordt de oplossing verkregen door het met *driehoek* in driehoeksvorm te brengen, en dan met *oplossing* op te lossen:

$$gauss \ = \ oplossing \ . \ driehoek$$

Opgaven

124. Definieer *gauss* zélf recursief; onderscheid de gevallen [] en $xs : xss$, en gebruik zo nodig de functies *elim* en *bij* en *subst*.
125. Delen door hele kleine getallen geeft grote onnauwkeurigheden. Daarom verdient het aanbeveling om bij de eliminatie van onbekende x niet 'de eerste de beste' vergelijking daarvoor te gebruiken, maar juist díe vergelijking waarin de coefficiënt van x maximaal is, in absolute waarde. (Het voorgaande veronderstelt dat de coefficiënten-matrix ge-equilibreerd is.) Het her-arrangeren van de vergelijkingen op zodanige manier dat die vergelijking op kop komt te staan, noemen we *pivoteren*.
    Definieer de her-arrangeer functie *pivot* waarmee een numeriek nauwkeuriger *driehoek* als volgt gedefinieerd kan worden:

$$\begin{array}{l} driehoek\ xss \\ = \ xs' : \ driehoek\ (map\ (xs'\ \underline{elim})\ xss') \\ \quad\ where \\ \quad\ xs' : \ xss' \ = \ pivot\ xss \end{array}$$

13 Technieken voor recursie

In het voorgaande hebben we 'inductie naar de opbouw van het argument' gezien als één van de vormen van een recursieve definitie. Er zijn veel meer vormen van recursie. Daarvan volgen nu een paar voorbeelden.

§13.1 Wederzijdse recursie. Niet alleen een enkele definitie mag recursief zijn (de gedefinieerde naam weer rechts bevatten), ook een stel definities mogen gezamenlijk recursief zijn. Hier is een heel eenvoudig voorbeeld:

$$
\begin{array}{l}
\mathit{even}\ 0 \ = \ \mathit{True} \\
\mathit{even}\ (n{+}1) \ = \ \mathit{odd}\ n \\
\mathit{odd}\ 0 \ = \ \mathit{False} \\
\mathit{odd}\ (n{+}1) \ = \ \mathit{even}\ n
\end{array}
$$

We zien dat *even* en *odd* elkaar recursief aanroepen.

§13.2 Samengestelde resultaten. Laat f gedefinieerd zijn door:

$$f\ xs \ = \ (\mathit{filter}\ \mathit{even}\ xs,\ \ \mathit{filter}\ \mathit{odd}\ xs)$$

We kunnen f recursief definiëren door:

$$
\begin{array}{l}
f\ [\,] \ = \ [\,] \\
f\ (x : xs) \\
\ = \ (x :\ \mathit{evens},\ \mathit{odds}),\ \mathit{if}\ \mathit{even}\ x \\
\ = \ (\mathit{evens},\ x :\ \mathit{odds}),\ \mathit{if}\ \mathit{odd}\ x \\
\quad\ \mathit{where} \\
\quad\ (\mathit{evens},\ \mathit{odds}) \ = \ f\ xs
\end{array}
$$

Je ziet hier dat $f\ xs$ een samengestelde waarde is, waarvan we de componenten afzonderlijk nodig hadden. Daartoe hebben we de onderdelen in een where-part met een patroon benoemd. Een andere manier is een hulpfunctie *bij* te gebruiken:

$$
\begin{array}{l}
f\ [\,] \ = \ [\,] \\
f\ (x : xs) \ = \ x\ \ \underline{\mathit{bij}}\ \ (f\ xs)
\end{array}
$$

waarbij:

$$
\begin{array}{l}
x\ \ \underline{\mathit{bij}}\ \ (\mathit{evens}, \mathit{odds}) \\
\ = \ (x : \mathit{evens},\ \mathit{odds}),\ \mathit{if}\ \mathit{even}\ x \\
\ = \ (\mathit{evens},\ x : \mathit{odds}),\ \mathit{if}\ \mathit{odd}\ x
\end{array}
$$

Tussen haakjes, het gevalsonderscheid kunnen we vermijden:

$$x\ \underline{bij}\ (evens, odds)\ =\ ([x|\ even\ x] +\!\!+\ evens,\ \ [x|\ odd\ x] +\!\!+\ odds)$$

§13.3 Hulpfuncties, variërende parameters. De standaardoperatie -- is een soort 'lijst-aftrekking': xs -- ys ontstaat uit xs door daaruit, zo mogelijk, ieder eerste voorkomen van de elementen van ys te schrappen. Bijvoorbeeld, voor verschillende a, b, c, d:

$$[a, b, c, a, b, c] \text{ -- } [b, c, b, d] \quad = \quad [a, \cancel{b}, \cancel{c}, a, \cancel{b}, c] \quad = \quad [a, a, c]$$

Het is niet mogelijk om xs -- ys te definiëren met inductie naar een opbouw van xs (met vaste ys), en evenmin met inductie naar een opbouw van ys (met vaste xs). Maar met een geschikte hulpfunctie *zonder* is er geen probleem. We definiëren xs -- ys met inductie naar de prefix opbouw van ys (met *variërende* xs):

$$\begin{array}{l} xs \text{ -- } [\,] \ = \ xs \\ xs \text{ -- } (y : ys) \ = \ (xs\ \ \underline{zonder}\ \ y) \text{ -- } ys \end{array}$$

Nu definiëren we $xs\ \ \underline{zonder}\ \ y$ met inductie naar de prefix opbouw van xs (met vaste y):

$$\begin{array}{l} [\,]\ \ \underline{zonder}\ \ y \ = \ [\,] \\ (x : xs)\ \ \underline{zonder}\ \ y \\ \ = \ xs,\ \ if\ x = y \\ \ = \ x : \ (xs\ \ \underline{zonder}\ \ y),\ \ if\ x \neq y \end{array}$$

In bovenstaande *rechter*leden zijn de haakjes overbodig, omdat in Miranda de infix geschreven '*indentifier*'-operatoren voorrang hebben op alle andere operatoren.

§13.4 Veralgemening: extra parameters. Het komt *vaak* voor dat een functie f niet gemakkelijk met inductie te definiëren is, terwijl een *algemenere* functie f' (met extra parameters) dat wel is. Bijvoorbeeld, laat f de volgende functie zijn:

$$f\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}0,\ \ x_1{\times}1,\ \ \ldots,\ \ x_{n-1}{\times}(n{-}1)]$$

Een definitie van f met inductie naar de prefix opbouw van het argument is lastig. Het gaat wel gemakkelijk voor een algemenere functie f':

$$f'\ k\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}(k{+}0),\ \ x_1{\times}(k{+}1),\ \ \ldots,\ \ x_{n-1}{\times}(k{+}n{-}1)]$$

Immers:

$$\begin{array}{l} f'\ k\ [\,] \ = \ [\,] \\ f'\ k\ (x : xs) \ = \ x{\times}k \ : \ f'\ (k{+}1)\ xs \end{array}$$

En f zelf is een speciaal geval van $f'\ k$:

$$f \ = \ f'\ 0 \qquad\qquad \|\ oftewel:\ \ \ f\ xs \ = \ f'\ 0\ xs$$

Hoe vind je zo'n veralgemening? Door goed te kijken waar de poging om een inductieve definitie te geven stuk loopt! Kijk maar; een poging om f inductief te definiëren leidt tot:

$$\begin{array}{lll} f\ [] & = & [] \\ f\ (x : xs) & = & x{\times}0\ :\ f_1\ xs \end{array}$$

waarbij f_1 wordt gespecificeerd door:

$$f_1\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}1,\ x_1{\times}2,\ \ldots,\ x_{n-1}{\times}n]$$

Een poging om f_1 inductief te definiëren leidt tot:

$$\begin{array}{lll} f_1\ [] & = & [] \\ f_1\ (x : xs) & = & x{\times}1\ :\ f_2\ xs \end{array}$$

waarbij f_2 wordt gespecificeerd door:

$$f_2\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}2,\ x_1{\times}3,\ \ldots,\ x_{n-1}{\times}n{+}1]$$

Een poging om f_2 inductief te definiëren leidt tot:

$$\begin{array}{lll} f_2\ [] & = & [] \\ f_2\ (x : xs) & = & x{\times}2\ :\ f_3\ xs \end{array}$$

enzovoorts. Definieer nu een functie f' door: $f'\ k = f_k$, oftewel:

$$f'\ k\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}(k{+}0),\ x_1{\times}(k{+}1),\ \ldots,\ x_{n-1}{\times}(k{+}n{-}1)]$$

Functie f' is gemakkelijk met inductie te definiëren, zoals we boven gezien hebben.

Vaak ontstaat de algemenere functie f' uit de gegeven functie f door één of meer constanten tot parameter te maken.

§13.5 Veralgemening: extra resultaten. Het komt soms voor dat een functie f niet gemakkelijk met inductie te definiëren is, terwijl een *algemenere* functie f' (met extra resultaten) dat wel is. Bijvoorbeeld, laat f de volgende functie zijn:

$$f\ [x_0, \ldots, x_n] \quad = \quad \text{een } i \text{ met: } x_i \text{ is het maximum van } x_0, \ldots, x_n$$

Deze specificatie zegt niets over f []. Stel, we willen f definiëren met inductie naar de prefix opbouw van z'n argument:

$$\begin{array}{l} f\ [x]\ =\ 0 \\ f\ (x : xs) \\ \ =\ 0,\ \mathit{if}\ \ x\ \geq\ \mathit{max}\ xs \\ \ =\ 1 + f\ xs,\ \mathit{if}\ \ x\ \leq\ \mathit{max}\ xs \end{array}$$

Om de condities in Miranda te formuleren zónder de standaard functie *max* te gebruiken, hebben we méér informatie over *xs* nodig dan f zelf oplevert. We veralgemenen de functie daarom tot f':

$$f'\ [x_0, \ldots, x_n] = \text{ een } (i, m) \text{ met: } x_i = m = \text{maximum van } x_0, \ldots, x_n$$

Functie f' kan nu wel gemakkelijk recursief gedefinieerd worden:

$$
\begin{array}{l}
f'\ [x]\ =\ (0, x) \\
f'\ (x : xs) \\
\ =\ (0, x),\ \mathit{if}\ x \geq m \\
\ =\ (1{+}i, m),\ \mathit{if}\ x \leq m \\
\qquad \mathit{where} \\
\qquad (i, m)\ =\ f'\ xs
\end{array}
$$

en dan:

$$f\ =\ \mathit{fst}\ .\ f' \qquad \|\ \mathit{ofwel}:\ \ f\ xs\ =\ \mathit{fst}\ (f'\ xs)$$

Hierboven heeft ons "inzicht" gesuggereerd om functie f' te definiëren en gebruiken. In feite hebben we de berekening die door *max* wordt opgeroepen, gecombineerd met die van f: de specificatie van f' luidt:

$$f'\ xs\ =\ (f\ xs,\ \ \mathit{max}\ xs)$$

En de definitie van f' volgt nu uit de oorspronkelijke definitie van f en de volgende definitie van *max*:

$$
\begin{array}{l}
\mathit{max}\ [x]\ =\ x \\
\mathit{max}\ (x : xs) \\
\ =\ x,\ \mathit{if}\ x\ \geq\ m \\
\ =\ m,\ \mathit{if}\ x\ \leq\ m \\
\qquad \mathit{where} \\
\qquad m\ =\ \mathit{max}\ xs
\end{array}
$$

§13.6 Inductie op het resultaat. Vaak komt inductie naar een opbouw van een argument voor. Soms komt inductie naar een opbouw van het resultaat voor. Bijvoorbeeld, beschouw de standaard functie *zip* die van een paar van lijsten een lijst van paren maakt ('zip' is engels voor 'ritssluiting'):

$$\mathit{zip}\ ([x_0, \ldots, x_{m-1}],\ [y_0, \ldots, y_{n-1}])\ \ =\ \ [(x_0, y_0),\ \ldots,\ (x_{k-1}, y_{k-1})]$$

waarbij k het minimum is van m en n. Het resultaat van *zip* is altijd van de vorm [] of $\ldots\ :\ \mathit{zip}\ \ldots$. Deze beschouwing leidt tot de volgende definitie-vorm:

$$
\begin{array}{l}
\mathit{zip}\ \ldots\ =\ [] \\
\mathit{zip}\ \ldots\ =\ \ldots\ :\ \mathit{zip}\ \ldots
\end{array}
$$

Het is niet moeilijk de ontbrekende $\ldots$ in te vullen:

$$
\begin{array}{l}
\mathit{zip}\ (x : xs,\ y : ys)\ =\ (x, y) :\ \mathit{zip}\ (xs, ys) \\
\mathit{zip}\ (xs,\ ys)\ =\ []\ \ \|\ \ \mathit{een\ van\ beide\ is\ leeg}
\end{array}
$$

Dit is een definitie met inductie naar de prefix opbouw van het resultaat.

§13.7 Standaard recursievormen. Beschouw de functie *sum*:

$$sum\ [x_0, \ldots, x_{n-1}] \quad = \quad x_0 + (\ldots + (x_{n-1} + 0))$$

of te wel:

$$\begin{aligned} sum\ [] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

Veel functies f hebben deze zelfde definitievorm (met een of andere $\underline{op}$ en a in plaats van $+$ en 0), dus:

$$f\ [x_0, \ldots, x_{n-1}] \quad = \quad x_0\ \underline{op}\ (\ldots\ \underline{op}\ (x_{n-1}\ \underline{op}\ a))$$

of te wel:

$$\begin{aligned} f\ [] &= a \\ f\ (x : xs) &= x\ \underline{op}\ f\ xs \end{aligned}$$

We kunnen a en op tot parameter maken, en krijgen dan een algemenere functie die we *foldr* noemen:

$$foldr\ op\ a\ [x_0, \ldots, x_{n-1}] \quad = \quad x_0\ \underline{op}\ (\ldots\ \underline{op}\ (x_{n-1}\ \underline{op}\ a))$$

oftewel:

$$\begin{aligned} foldr\ op\ a\ [] &= a \\ foldr\ op\ a\ (x : xs) &= x\ \underline{op}\ foldr\ op\ a\ xs \end{aligned}$$

Functie *sum* zelf is nu een speciaal geval van *foldr*:

$$sum \ = \ foldr\ (+)\ 0 \qquad \|\ ofwel: \ \ sum\ xs \ = \ foldr\ (+)\ 0\ xs$$

Veel functies kunnen heel kort met *foldr* gedefinieerd worden; zie de opgaven. De naam '*foldr*' komt van 'fold to the right': de lijst wordt opgevouwen tot één element, met steeds operator *op* tussen de elementen, startwaarde a, en de haakjes naar *rechts* toe gegroepeerd. Functie *foldr* is een standaard functie.

Net zo bestaat er een standaard functie *foldl*: 'fold to the left'. De definitievorm is:

$$f\ [x_0, \ldots, x_{n-1}] \quad = \quad ((a\ \underline{op}\ x_0)\ \underline{op}\ \ldots)\ \underline{op}\ x_{n-1}$$

oftewel:

$$\begin{aligned} f\ [] &= a \\ f\ (xs +\!\!+ [x]) &= f\ xs\ \underline{op}\ x \end{aligned}$$

Bijvoorbeeld, neem $(f, \underline{op}\ , a) = (sum, +, 0)$, en er staat een definitie van sum. De algemenere functie, met op en a als parameter, heet $foldl$:

$$foldl\ op\ a\ [x\_0, \ldots, x\_n{-}1] \quad = \quad ((a\ \underline{op}\ x_0)\ \underline{op}\ \ldots)\ \underline{op}\ x_{n-1}$$

oftewel:

$$\begin{aligned} &foldl\ op\ a\ [\,] \ = \ a \\ &foldl\ op\ a\ (xs +\!\!+ [x]) \ = \ foldl\ op\ a\ xs \ \ \underline{op}\ \ x \end{aligned}$$

Functie f zelf is een speciaal geval van $foldl$:

$$f \ = \ foldl\ op\ a$$

Met name geldt: $sum \ = \ foldl\ (+)\ 0$.

§13.8 Parameteraccumulatie. Een veel voorkomende techniek bij recursieve definities is die van parameteraccumulatie: in een 'extra' parameter wordt stap-voor-stap het gewenste eindresultaat opgebouwd. Bijvoorbeeld, beschouw de standaard functie *reverse* die z'n argument omgekeerd oplevert:

$$reverse\ [x_0, \ldots, x_{n-1}] \quad = \quad [x_{n-1}, \ldots, x_0]$$

Een definitie met inductie naar de postfix opbouw van het argument luidt als volgt:

$$\begin{aligned} &reverse\ [\,] \ = \ [\,] \\ &reverse\ (xs +\!\!+ [x]) \ = \ x : \ reverse\ xs \end{aligned}$$

We definiëren nu een functie *rev* waarbij de omkering van het argument *xs* in een extra parameter *ys* stap-voor-stap wordt opgebouwd, en uiteindelijk opgeleverd:

$$\begin{aligned} &rev\ ys\ [\,] \ = \ ys \\ &rev\ ys\ (x : xs) \ = \ rev\ (x : ys)\ xs \end{aligned}$$

Functie *reverse* zelf is nu een speciaal geval van $rev\ \ldots$:

$$reverse \ = \ rev\ [\,] \qquad\qquad \|\ ofwel: \ \ reverse\ xs \ = \ rev\ [\,]\ xs$$

Berekeningen volgens de recursie-vorm $f\ x \ = \ f\ (\ldots)$, zoals bij *rev*, zijn soms efficiënter dan volgens recursie-vormen zoals $f\ x \ = \ \ldots\ (f\ \ldots)\ \ldots$, zoals bij *reverse*. Een definitie zonder parameteraccumulatie is meestal veel duidelijker.

In feite hebben we functie *rev* als volgt verzonnen. Stel dat je de uitkomst van *reverse* voor zeker argument wil uitdrukken als een functie, zeg *rev*, van een staartstuk *xs* van dat argument:

$$reverse\ (ys +\!\!+ xs) \ = \ rev\ \ldots\ \ xs$$

Wat moet *rev* dan nog méér kennen dan *xs* opdat-ie de gewenste uitkomst kan opleveren? Iets over *ys*, natuurlijk. En voldoende informatie voor *rev* is de kennis van de omgekeerde van *ys*. Dus we specificeren *rev* door:

$$rev\ ys'\ xs\ =\ reverse\ (ys \mathbin{+\!\!+} xs) \qquad MITS\ ys'\ =\ reverse\ ys$$

Ietwat korter geformuleerd:

$$rev\ (reverse\ ys)\ xs\ =\ reverse\ (ys \mathbin{+\!\!+} xs)$$

De gegeven definitie van *rev* volgt nu uit deze specificatie en de al bekende definitie van *reverse*. En ook volgt dan dat $reverse\ xs\ =\ rev\ []\ xs$.

Opgaven

126. Definieer een functie f zó dat een afdruk van $f\ [x_0, \ldots, x_{n-1}]$ bestaat uit n regels met op de $i$de regel: i spaties en dan x_i. Gebruik geen lijstcomprehensie, maar alleen recursie (inductie).
127. Geef een recursieve definitie van *unzip*; gebruik inductie naar de prefix opbouw van het argument. (Vergelijk met opgaven 59 en 105.)
128. Een string die louter uit haakjes bestaat, heet *goedgevormd* als er, door herhaaldelijk delen '()' weg te laten, uiteindelijk niets overblijft. Voorbeeld:

"(()()(()))"	goedgevormd
"(()()()))"	niet goedgevormd

Definieer het predicaat *isGoed* die aangeeft of z'n argument goedgevormd is. Wenk: laat een extra parameter aangeven *hoeveel* sluithaakjes er nog moeten komen.
129. Het zelfde als in de vorige opgave, maar nu met verscheidene soorten haakjes. Wenk: laat de extra parameter aangeven welke sluithaakjes (en in *welke* volgorde) er nog moeten komen; bijvoorbeeld: "]))" geeft aan dat er éérst nog een ']' moet komen en dan nog een ')' en een ')'.
130. Een *regel* van *xs* is een zo groot mogelijk segment van *xs* dat geen newlines bevat. Standaard functie *lines* levert de lijst van regels van *xs*; preciezer gezegd, als elk van $xs_0, xs_1, \ldots, xs_{n-1}$ geen newline-karakter bevat, dan:

$$lines\ (xs_0 \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"} \mathbin{+\!\!+} xs_1 \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} xs_{n-1} \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"}) = [xs_0, xs_1, \ldots, xs_{n-1}]$$

Geef een recursieve definitie van *lines*. Doe dit één keer met patronen [] en $x : xs$ (inductie), en één keer met met gebruik van *takewhile* en *dropwhile* en de patronen [] en *xs*.
Wat is het resultaat van *lines xs* als *xs* niet eindigt op $'\backslash n'$?
Onder welke voorwaarden gelden de volgende vergelijkingen:

$$\begin{aligned} lay\ (lines\ xs) &= xs \\ lines\ (lay\ xss) &= xss \end{aligned}$$

131. Een *woord* van *xs* is een zo groot mogelijk niet-leeg segment van *xs* dat geen spaties of newlines bevat. Functie *words* levert de lijst van woorden van *xs*. Een precieze definitie luidt als volgt:

$$words\ (sep_0 \mathbin{+\!\!+} xs_0 \mathbin{+\!\!+} sep_1 \mathbin{+\!\!+} xs_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} xs_{n-1} \mathbin{+\!\!+} sep_n) = [xs_0, xs_1, \ldots, xs_{n-1}]$$

wanneer ieder van $xs_0, \ldots, xs_{n-1}$ niet-leeg is en géén spatie of newline bevat, en ieder van $sep_0, \ldots, sep_n$ alléén maar spaties en newlines bevat, en $sep_1, \ldots, sep_{n-1}$ niet-leeg zijn.
Geef een recursieve definitie van *words*. Doe dit één keer met inductie naar de prefix opbouw, en één keer met gebruik van *takewhile* en *dropwhile*.
Geef ook een definitie van *words* op de volgende manier: wijzig in de definitie van *lines* de test 'is het newline-karakter' in de test 'is het spatie- of het newline-karakter'. De resulterende functie levert dan niet alleen alle woorden, maar ook nog "lege woorden". De echte woorden krijg je dan door de "lege woorden" er uit te filteren.

132. Geef met behulp van functiecompositie en functies *lines* en *words* uit de vorige opgaven (en algemene functies zoals *map* en *filter* en *concat*) definities voor:

    $f\ xs$ = een lijst van woord-lijsten: alle woorden van xs, gegroepeerd per regel;
    $g\ xs$ = de lijst van alle woorden van xs;
    $h\ xs$ = het aantal woorden in xs;
    $fmt\ xs$ = een lijst van karakters: met "dezelfde" regels als xs maar waarin de woorden onderling gescheiden zijn door één enkele spatie.

133. In §13.4 werd voor de volgende functie een recursieve definitie gegeven:

$$f\ [x_0, x_1, \ldots, x_{n-1}] \quad = \quad [x_0{\times}0,\ x_1{\times}1,\ \ldots,\ x_{n-1}{\times}(n{-}1)]$$

    Definieer f nu eenmaal met behulp van lijstcomprehensie, en eenmaal met behulp van *zip* of *zipwith*.

134. Geef een definitie van *fib* zó dat de berekening opgeroepen door *fib* n slechts evenredig toeneemt met n. Doe dit eenmaal door twee functies *fib* te combineren, en eenmaal door *fib* te veralgemenen met twee extra parameters.

135. Een *plateau* van xs is een zo lang mogelijk segment van xs waarvan alle elementen aan elkaar gelijk zijn. Definieer de functie *plateaus* die de lijst van alle plateaus van z'n argument oplevert. Bijvoorbeeld: *plateaus* $[3, 3, 2, 2, 2, 4, 1, 1, 1, 4, 4, 4] = [[3, 3], [2, 2, 2], [4], [1, 1, 1], [4, 4, 4]]$.

136. Definieer de functie *llp* (lengte langste plateau) met:

$$llp \;=\; max\ .\ map\ (\#)\ .\ plateaus$$

    Hierbij is *plateaus* de functie van opgave 135, maar die mag in de definitie van *llp* niet gebruikt worden. Er zijn velerlei definities van *llp* mogelijk.

137. Definieer de standaard functie *max* volgens de volgende werkwijze. Van het argument $[x_0, x_1, \ldots]$ wordt een nieuwe, half-zo-lange, lijst gemaakt: $[x_{0,1}, x_{2,3}, \ldots]$ met $x_{i,j} = x_i\ \underline{max2}\ x_j$; deze wordt weer aan deze zelfde werkwijze onderworpen, totdat de lijst één lang is...
    Merk op dat dit de methode is om bij toernooien de winnaar te bepalen: in iedere ronde valt de helft van de deelnemers af.

138. Een *keten* is een eindige rij van woorden van gelijke lengte, waarvoor geldt dat elk woord op precies één positie van zijn buren verschilt. Voorbeeld:

    ["*groot*", "*groet*", "*griet*", "*grief*", "*brief*"]

    Definieer de functie *isKeten* met: *isKeten* xss = 'xss is een keten'.
    Doe dit zowel recursief, als ook niet-recursief (met *map*, *zip* etc).

139. Een *ketting* is (in deze opgave) een eindige rij van niet-lege woorden (karakterrijen) waarvoor het volgende geldt:

voor elk tweetal opeenvolgende woorden, wordt het tweede verkregen uit het eerste door precies m letters aan het begin te schrappen en precies n letters aan het eind toe te voegen, waarbij m en n hoogstens 5 zijn en hooguit 2 van elkaar verschillen.

Definieer het predicaat *isKetting* dat aangeeft of zijn argument een ketting is. Doe dit zowel recursief, als ook niet-recursief (met *map*, *zip* etc).
Wenk. Definieer eerst een hulpfunctie *magVoor* zó dat v $\underline{magVoor}$ w precies dan *True* is wanneer v in een ketting direct aan w vooraf mag gaan.

140. Standaard functie *merge* mengt twee gesorteerde rijen tot één gesorteerde rij. De specificatie luidt:

$$merge\ (sort\ xs)\ (sort\ ys) \;=\; sort\ (xs +\!\!+\ ys)$$

Er wordt hiermee niets gezegd over *merge xs ys* wanneer *xs* of *ys* niet gesorteerd zijn. Geef een efficiënte definitie van *merge*. Wenk: gebruik inductie naar de prefix opbouw van het resultaat, net als bij *zip*.

141. Geef een efficiënte definitie van *merge′* die net als *merge* twee gesorteerde rijen tot één gesorteerde rij mengt, maar bovendien dubbele voorkomens verwijdert:

$$merge'\ (sort\ (mkset\ xs))\ (sort\ (mkset\ ys)) \;=\; sort\ (mkset\ (xs +\!\!+\ ys))$$

Standaard functie *mkset* is een inefficiënte methode om dubbele voorkomens te verwijderen. Let op: de specificatie van *merge′* zegt niets over de uitkomst van *merge′ xs ys* wanneer *xs* of *ys* zelf al dubbele voorkomens bevat of niet gesorteerd is.

142. Representeer verzamelingen als lijsten waarin de elementen gerangschikt zijn naar opklimmende grootte en niet dubbel voorkomen. Geef voor deze representatie de definities van: vereniging $\cup$, doorsnee $\cap$, symmetrisch verschil $\uplus$, verwijdering $\setminus$, isDeel $\subseteq$, isIn $\in$:

$$\begin{array}{lcl}
\{1,2,3,4,5\} \cup \{3,4,6\} & = & \{1,2,3,4,5,6\} \\
\{1,2,3,4,5\} \cap \{3,4,6\} & = & \{3,4\} \\
\{1,2,3,4,5\} \uplus \{3,4,6\} & = & \{1,2,5,6\} \\
\{1,2,3,4,5\} \setminus \{3,4,6\} & = & \{1,2,5\} \\
\{1,2,3,4,5\} \subseteq \{3,4,6\} & = & \mathit{False} \\
3 \in \{3,4,6\} & = & \mathit{True}
\end{array}$$

143. Definieer met behulp van *foldr* en *foldl*: *sum*, *product*, *and*, *or*, #, *concat*.

144. Definieer *reverse* in de vorm:

$$reverse\ [x_0, \ldots, x_{n-1}] \;=\; ((a\ \underline{op}\ x_0) \ldots)\ \underline{op}\ x_{n-1}$$

Bedenk daartoe wat *op* en *a* moeten zijn. Definieer *reverse* vervolgens als een speciaal geval van *foldl* of *foldr*.

145. Definieer de functie *rev ys* (een functie van *xs*; zie §13.8) als een *foldl* of *foldr*.

146. Er zijn nog meer speciale vormen van recursie die veel voorkomen, en waarvoor het de moeite loont om aparte functies, zoals *foldr* en *foldr*, te definiëren. Beschouw de volgende specificatie:

$$f\ x \;=\; hd\ (dropwhile\ (\leq 999)\ [x,\ 2\times\ x,\ 2\times\ (2\times\ x),\ 2\times\ (2\times\ (2\times\ x)),\ \ldots\ ])$$

Dus $f\ 1$ is de kleinste tweemacht groter dan 999. Definieer f zonder enige andere functies te gebruiken dan $(2\times)$ en (> 999); gebruik recursie.
Veralgemeen f door $(2\times)$ en (> 999) tot parameter te maken (en de veralgemening *until* te noemen); geef de definitie van *until*. Druk de volgende functies hiermee uit:

$$\begin{array}{lcl} \textit{spatiesWeg}\ xs & = & \text{de karakterlijst } xs \text{ met weglating van de spaties vóóraan} \\ \textit{dropwhile}\ p\ xs & = & xs \mathbin{-\!-} \textit{takewhile}\ p\ xs \end{array}$$

Om te onthouden

• Om een functie met recursie te definiëren is het soms nodig om hem te veralgemenen met een extra parameter. In zo'n extra parameter kan extra informatie "van buiten (de aanroep) naar binnen" worden gebracht.

• Om een functie met recursie te definiëren is het soms nodig om hem te veralgemenen met een extra resultaat. In zo'n extra resultaat kan extra informatie "van binnen naar buiten" worden gebracht.

• Parameter-accumulatie is de techniek om in een extra parameter het functie-resultaat beetje-bij-beetje op te bouwen.

• Functies die met gelijke inductie worden gedefinieerd, kunnen tot één functie (met een tupel als resultaat) gecombineerd worden. Dat verbetert de efficiëntie.

• Wanneer een functie met een tupel als resultaat recursief gedefinieerd wordt, worden de onderdelen van de recursieve aanroepen vaak in een where-part benoemd.

• Functies *foldl* en *foldr* geven een veel voorkomende, eenvoudige vorm van recursie weer. Met name:

$$\begin{array}{lcl} \textit{foldl}\ \textit{op}\ a\ [x_0, x_1, x_2, x_3, x_4, \ldots] & = & (((((a \ \underline{\textit{op}}\ x_0)\ \underline{\textit{op}}\ x_1)\ \underline{\textit{op}}\ x_2)\ \underline{\textit{op}}\ x_3)\ \underline{\textit{op}}\ x_4)\ldots \\ \textit{foldr}\ \textit{op}\ a\ [x_0, x_1, x_2, x_3, x_4, \ldots] & = & (x_0\ \underline{\textit{op}}\ (x_1\ \underline{\textit{op}}\ (x_2\ \underline{\textit{op}}\ (x_3\ \underline{\textit{op}}\ (x_4 \ldots\ \underline{\textit{op}}\ a))))) \end{array}$$

14 Sorteren

Sorteren is het rangschikken, volgens de $\leq$-ordening, van de elementen in een lijst. Veel problemen kunnen efficiënt opgelost worden door eerst efficiënt te sorteren. Sorteren kan op diverse manieren uitgedrukt worden, sommige heel efficiënt.

§14.1 Nut van sorteren. Het moet in dit stadium niet moeilijk zijn om functies te definiëren voor:

het aantal gemeenschappelijke elementen in xs en ys,
het aantal verschillende elementen in xs,
de *mediaan* van xs, dat is: qua grootte het middelste element van xs,

enzovoorts. Bij voor de hand liggende definities groeit het aantal benodigde rekenstappen kwadratisch met $\#xs$ en $\#ys$: bij twee keer zo lange xs zijn vier maal zo veel rekenstappen nodig. Maar het kan veel efficiënter: door éérst de rijen te sorteren, en dan —gebruik makend van de sortering— de uitkomst te bepalen op een manier waarvan het aantal benodigde rekenstappen lineair groeit met $\#xs$ en $\#ys$. De nu extra benodigde sorteer-slag behoeft niet veel tijd te kosten: er zijn sorteer-algoritmen waarvan het aantal rekenstappen in de orde van grootte van $n \; log \; n$ is, met $n = \#xs$; en dat is voor toenemende n veel minder dan n^2.

§14.2 Specificatie. In Miranda zijn de vergelijkingsoperatoren $<, \leq$ enzovoorts op alle waarden gedefinieerd. Functie *sort* sorteert naar de standaardvergelijking:

$$\begin{aligned} &sort &&:: \; [\alpha] \to [\alpha] \\ &sort\ xs &&= \text{ de permutatie } [y_0, \ldots, y_{n-1}] \text{ van } xs \text{ zó dat } y_0 \leq y_1 \leq \cdots \leq y_{n-1} \end{aligned}$$

We zullen hier een aantal definities van *sort* ‘verzinnen’, en de standaard namen ervoor geven. De definities zijn nog niet af; de voltooiing van de definities komt in de opgaven aan bod.

(a) Definieer *sort* op zo’n manier dat uiteindelijk de gewenste situatie bereikt is:

$$\forall i, j : \qquad i \leq j \Rightarrow x_i \leq x_j$$

We doen dit door het aantal ‘foute’ paren x_i, x_j te verminderen.

Bij bubble sort worden steeds twee naburige elementen x, y indien nodig verwisseld. Daarmee komt de grootste ‘bovendrijven’ (naar rechts), als een belletje in het water, en staat het meest rechtse element dus op z’n plaats. Na voldoende veel herhalingen, op steeds kleinere beginstukken, staat alles op z’n plaats:

$$\begin{aligned} &bubble\ (x : y : zs) \\ &= x :\ bubble\ (y : zs),\ if\ x \leq y \\ &= y :\ bubble\ (x : zs),\ if\ y \leq x \\ &sort\ xs \;=\; sort\ xs' \;+\!\!+\; [x'] \quad where\ \ xs' +\!\!+ [x'] \;=\; bubble\ xs \end{aligned}$$

Bij shell sort worden steeds ver uit elkaar liggende elementen x, y indien nodig verwisseld; verder gaat het als bij bubble sort. De idee hierachter is dat verwisselingen over grote afstand sneller effect hebben. We werken shell sort hier niet uit.

(b) Definieer *sort* met inductie naar de opbouw van het argument. Inductie naar de prefix opbouw leidt tot insertion sort:

$$sort\ (x : xs)\ =\ x\ \underline{insert}\ sort\ xs$$

waarbij operatie $\underline{insert}$ het element x op de juiste plaats in *sort xs* zet.

Inductie naar de $+\!\!+$-opbouw leidt tot merge sort:

$$sort\ (xs +\!\!+ ys)\ =\ sort\ xs\ \underline{merge}\ sort\ ys$$

Functie *merge* is al behandeld in opgave 140.

(c) Definieer *sort* met inductie naar de opbouw van het resultaat. Inductie naar de prefix opbouw leidt tot selection sort:

$$sort\ xs\ =\ m\ :\ sort\ (xs -\!\!- [m])\quad where\ m\ =\ min\ xs$$

In iedere recursie-stap wordt uit het argument precies dát element geselecteerd dat op kop van de resultaatlijst komt te staan.

Inductie naar de $+\!\!+$-opbouw leidt tot quick sort:

$$sort\ xs\ =\ sort\ (kleintjes\ xs)\ +\!\!+\ sort\ (groten\ xs)$$

Voor iedere k uit *kleintjes xs* en iedere g uit *groten xs* moet gelden $k \leq g$. We kunnen zoiets als *filter* $(<\ hd\ xs)\ xs$ nemen voor *kleintjes xs*, en analoog voor *groten xs*.

Opgaven

147. Voltooi de definitie van bubble sort. Let er op dat *sort* ook moet werken voor lijsten met lengte nul of één, en dat $+\!\!+$ niet in een patroon is toegestaan. Hoeveel vergelijkingen '... <...' kost de berekening van *sort xs* (uitgedrukt in $\#xs$)?
Schrijf de definitie van *bubble* zonder gevalsonderscheid; gebruik $min2$ en $max2$.
148. Voltooi de definitie van insertion sort. Definieer de insertion operatie $\underline{insert}$ eenmaal met behulp van *takewhile* en *dropwhile*, en eenmaal met inductie naar de prefix opbouw van het argument. Hoeveel vergelijkingen kost de berekening van *sort xs* (uitgedrukt in $\#xs$)?
149. Voltooi de definitie van merge sort. Let er op dat *sort* ook moet werken voor lijsten met lengte nul of één, en dat $+\!\!+$ niet in een patroon is toegestaan. Neem voor *xs* en *ys* de eerste en laatste helft van het argument van *sort*. Hoeveel vergelijkingen kost de berekening van *sort xs* (uitgedrukt in $\#xs$)?
150. Voltooi de definitie van selection sort. Denk er aan dat de standaard functie *min* alleen gedefinieerd is voor niet-lege lijsten. Hoeveel vergelijkingen kost de berekening van *sort xs* (uitgedrukt in $\#xs$)?
151. Voltooi de definitie van quick sort. Let op: opdat iedere berekening volgens deze definitie eindigt, moeten zowel *kleintjes xs* als ook *groten xs* minder elementen bevatten dan *xs* zelf. Wenk: wijzig het rechterlid in zoiets als $sort\ (kleintjes\ xs)\ +\!\!+\ [hd\ xs]\ +\!\!+\ sort\ (groten\ xs)$. Hoeveel vergelijkingen kost de berekening van *sort xs* (uitgedrukt in $\#xs$)? Beantwoord deze vraag twee keer; één keer waarbij je aanneemt dat steeds de lijsten *kleintjes xs* en *groten xs*

ongeveer even lang zijn, en één keer waarbij je aanneemt dat de te sorteren rij al geheel gesorteerd is.

152. Geef een efficiënte definitie voor de functie min' met specificatie:

$$min'\ n\ xs \quad = \quad sort\ xs\ !\ n$$

Doe dit zonder de hele lijst te sorteren. Wenk: pas de definitie van quick sort aan.
Druk hiermee uit: het kleinste element, het een-na-kleinste element, de mediaan. Hoeveel rekenstappen kosten deze berekeningen, en hoeveel rekenstappen vergt de berekening van $sort\ xs\ !\ n$?

153. Geef een efficiënte definitie van *mkset* voor eindige lijsten. De specificatie luidt:

(1) $x \in mkset\ xs \quad \equiv \quad x \in xs$

(2) $mkset\ xs$ heeft geen dubbele voorkomens

Wenk: enerzijds kan dit door aanpassing en wijziging van de definitie van quick sort, en anderzijds door éérst te sorteren en dán de dubbelen te verwijderen.

154. Definieer de volgende functies:

$$\begin{array}{lcl} f\ xs & = & \text{het aantal verschillende elementen in } xs \\ g\ xs\ ys & = & \text{het aantal gemeenschappelijke elementen in } xs \text{ en } ys \\ h\ xs & = & \text{het element } x \text{ dat qua grootte het middelste is van } xs \\ & = & \text{de mediaan van } xs \end{array}$$

Formuleer de definities zó dat het aantal vergelijkingen ($x < y$ of $x = y$) in de grootte-orde van $n\ log\ n$ is, waarbij $n =$ de lengte van het argument.

155. Definieer de standaard functie *transpose* die van een 'kolom van rijen' een 'rij van kolommen' maakt:

$$transpose\ [[1,2,3],\ [4,5,6]]\ =\ [[1,4],\ [2,5],\ [3,6]]$$

Algemener:

$$transpose\ [xs, ys, \ldots, zs] \quad = \quad [[x_0, y_0, \ldots, z_0], \ldots, [x_{n-1}, y_{n-1}, \ldots, z_{n-1}]]$$

Ga er van uit dat het argument van *transpose* een niet-lege lijst is van evenlange lijsten. Doe het twee keer: eenmaal met inductie naar de opbouw van het argument (dit is al gedaan in opgave 120), en eenmaal met inductie naar de opbouw van het resultaat.

Om te onthouden

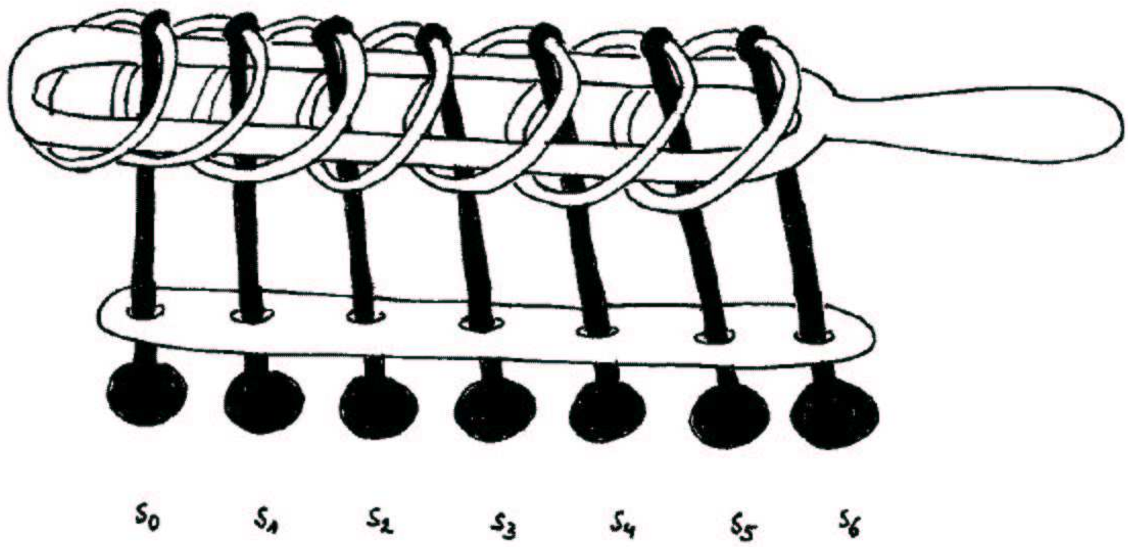
• Sommige problemen kunnen efficiënt opgelost worden door eerst efficiënt te sorteren, en dan een recht-toe recht-aan algoritme toe te passen.

• Een efficiënt sorteer-algoritme kost, in grootte-orde, $n \log n$ rekenstappen, waarbij n het aantal te sorteren element is. De sleutel tot deze efficiëntie is de 'verdeel-en-heers' aanpak: de te sorteren collectie wordt in twee ongeveer even grote delen verdeeld, die onafhankelijk van elkaar gesorteerd worden.

• Bekende efficiënte sorteer-algoritmen zijn *quicksort, merge sort.*

15 Voorbeeld: de Chinese Ringen

§15.1 De puzzel. Bekijk de figuur voor een afbeelding van het puzzelding.



Zoals je ziet bestaat het puzzelding uit een rij van zeven verticale staafjes, van links naar rechts $s_0, \ldots, s_6$ genoemd. Ze zijn aan de onderzijde met iets aan elkaar verbonden dat geen belemmering vormt voor verticale bewegingen van de staafjes. Aan de bovenzijde is ieder staafje scharnierend vastgemaakt aan een eigen ring. Elk staafje s_i (behalve de laatste) gaat door de ring van de rechterbuur s_{i+1}. Bovendien is er een langwerpige horizontale 'ellips', met een handvat aan de rechterkant. De verticale staafjes s_i gaan door de opening van de ellips, terwijl de ringen óm de ellips heen gaan (de ellips zelf gaat dus dóór alle ringen heen).

§15.2 De opgave. De bedoeling is om door schuiven en bewegen alle staafjes **uit** de ellips te krijgen (de ellips dus uit de ringen); en daarna weer **in** de ellips.

§15.3 Historie. Deze puzzel wordt in China *Ryou-Kaik-Tjyo* genoemd, oftewel *voorwerp dat gast doet blijven.* Het is één van de oudste mechanische puzzels. Het is niet geheel duidelijk waar hij oorspronkelijk vandaan komt. Volgens de Chinese legende werd hij uitgevonden door de militaire held Hung Ming, die leefde van 181 tot 234 en hem gaf aan zijn vrouw toen hij naar het front moest. Zij was zó intensief bezig de oplossing te vinden dat ze vergat te treuren over de afwezigheid van haar man. In Europa verscheen de puzzel in het midden van de 16de eeuw toen hij werd beschreven door de Italiaanse wiskundige Cardano, ook bekend van de cardan-as. In 1693 werd de puzzel in een Engels boek vermeld en omstreeks die tijd kreeg hij bekendheid in vele landen. De puzzel wordt ook wel de Meleda puzzel genoemd.

§15.4 De oplossing. We gaan nu de oplossing van de puzzel in Miranda beschrijven.
Stap 0: de probleem-analyse. Uit *de afbeelding* van het puzzelding concluderen we het volgende (het daadwerkelijk in de hand hebben van het puzzelding helpt nauwelijks):

Is de ring van een staafje s_i geheel bóven de ellips (in plaats van er om heen), dan kan hij dóór de ellips heen geschoven worden, en is staafje s_i daarmee **uit** de ellips.

a. De ring van s_0 kan eenvoudig aan de linkerkant geheel boven de ellips gebracht worden; daarmee is s_0 dus **uit** de ellips te halen.

b. De ring van s_{i+1} kan aan linkerkant geheel boven de ellips gebracht worden, mits staafjes $s_0, \ldots, s_{i-1}$ **uit** de ellips zijn en s_i er **in** zit; daarmee is s_{i+1} dus **uit** de ellips te halen.

Door de handelingen in in tegengestelde richting, uit te voeren krijg je een staafje weer **ín** de ellips.

Stap 1: de representatie. We representeren de oplossing door een lijst van getallen $-i$ en $+i$. De reeks lijstelementen geeft de reeks handelingen aan; een getal $-i$ betekent: breng staafje s_i **uit** de ellips volgens (a) of (b) van Stap 0. Analoog voor $+i$.

In feite zijn de tekens $-$ en $+$ overbodig, omdat nooit handelingen $-i$ en $+i$ beide tegelijk toepasbaar zijn. Da's maar goed ook, want -0 en $+0$ zijn hetzelfde.

Stap 2: eerste veralgemening. We schrijven een *functie uit* die bij argument n de handelingenreeks produceert om $s_0, \ldots, s_{n-1}$ **uit** de ellips te halen (in de veronderstelling dat deze er allemaal **in** zitten).

Waarom deze veralgemening? Nou, ehh, uit ervaring vermoed ik dat het gemakkelijker is om zo'n functie met recursie (inductie) te definiëren dan de hele oplossingslijst in één keer uit te drukken. Merk op dat we de constante 7 hebben veralgemeend tot een parameter n. En heb je nog geen ervaring, dan zal je zien —als je goed oplet— dat de oplossing voor de zeven staafjes gemakkelijk(er) kan worden uitgedrukt in termen van de oplossing voor zes staafjes, enzovoorts.

Stap 3: tweede veralgemening. We schrijven niet alleen de functie *uit* maar ook een functie *in*; de uitkomst van *in* n is de handelingenreeks om $s_0, \ldots, s_{n-1}$ **in** de ellips te brengen (in de veronderstelling dat ze er allemaal **uit** zijn).

Waarom deze veralgemening? Nou, ehh, ervaren als ik ben, zie ik onmiddellijk aan de hand van de probleem-analyse in Stap 0 dat deze functie ook wel nodig zal zijn. En heb je nog geen ervaring, dan zal je zien —als je goed oplet— dat *uit* n gemakkelijk(er) kan worden uitgedrukt in termen van de oplossing voor '**in**' voor kleinere argumenten.

Stap 4: inductieve definitie. Ter herinnering de specificatie van $-i$ en *uit* n:

> $-i$ representeert de handeling om s_i **uit** de ellips te brengen, onder de voorwaarde dat $i = 0$, of: $i > 0$ en $s_0, \ldots, s_{i-2}$ zijn **uit** de ellips en s_{i-1} is er **in**.
>
> *uit* n is de handelingenreeks om $s_0, \ldots, s_{n-1}$ **uit** de ellips te brengen, onder de voorwaarde dat ze bij de start van de reeks allemaal **in** de ellips zitten.

En analoog voor $+i$ en *in* n.

Nu proberen we *uit* n uit te drukken in termen van de handelingen $-i$. Gevallen *uit* 0 en *uit* 1 zijn eenvoudig:

$$\begin{aligned} \mathit{uit}\ 0 &= [] \\ \mathit{uit}\ 1 &= [-0] \end{aligned}$$

Voor *uit* $(n{+}2)$ gebruiken we *uit* en *in* voor kleinere argumenten:

$$uit\ (n{+}2) \;=\; uit\ n \mathbin{+\!\!+} [-(n{+}1)] \mathbin{+\!\!+} in\ n \mathbin{+\!\!+} uit\ (n{+}1)$$

We tonen nu aan dat in het rechterlid de voorwaarden uit de specificaties vervuld zijn en de gespecificeerde eindsituatie bereikt wordt, als tenminste de voorwaarde voor het linkerlid vervuld is:

> Direct aan het begin van het rechterlid zijn, volgens de voorwaarde van het linkerlid, staafjes $s_0, \ldots, s_{n+1}$ er allemaal **in**; de voorwaarde voor *uit n* is vervuld.
>
> Na de reeks *uit n* zijn staafjes $s_0, \ldots, s_{n-1}$ **uit** de ellips en s_n, s_{n+1} nog er **in**; de voorwaarde voor '$-(n{+}1)$' is vervuld.
>
> Na de handeling '$-(n{+}1)$' zijn $s_0, \ldots, s_{n-1}$ er **uit**, is s_n er nog **in**, en is s_{n+1} er al **uit**; de voorwaarde voor *in n* is vervuld.
>
> Na de reeks *in n* zijn staafjes $s_0, \ldots, s_{n-1}$ en s_n er allemaal **in**, en is staafje s_{n+1} er nog **uit**; de voorwaarde voor *uit* $(n{+}1)$ is vervuld.
>
> Na de reeks *uit* $(n{+}1)$ zijn staafjes $s_0, \ldots, s_n$ en s_{n+1} er allemaal **uit**; de gespecificeerde eindsituatie van het linkerlid is vervuld.

In feite is dit geen redenering achteraf, maar heeft deze redenering mede geleid tot bovenstaande definitie. Met andere woorden, niet alleen 'goede ingevingen' maar ook de specificaties hebben sturing gegeven aan de manier waarop we de handelingenreeks *uit* $(n{+}2)$ hebben samengesteld uit aanroepen van *uit* en *in* en $-i$ en $+i$. We laten het aan de lezer over om, geheel analoog, functie *in* te definiëren.

Stap 5: de oplossing. De gevraagde oplossing is nu een speciaal geval van wat we hierboven gedefinieerd hebben:

$$uit\ 7$$

§15.5 Nabeschouwing 1. Uit hoeveel handelingen bestaat *uit n*? Het antwoord is eenvoudig: $aantal\ n = \#uit\ n = \#in\ n$. Maar dat kan ook efficiënter uitgerekend worden. We leiden uit de inductieve definitie van *uit* een andere definitie voor *aantal* af:

$$\begin{aligned}
\#uit\ 0 \quad &= \; \#[] \\
\#uit\ 1 \quad &= \; \#[-0] \\
\#uit\ (n{+}2) \quad &= \; \#(uit\ n \mathbin{+\!\!+} [-(n{+}1)] \mathbin{+\!\!+} in\ n \mathbin{+\!\!+} uit\ (n{+}1)) \\
&= \; \#uit\ n \;+\; \#[-(n{+}1)] \;+\; \#in\ n \;+\; \#uit\ (n{+}1)
\end{aligned}$$

Dus:

$$\begin{aligned}
&aantal\ 0 \;=\; 0 \\
&aantal\ 1 \;=\; 1 \\
&aantal\ (n{+}2) \;=\; aantal\ n + 1 + aantal\ n + aantal\ (n{+}1)
\end{aligned}$$

Deze definitie van *aantal* geeft efficiëntere berekeningen dan de definitie $aantal\ n = \#uit\ n$, omdat het opbouwen en daarna weer doorlopen van de (hele grote!) lijst van handelingen nu achterwege blijft. Uit de derde clausule blijkt dat *aantal* $(n+2)$ groter is dan tweemaal *aantal* n, dus het aantal handelingen in de reeks *uit* n neemt exponentioneel toe met n.

§15.6 Nabeschouwing 2. We willen een nette afdruk van de achtereenvolgende posities van de staafjes, gedurende het uitvoeren van de handelingen *uit* n. Bijvoorbeeld, voor *uit* 4:

```
 1) + + + +
 2) + - + +
 3) - - + +
 4) - - + -
 5) - + + -
 6) + + + -
 7) - + + -
 8) - + - -
 9) + + - -
10) + - - -
11) - - - -
```

Per regel staat er één toestand genoteerd (de posities van staafjes s_0, s_1, s_2, s_3); een `+` voor 'erin' en een `-` voor 'eruit'. In het programma representeren we een toestand t van de rij staafjes met een rij boolean waarden; de begintoestand is $t0\ n\ =\ [True, True, \ldots, True]$. Door *showt* wordt zo'n toestand t (de rij van posities) getoond als een rij van + en − tekens:

$$
\begin{array}{l}
toestand\ \equiv\ [bool] \\
t0\ ::\ num \rightarrow toestand \\
t0\ n\ =\ rep\ n\ True \\
showt\ ::\ toestand \rightarrow [char] \\
showt\ =\ concat\ .\ map\ shows \\
shows\ True\ =\ \text{"+ "} \\
shows\ False\ =\ \text{"− "}
\end{array}
$$

Functie *doe* voert één handeling uit:

$$
\begin{array}{l}
handeling\ \equiv\ num \quad \|\ +i\ \ of\ \ -i\ \ \ voor\ i\ in\ 0\ ..\ n{-}1 \\
doe\ ::\ toestand \rightarrow handeling \rightarrow toestand \\
t\ \underline{doe}\ i\ =\ take\ i'\ t\ +\!\!+\ [\neg\ t!i']\ +\!\!+\ drop\ (i'{+}1)\ t \quad where\ i'\ =\ abs\ i
\end{array}
$$

De gewenste afdruk wordt door opgeleverd door functie *verloop*:

$$
\begin{array}{l}
verloop\ ::\ num \rightarrow [char] \\
verloop\ n\ =\ (layn\ .\ map\ showt\ .\ scan\ doe\ (t0\ n)\ .\ uit)\ n
\end{array}
$$

Hierbij is *scan* de standaard functie met het volgende effect:

$$
\begin{array}{l}
scan\ f\ a\ [x_0, x_1, x_2, \ldots]\ \ =\ \ [y_0, y_1, y_2, \ldots] \\
\text{waarbij} \\
y_0\ \ = a
\end{array}
$$

$$
\begin{aligned}
y_1 &= a \underline{f} x_0\\
y_2 &= ((a \underline{f} x_0) \underline{f} x_1)\\
y_3 &= (((a \underline{f} x_0) \underline{f} x_1) \underline{f} x_2)\\
&\vdots\\
y_{i+1} &= ((((a \underline{f} x_0) \underline{f} x_1) \underline{f} x_2) \ldots) \underline{f} x_i\\
&= y_i \underline{f} x_i
\end{aligned}
$$

Opgaven

156. Geef een definitie van functie *in*. Doe dit eenmaal analoog aan de definitie van *uit*, en eenmaal met behulp van *uit* en *reverse*. Welk van beide definities geeft efficiëntere berekeningen (en waarom)?

157. Stel *uit* en *in* samen tot een functie uit' waarvoor geldt: $uit'\ m\ n$ is de handelingenreeks om $s_m, \ldots, s_{n-1}$ uit de ellips te halen. Onder welke voorwaarde op de positie van $s_0, \ldots, s_{n-1}$ heeft de handelingenreeks het gewenste resultaat?

◦ (Lastig.) Definieer een functie *uitin* die, gegeven n en de posities van $s_0, \ldots, s_{n-1}$, twee handelingenreeksen oplevert: één om $s_0, \ldots, s_{n-1}$ vanuit die posities uit de ellips te halen, en één om ze er weer in te brengen op die posities.
Definieer een functie uit'' waarvoor geldt: $uit''\ m\ n\ ps$ is de handelingenreeks om $s_m, \ldots, s_{n-1}$ uit de ellips te halen vanuit posities ps.

158. Maak de definitie van *aantal* nog efficiënter door (1) extra resultaten (verscheidene *aantal*'s te combineren), (2) extra parameters (parameteraccumulatie), en (3) nog slimmer te zijn (bekijk de uitkomsten bij 0,1, ...).

159. **Torentjes van Hanoi.** In Hanoi staat een tempel waar al sinds mensenheugenis priesters bezig zijn 64 gouden schijven te verplaatsen. De schijven hebben alle een verschillende diameter, en stonden oorspronkelijk als een toren op plaats A. Uiteindelijk moeten ze op plaats B komen. De regels waaraan de priesters zich te houden hebben zijn deze:

    nooit mag een grotere schijf op een kleinere komen;
    er mag slechts één schijf tegelijk verplaatst worden;
    de schijven mogen uitsluitend op plaatsen A, B en C gestapeld worden.

    Het gerucht gaat dat het Einde der Wereld dáár is, nog voordat de priesters hun taak volbracht hebben (alle schijven als een toren op plaats B).
    Definieer een lijst *hanoi* van verplaatsingen waarmee de 64 schijven van A naar B verplaatst kunnen worden. Representeer een verplaatsing als een tweetal (*van*, *naar*).
    Wenk:

    Veralgemeen de lijst *hanoi* tot een functie door de constante 64 tot een parameter n te maken: het aantal schijven dat verplaatst moet worden.

    Veralgemeen voorts de functie door de constanten A, B en C tot parameter x, y en z te maken: de plaatsen waar de n schijven *vandaan* en *naar toe* verplaatst moeten worden met de derde plaats als *hulp*.

    Probeer de lijst voor $n{+}1$ schijven uit te drukken, als je al beschikt over lijsten voor verplaatsingen van torentjes ter grootte n.

Definieer een efficiënte functie voor het aantal benodigde verplaatsingen. Gebruik deze om uit te rekenen hoeveel jaren de gehele verplaatsing duurt, onder aanname dat de verplaatsing van één schijf tien seconden duurt.

16 Doe Het Zelf typen

De typen die tot nu toe aan bod zijn gekomen, zijn opgebouwd uit de elementaire typen *num*, *char*, *bool*, middels de type-vormers voor tupels $(\ldots)$, lijsten $[\ldots]$ en functies $\ldots \rightarrow \ldots$. We kunnen ook zelf typen en type-vormers maken.

§16.1 Opsomming. De meest eenvoudige vorm van een nieuw type is een opsomming (enumeratie) van een aantal 'nieuwe' waarden:

$$\begin{array}{l} sekse \;::=\; M \mid V \\ week \;::=\; Ma \mid Di \mid Wo \mid Do \mid Vr \mid Za \mid Zo \end{array}$$

In de context van deze type-definitie zijn *sekse* en *week* typen, net zo als *num*, *char* en *bool*. Het type *sekse* bestaat uit precies twee waarden; de notatie van die waarden is M en V. Het type *week* bestaat uit precies zeven waarden; de notatie van die waarden is Ma, ..., Zo:

$$\begin{array}{l} M,\ V \;::\; sekse \\ Ma,\ Di,\ \ldots,\ Zo \;::\; week \end{array}$$

De namen M, V, Ma, ... , Zo heten *constructoren*; zij mogen net zo gebruikt worden als getal-, karakter- en boolean notaties. In het bijzonder mogen ze in patronen gebruikt worden:

$$\begin{array}{l} showsekse \;::\; sekse \rightarrow [char] \\ showsekse\ M \;=\; \texttt{"}mannelijk\texttt{"} \\ showsekse\ V \;=\; \texttt{"}vrouwelijk\texttt{"} \end{array}$$

$$\begin{array}{l} isWerkdag \;::\; week \rightarrow bool \\ isWerkdag\ Za \;=\; False \\ isWerkdag\ Zo \;=\; False \\ isWerkdag\ x \;=\; True \end{array}$$

Een alternatieve definitie voor *isWerkdag* luidt:

$$isWerkdag\ x \;=\; Za \neq x \neq Zo$$

De standaard grootheden *bool*, *True* en *False* zijn met opsomming gedefinieerd:

$$bool \;::=\; False \mid True$$

Constructoren moeten met een hoofdletter beginnen, en alle namen die met een hoofdletter beginnen zijn constructoren. Typen gedefinieerd met een ' ::=' heten *algebraïsch type*.

§16.2 Parameters I. Constructoren mogen parameters hebben; het zijn dan functies. Bijvoorbeeld:

$$voertuig \ ::= \ Auto\ num \mid Fiets$$

Hier is *Auto* een constructor met één parameter. Het argument van *Auto* kan bijvoorbeeld gebruikt worden voor het verbruik (aantal kilometers per liter); je zou dat argument ook een andere rol kunnen laten spelen, zoals het bouwjaar, of de cilinderinhoud, of het chassisnummer. De type-specificaties van de constructoren luidt nu:

$$\begin{array}{l} Auto \ :: \ num \to voertuig \\ Fiets \ :: \ voertuig \end{array}$$

Er bestaan oneindig veel waarden van het type *voertuig*:

$$Fiets,\ Auto\ 11.0\ ,\ Auto\ 13.5\ ,\ \ldots\ ::\ voertuig$$

De constructoren mogen in patronen gebruikt worden:

$$\begin{array}{l} wielental \ :: \ voertuig \to num \\ wielental\ (Auto\ n) \ = \ 4 \\ wielental\ Fiets \ = \ 2 \end{array}$$

$$\begin{array}{l} isZuinig \ :: \ voertuig \to bool \\ isZuinig\ (Auto\ n) \ = \ n \geq 13.0 \\ isZuinig\ x \ = \ True \end{array}$$

De haakjes zijn nodig, want bij '*isZuinig Auto n*' wordt '*isZuinig*' opgevat als een functie met '*Auto*' en '*n*' als twee *aparte* argumenten; en dat is niet wat ik bedoelde (en bovendien: in een patroon *moet* iedere constructor met álle argumenten gebruikt worden!).

§16.3 Parameters II. Type-definities mogen parameters hebben; er worden dan typevormers gedefinieerd. Beschouw als voorbeeld een *tabel*, dat wil zeggen, een verzameling paren (x, y) waarmee gemakkelijk bij iedere 'sleutel' x het bijhorende 'item' y gevonden kan worden. De typen van de sleutels en items zijn vrij te kiezen (maar liggen vast per tabel). Zo'n tabel kan op verschillende manieren gerepresenteerd worden. Afhankelijk van de grootte van de tabel ligt een lijst van paren, dan wel een echte functie, voor de hand:

$$tabel\ \alpha\ \beta \ ::= \ Klein\ [(\alpha, \beta)] \ \mid \ Groot\ (\alpha{\to}\beta)$$

De α en β staan voor een willekeurig type; bijvoorbeeld, $[char]$ en *num*:

$$tabel\ [char]\ num$$

Met dit type kunnen we een telefoonboek representeren: bij een naam kan het bijhorende nummer gevonden worden.

Voor de functie *item* die bij gegeven tabel t en sleutel x het bijhorende item y oplevert, luidt de definitie:

$$\begin{aligned} \mathit{Klein}\ t\ \underline{\mathit{item}}\ x &= \mathit{hd}\ [y \mid (x', y) \leftarrow t;\ x' = x] \\ \mathit{Groot}\ t\ \underline{\mathit{item}}\ x &= t\ x \end{aligned}$$

Wanneer *Klein* t een *tabel* $\alpha\ \beta$ is, is t zelf een lijst van type $[(\alpha, \beta)]$; en wanneer *Groot* t een *tabel* $\alpha\ \beta$ is, is t zelf een functie van type $\alpha{\rightarrow}\beta$. Functie *item* is polymorf; het meest algemene type luidt:

$$\mathit{item}\ ::\ \mathit{tabel}\ \alpha\ \beta \rightarrow \alpha \rightarrow \beta$$

Dus *item* werkt óók op het hierboven genoemde type voor telefoonboeken:

$$\mathit{item}\ ::\ \mathit{tabel}\ [\mathit{char}]\ \mathit{num} \rightarrow [\mathit{char}] \rightarrow \mathit{num}$$

§16.4 Recursie. Tenslotte, type-definities mogen ook recursief zijn. Een veelvoorkomende recursieve structuur is die van een *boom*: een vertakkende structuur, waarbij de splitsingen *knopen* heten en de eindpunten de *bladeren*. Iedere knoop en ieder blad bevat ook een waarde oftewel *label*; de labels van de bladeren zijn alle van eenzelfde soort (type), en net zo voor de labels van de knopen. Bijvoorbeeld:

```
       a                   Knoop a
      / \                        (Knoop b
     /   \                              (Blad d)
    b     c                             (Knoop e (Blad h) (Blad j)))
   / \   / \                     (Knoop c
  d   e f   g                           (Blad f)
     / \                                (Blad g))
    h   j
```

De labels zijn schematisch aangeduid met $a, b, c, d, e, f, g, h, j$; links staat een plaatje, rechts staat de notatie ervan volgens de definitie die we hieronder geven. Zonder opmaak ziet de notatie er zó uit:

$$\mathit{Knoop}\ a\ (\mathit{Knoop}\ b\ (\mathit{Blad}\ d)\ (\mathit{Knoop}\ e\ (\mathit{Blad}\ h)\ (\mathit{Blad}\ j)))\ (\mathit{Knoop}\ c\ (\mathit{Blad}\ f)\ (\mathit{Blad}\ g))$$

Als voorbeeld volgt hier een definitie van een type *boom* met getallen als labels in de knopen, en strings als labels in de bladeren:

$$\mathit{boom}\ ::=\ \mathit{Blad}\ [\mathit{char}] \mid \mathit{Knoop}\ \mathit{num}\ \mathit{boom}\ \mathit{boom}$$

Hiermee geldt:

$$\begin{aligned} \mathit{Blad}\ &::\ [\mathit{char}] \rightarrow \mathit{boom} \\ \mathit{Knoop}\ &::\ \mathit{num} \rightarrow \mathit{boom} \rightarrow \mathit{boom} \rightarrow \mathit{boom} \end{aligned}$$

De grootte van een boom (het aantal knopen zonder de bladeren) wordt gedefinieerd door:

$$\begin{array}{l} \mathit{grootte} \ :: \ \mathit{boom} \to \mathit{num} \\ \mathit{grootte}\ (\mathit{Blad}\ \mathit{xs}) \ = \ 0 \\ \mathit{grootte}\ (\mathit{Knoop}\ n\ x\ y) \ = \ 1 + \mathit{grootte}\ x + \mathit{grootte}\ y \end{array}$$

Je ziet dat de functie *grootte* is gedefinieerd met inductie naar de opbouw van het argument.

Bovenstaand type *boom* kunnen we iets veralgemenen door de constanten [*char*] en *num* tot parameter te maken. Dan luidt de definitie:

$$\mathit{boom}\ \alpha\ \beta \ ::= \ \mathit{Blad}\ \alpha \mid \mathit{Knoop}\ \beta\ (\mathit{boom}\ \alpha\ \beta)\ (\mathit{boom}\ \alpha\ \beta)$$

Hiermee geldt:

$$\begin{array}{l} \mathit{Blad} \ :: \ \alpha \to \mathit{boom}\ \alpha\ \beta \\ \mathit{Knoop} \ :: \ \beta \to (\mathit{boom}\ \alpha\ \beta) \to (\mathit{boom}\ \alpha\ \beta) \to \mathit{boom}\ \alpha\ \beta \end{array}$$

en kunnen we de grootte van een boom precies zo als hierboven definiëren:

$$\begin{array}{l} \mathit{grootte} \ :: \ \mathit{boom}\ \alpha\ \beta \to \mathit{num} \\ \mathit{grootte}\ (\mathit{Blad}\ \mathit{xs}) \ = \ 0 \\ \mathit{grootte}\ (\mathit{Knoop}\ n\ x\ y) \ = \ 1 + \mathit{grootte}\ x + \mathit{grootte}\ y \end{array}$$

Heel veel functies die met waarden van recursief gedefinieerde typen werken, zullen recursief gedefinieerd zijn.

Opgaven

160. Definieer, in de context van §16.1, de functie *dagnr* die van iedere dag van de week het dagnummer oplevert:

dag	Ma	Di	...	Zo
dagnr	0	1	...	6

161. Definieer, in de context van §16.1, de functie *dagNa* die van een dag in de week de dag erna oplevert: $\mathit{dagNa}\ \mathit{Ma} = \mathit{Di}, \ldots\ \mathit{dagNa}\ \mathit{Zo} = \mathit{Ma}$.

162. Gegeven is het type *temp* waarmee een temperatuur op twee verschillende manieren gerepresenteerd kan worden, Celcius en Fahrenheit:

$$\mathit{temp} \ ::= \ C\ \mathit{num} \mid F\ \mathit{num}$$

Geef definities voor de volgende functies:

$$\begin{array}{lcl} t\ \underline{\mathit{verhoogdMet}}\ n & = & \text{de representatie van “}t + (n \text{ graden)”} \\ t\ \underline{\mathit{isEvenwarm}}\ t' & = & t \text{ en } t' \text{ stellen gelijke temperaturen voor} \\ t\ \underline{\mathit{isWarmer}}\ t' & = & t \text{ stelt een hogere temperatuur voor dan } t' \end{array}$$

Bedenk dat $0^o\ C$, $100^o\ C$ en $100^o\ F$ de temperaturen zijn van smeltend ijs, kokend water, en de lichaamstemperatuur van een mens ($37^o\ C$). Verder is $0^o\ F = -17.7^o\ C$ (de indertijd

laagst bereikbare temperatuur). Dus $32^o\ F = 0^o\ C$, en één graad fahrenheit moet met 5/9 vermenigvuldigd worden om één graad Celsius of Kelvin te krijgen.
Wenk: gebruik conversiefuncties (Celsius en Fahrenheit naar Kelvin) om lelijk gevalsonderscheid zoveel mogelijk te vermijden.

163. Geef een (de!) definitie van het type *bool*.
164. Definieer een type *nat* voor de 'natuurlijke getallen'; die getallen zijn opgebouwd uit 'nul' en de 'opvolger' (die bij ieder natuurlijk getal z'n opvolger oplevert).
Definieer de optelling en vermenigvuldiging voor dat type; $plus,\ mult\ ::\ nat \to nat \to nat$.
Wenk: gebruik inductie naar de opbouw van het rechter argument.
165. Definieer een type *lijst* waarmee je Miranda-lijsten kunt representeren; maak geen gebruik van de Miranda-lijsthaken [en].
Definieer de *map*-, ++- en *concat*-functie voor dit type. Wenk: gebruik bij de ++-implementatie inductie naar het linker argument.
166. Definieer een type *tupel*3 waarmee je Miranda's 3-tupels kunt representeren; maak geen gebruik van Miranda's 3-tupels (, ,).
Definieer de functies *fst*3 en *snd*3 voor dit type.
167. Definieer, in de context van §16.3, een functie *met* zó dat:

$$t\ \underline{met}\ (x, y) \quad = \quad \text{de tabel die identiek is aan } t \text{ behalve dat nu bij sleutel } x \text{ het item } y \text{ hoort}$$

Geef ook het type van *met*.
168. Gegevens voor één persoon, zeg Wietske, worden vastgelegd in lijsten zoals:

$$[Naam\ \texttt{"}wietske\texttt{"},\ Geboren\ 1977,\ Sekse\ V]$$
$$[Naam\ \texttt{"}wietske\texttt{"}]$$
$$[Geboren\ 1977,\ Naam\ \texttt{"}wietske\texttt{"}]$$
$$[Sekse\ V,\ Naam\ \texttt{"}wietske\texttt{"},\ Geboren\ 1977]$$

(a) Definieer een type *gegeven* zó dat bovenstaande lijsten van type [*gegeven*] zijn.
Zij nu:

$$persoon\ \equiv\ [gegeven]$$

(b) Definieer de functie *vrouwental* met:

$$\begin{aligned} vrouwental\ &::\ [persoon] \to num \\ vrouwental\ ps\ &=\ \text{het aantal personen } p \text{ in } ps \text{ dat } Sekse\ V \text{ bevat} \end{aligned}$$

(c) Definieer de functie $geslacht\ ::\ [persoon]{\to}[char]{\to}[char]$ met de volgende eigenschappen. Laat $nm :: [char]$ en $ps :: [persoon]$, en stel dat ps geen persoon $[\ldots Naam\ nm \ldots]$ bevat. Dan:

$$geslacht\ ps\ nm\ =\ \text{“afwezig”}$$

Als ps precies één persoon $[\ldots\ Naam\ nm\ \ldots]$ bevat, zeg p, dan:

$$geslacht\ ps\ nm\ =\ \text{“man”, als } p \text{ bevat } Sekse\ M$$

$$= \text{“vrouw”, als } p \text{ bevat } \mathit{Sekse}\ V$$
$$= \text{“onbekend”, als } p \text{ bevat geen } \mathit{Sekse}\ \ldots$$

Ga er van uit dat ieder soort gegeven hoogstens eenmaal voorkomt in een lijst van persoonsgegevens.

169. Functies zoals *hd*, *max*, *div* resulteren bij sommige argumenten in een foutstop; we zeggen dat *hd* [] en x *div* 0 enzovoort 'niet bestaan'. We definiëren nu varianten van die functies die altijd goed eindigen:

$$\begin{array}{l} \mathit{maybe}\ \alpha \ ::=\ \mathit{No} \mid \mathit{Yes}\ \alpha \\ \mathit{hd}'\ [\,] \ =\ \mathit{No} \\ \mathit{hd}'\ (x : xs) \ =\ \mathit{Yes}\ x \end{array}$$

Nu wordt een niet bestaand resultaat gemodelleerd met *No*, en een bestaande waarde w met *Yes* w.
Geef het type van hd'. Definieer max', div' analoog, en specificeer hun type.
Definieer een compositie *cmp* die de varianten f', g' van f, g als volgt samenstelt:

$$\begin{array}{lll} (f'\ \underline{\mathit{cmp}}\ g')\ x \ =\ \mathit{Yes}\ z & \text{wanneer } g\ x \text{ en } f\ (g\ x) \text{ bestaan, en } z\ =\ (f\ (g\ x) \\ (f'\ \underline{\mathit{cmp}}\ g')\ x \ =\ \mathit{No} & \text{wanneer } g\ x \text{ of } f\ (g\ x) \text{ niet bestaat.} \end{array}$$

Gebruik *cmp* om de variant f' van de volgende functie te definiëren:

$$f\ xs \ =\ 3\ \mathit{div}\ (\mathit{max}\ xs)$$

Om te onthouden

- Inductief opgebouwde waarden worden in Miranda met een algebraïsch type gerepresenteerd. Een bijzonder geval hiervan zijn opsommingen (zoals: de dagen van de week, of waarheidswaarden *True* en *False*.)
- Algebraïsch gedefinieerde typen mogen parameters hebben: α, β, ...; het zijn dan "typeconstructoren", en de constructoren zijn polymorf.
- In principe zijn de standaard typen en type-constructoren zelf te definiëren: met algebraïsche typen.

17 Bomen

Boomstructuren komen veel voor bij het representeren van gegevens. Aangezien boomstructuren recursief gedefinieerd worden, zullen veel functies die daarmee werken recursief gedefinieerd zijn. Alle technieken van recursieve definities (zie Hoofdstuk 13) gelden ook hier. Met name de techniek van veralgemening tot een functie met extra parameters en/of extra resultaten.

§17.1 Boombegrippen. Boomstructuren komen misschien nog wel vaker voor dan lijststructuren. Toch is er geen standaard notatie voor bomen, omdat er zo veel verschillende varianten zijn. Bijvoorbeeld: wel of geen labels in de bladeren, wel of geen labels in de knopen, twee of meer deelbomen per knoop, een vast of variabel aantal deelbomen per knoop, verscheidene *soorten* knopen, enzovoorts. Ook zijn er vele synoniemen in omloop voor de terminologie:

$$\mathit{Blad} \approx \mathit{Tip} \approx \mathit{Nil} \approx \mathit{Leaf}$$

$$\mathit{Knoop} \approx \mathit{Join} \approx \mathit{Cons} \approx \mathit{Node} \approx \mathit{Fork}$$

Hier zijn nog wat belangrijke begrippen:

De *wortel* van een boom is (het label van) de “bovenste” knoop.
Een *pad* van een boom is de rij (labels van) deelbomen vanaf de wortel tot en met (of tot aan) een blad.
De *diepte* van een boom is de lengte van een langste pad.
De *grootte* van een boom is het aantal knopen (of: knopen én bladeren).
Een boom is *vol* als alle paden even lang zijn.
Van een (deel)boom $x = \mathit{Knoop}\ w\ y\ z$ heten de deelbomen y en z de *zonen* van x, en heet x de *vader* van y en z.
De *deelbomen* van een boom zijn de boom zelf (!) en alle deelbomen van alle zonen van de boom.
Een boom is *evenwichtig* als in iedere knoop de dieptes van de zonen hoogstens één verschillen.

§17.2 Pre-, in-, post-order. Er zijn verscheidene manieren om de labels van een boom op te sommen. Beschouw weer als voorbeeld de volgende boom:

```
       a
      / \
     /   \
    b     c
   / \   / \
  d   e f   g
     / \
    h   j
```

```
Knoop a
      (Knoop b
             (Blad d)
             (Knoop e (Blad h) (Blad j)))
      (Knoop c
             (Blad f)
             (Blad g))
```

Eén mogelijke opsomming is laag-voor-laag:

$a\ b\ c\ d\ e\ f\ g\ h\ j$: breadth-first opsomming

In deze opsomming is het *niet* zo dat de labels van een deelboom bij elkaar blijven; dat is wel het geval bij de volgende *depth-first* opsommingen. We onderstrepen de deelbomen:

$$\begin{array}{lcl}
a\ \ b\ d\ e\ h\ j\ \ c\ f\ g & : & \text{pre-order opsomming} \\
d\ b\ h\ e\ j\ \ a\ \ f\ c\ g & : & \text{in-order opsomming} \\
d\ h\ j\ e\ b\ \ f\ g\ c\ \ a & : & \text{post-order opsomming}
\end{array}$$

De variaties betreffen de plaats van de label van een knoop ten opzichte van de opsommingen van de bijhorende deelbomen; zie met name de plaats van a. Dit is wellicht nog duidelijker in de definities van de functies die de labels van een boom opsommen. Op knopen werken zij als volgt:

$$\begin{array}{lcll}
\mathit{labels}\ (\mathit{Knoop}\ a\ x\ y) & = & [a] \mathbin{+\!\!+} \mathit{labels}\ x \mathbin{+\!\!+} \mathit{labels}\ y & \|\ \mathit{pre{-}order} \\
\mathit{labels}\ (\mathit{Knoop}\ a\ x\ y) & = & \mathit{labels}\ x \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \mathit{labels}\ y & \|\ \mathit{in{-}order} \\
\mathit{labels}\ (\mathit{Knoop}\ a\ x\ y) & = & \mathit{labels}\ x \mathbin{+\!\!+} \mathit{labels}\ y \mathbin{+\!\!+} [a] & \|\ \mathit{post{-}order}
\end{array}$$

Merk op dat de volgorde van x en y links en rechts hetzelfde is. Bij de pre- / in- / post-order opsomming staat de label van de knoop vóór / tussen / ná de opsommingen van de deelbomen x en y. De opsomming kan ook zó beschreven worden:

> Doorloop het plaatje van de boom *vlak langs* de verbindingslijnen, beginnend bij de top, aan de linkerkant. Daarbij passeer je ieder label driemaal: eerst aan de linkerkant, dan aan de onderkant, en tenslotte aan de rechterkant. Bij de pre-order opsomming wordt elk label opgesomd bij de eerste passage; bij de in-order opsomming wordt elk label opgesomd bij de tweede passage; en bij de post-order opsomming gebeurt het pas bij de derde passage.

Opgaven

170. *Dit is een inleiding voor alle opgaven in deze serie.*
Beschouw structuren t en t' van de volgende vorm:

```
t  =     7                t’ =        7
       /  \                      --------------
      6    9                     6      8  2  9
    /  \                    ---------  -- --  --
   1    4                   1   5   4
                           --  --  --
```

Deze kunnen weergegeven worden door waarden van de volgende typen:

$$\begin{array}{lcl}
\mathit{tree} & ::= & \mathit{Tip\ num} \mid \mathit{Node\ num\ tree\ tree} \\
\mathit{tree}' & ::= & \mathit{Node}'\ \mathit{num}\ [\mathit{tree}']
\end{array}$$

Immers, we kunnen t en t' zien als een plaatje van:

$$\begin{array}{lcl}
t & = & \mathit{Node}\ \ 7\ \ (\mathit{Node}\ 6\ (\mathit{Tip}\ 1)\ (\mathit{Tip}\ 4))\ \ (\mathit{Tip}\ 9) \\
t' & &
\end{array}$$

$$
\begin{aligned}
= \mathit{Node'}\ 7\ [& \\
& \mathit{Node'}\ 6\ [\mathit{Node'}\ 1\ [],\ \mathit{Node'}\ 5\ [],\ \mathit{Node'}\ 4\ []], \\
& \mathit{Node'}\ 8\ [], \\
& \mathit{Node'}\ 2\ [], \\
& \mathit{Node'}\ 9\ []]
\end{aligned}
$$

De opgave is om (1) een definitie, (2) de type-specificatie, én (3) een goede naam (kort en krachtig) te geven voor elke functie die hieronder beschreven wordt.
In de antwoorden worden $t, t', \ldots$ of $s, s', t, t', \ldots$ gebruikt als variabelen van het type *tree*, en $ss, ts, \ldots$ als variabelen van het type [*tree'*]. Voor het leesbaar tonen van bomen, bij het uittesten van een definitie, kun je functies g en g' uit opgave 181 of functie *showtree* uit opgave 182 gebruiken.

171. (Zie opg 170.) Functies *depth* en *depth'* leveren de diepte van de boom op. De diepte van *Tip* n en van *Node'* n [] stellen we op 0. Dus $\mathit{depth}\ t = \mathit{depth'}\ t' = 2$. Gebruik deze functies zo nodig in volgende opgaven.

172. (Zie opg 170.) Functie f levert de grootte (hier: het aantal *interne* knopen) van de boom op. Voorbeeld:

$$
\begin{aligned}
f\ t &= 2 \\
f'\ t' &= 8
\end{aligned}
$$

Het aantal *interne* Nodes in t is 2; het aantal *interne* Nodes in t' is 8. Analoog voor de accent-versie.

173. (Zie opg 170.) Functie f levert de som van alle Tip-waarden. Voorbeeld:

$$
\begin{aligned}
f\ \ t &= \mathit{som}\ \{1,4,9\} = 14 \\
f'\ t' &= \mathit{som}\ \{1,5,4,8,2,9\} = 29
\end{aligned}
$$

174. (Zie opg 170.) Functie f levert de som van alle Tip- en Node-waarden. Voorbeeld:

$$
\begin{aligned}
f\ t &= \mathit{som}\ \{1,4,6,7,9\} = 27 \\
f'\ t' &= \mathit{som}\ \{1,5,4,6,8,2,9,7\} = 42
\end{aligned}
$$

175. (Zie opg 170.) Functie f geeft aan of een gegeven getal in de boom voorkomt. Voorbeeld:

$$
\begin{aligned}
f\ 3\ t &= \mathit{False} \qquad & f'\ 3\ t' &= \mathit{False} \\
f\ 6\ t &= \mathit{True} \qquad & f'\ 6\ t' &= \mathit{True}
\end{aligned}
$$

176. (Zie opg 170.) Functie f “vermenigvuldigt” alle Tip-waarden met 10. Voorbeeld:

```
f t =    7            f’ t’ =          7
        / \                    --------------
       6   90                  6      80 20 90
      / \              ----------  -- -- --
    10   40            10  50  40
                       --  --  --
```

177. (Zie opg 170.) Functie f levert de gespiegelde boom. Voorbeeld:

```
f t  =    7           f' t' =          7
        /  \                   --------------
       9    6                  9  2  8      6
           / \                -- -- --  ---------
          4    1                         4   5   1
                                        --  --  --
```

178. (Zie opg 170.) Functie f levert een lijst van alle paden. Voorbeeld:

$f\ t$ bevat precies $[7,6,1]$, $[7,6,4]$ en $[7,9]$;
$f'\ t'$ bevat precies $[7,6,1]$, $[7,6,5]$, $[7,6,4]$, $[7,8]$, $[7,2]$ en $[7,9]$.

179. (Zie opg 170.) Functie f' geeft van een $tree'$ aan of, langs ieder pad naar beneden toe, het aantal subbomen per knoop steeds kleiner wordt. Voorbeeld: $f'\ t1 = True$, en $f'\ t2 = False$, wanneer:

```
t1 =         7                              t2 =            7
       --------------                               ------------------
       6      8  2   9                                 6                9
  --------  -- --   --                             ---------           --
   1  5   4                                        1   5   4
  -- --  --                                       --  --  --
```

180. (Zie opg 170.) Functies f, g, h, j sommen de getallen uit de boom op in pre-order, resp. in-order, post-order volgorde, en per diepte (= breadth-first). Voorbeeld:

$$
\begin{array}{rclrcl}
f\ t &=& [7,6,1,4,9] & f'\ t' &=& [7,6,1,5,4,8,2,9] \\
g\ t &=& [1,6,4,7,9] & & & \\
h\ t &=& [1,4,6,9,7] & h'\ t' &=& [1,5,4,6,8,2,9,7] \\
j\ t &=& [[7],[6,9],[1,4]] & j'\ t' &=& [[7],[6,8,2,9],[1,5,4]]
\end{array}
$$

Voor de accent-versie is de in-order opsomming niet zinvol.

181. (Zie opg 170.) Functies f, g, h geven een lineaire representatie van de boom, in pre- / in- / post-order, met haakjes die de boomstructuur aangeven: een paar haakjes voor elke *Node*, geen haakjes voor de *Tip*s. Voorbeeld:

$f\ t$ = "(7 (6 1 4) 9)" $f'\ t'$ = "(7 (6 1 5 4) 8 2 9)"
$g\ f$ = "((1 6 4) 7 9)"
$h\ t$ = "((1 4 6) 9 7)" $h'\ t'$ = "((1 5 4 6) 8 2 9 7)"

Voor de accent-versie is de in-order representatie niet zinvol.

182. (Zie opg 170.) Functie *showtree* toont een boom op een leesbare manier. De afdruk van *showtree* t, respectievelijk *showtree'* t', is:

```
7                                | 7
      6                          |       6
            1                    |             1
            4                    |             5
      9                          |             4
                                 |       8
                                 |       2
```

```
|     9
```

Wenk: veralgemeen de functie *showtree* tot *showtr* die een extra parameter heeft voor de hoeveelheid indentatie waarmee de boom moet worden getoond.

183. (Zie opg 170.) Functie *showtreeC* toont een boom op een leesbare manier, net als in de vorige opgave, maar nu iets Compacter. De afdruk van *showtreeC* t, respectievelijk *showtreeC'* t', is:

```
7   6   1                              | 7   6   1
        4                              |         5
    9                                  |         4
                                       |     8
                                       |     2
                                       |     9
```

Dus de eerste subboom van een Node staat op dezelfde regel als de label van de Node.
Wenk. Pas de definitie van *showtree* uit de vorige opgave aan. In vergelijking met *showtree* moet de "$\backslash n$" aan het begin de éérste subtree vervallen, evenals een x-aantal spaties (waarbij $x =$ het aantal karakters dat al op die regel wordt afgedrukt).

184. (Zie opg 170.) Functie f levert uit een lijst xs een evenwichtige boom op waarvan de pre-order labelopsomming der *Node*s precies xs is; alle *Tip*-waarden moeten 0 zijn. Merk op dat verschillende bomen eenzelfde pre-order opsomming van de labels kunnen hebben. Door de eis van evenwichtigheid is er weinig keus: de linker en rechter zoon moeten 'ongeveer' evenveel knopen hebben.
Net zo voor g, h met in- en post-order in plaats van pre-order.

185. (Zie opg 170.) Functie f geeft aan of iedere Node-waarde de som is van de waarden er direct onder. Voorbeeld: f $t1 = False$, en f $t2 = True$, wanneer:

```
t1   =   7              t2   =   7
        / \                     / \
       6   9                   6   1
      / \                     / \
     1   4                   2   4
```

Analoog voor de accent-versies.

186. (Zie opg 170.) Functie f geeft aan of de boom evenwichtig is.

187. (Zie opg 170.) Functie f geeft aan of in iedere knoop de diepte van de linkerzoon hoogstens de diepte van de rechterzoon is. Analoog voor f'.

188. (Zie opg 170.) Functie f geeft aan of de boom vol is.

189. (Zie opg 170.) Functie f geeft aan of in iedere knoop de zonen allemaal een *Node* zijn of allemaal een *Tip*. (Voor de accent-versie: lees *Node* n [] voor *Tip*.)

190. (Zie opg 170.) Functie f werkt op twee bomen en geeft aan of beide bomen gelijke structuur hebben, dat wil zeggen, alleen maar verschillen in de label waarden. Analoog voor de accent-versie.

191. (Zie opg 170.) Functie f geeft aan of de boom een "zoekboom" is, dat wil zeggen, in iedere (*Node* n t t') geldt dat $x < n < y$ voor alle getallen x in t en y in t'.
Voorbeeld: f $t1 = False$, en f $t2 = True$, wanneer:

```
t1   =   7                t2   =   7
        / \                       / \
       6   9                     4   9
      / \                       / \
     1   4                     1   6
```

Geen accent-versie van deze opgave.
Wenk. Veralgemeen f tot een functie g met extra parameters, of tot een functie h met een extra resultaat; beide mogelijkheden lukken.

192. (Zie opg 170 en 191.) Functie f geeft, net als in opgave 175, aan of een gegeven getal in de boom voorkomt. Geef een efficiëntere definitie voor f, gegeven dat de boom een zoekboom is. Geen accent-versie.
193. (Zie opg 170.) Functie f zet in iedere Node en Tip: de som van de getallen in het pad vanaf die Node of Tip naar boven. Voorbeeld:

```
f t =    7
        / \
      6+7  9+7
      / \
  1+6+7 4+6+7
```

Analoog voor de accent-versie.

194. (Zie opg 170.) Functie f zet in iedere Node en Tip: de som van alle getallen in de deelboom vanaf die Node of Tip naar beneden. Voorbeeld:

```
f t    =   7+6+1+4+9
             / \
      6+1+4 9
        / \
       1   4
```

Analoog voor de accent-versie.

195. (Zie opg 170.) Functie f verhoogt van links naar rechts de Tip-waarden met 0,1,2,3,... Voorbeeld:

```
f t =   7
       /  \
      6    9+2
    /  \
   1+0  4+1
```

Analoog voor de accent-versie.

196. (Zie opg 170.) Functie f verhoogt iedere Node en Tip waarde in pre-order (of in-order, post-order, etc) volgorde met 0,1,2,3,... Voorbeeld:

```
f t =   7+0
       /  \
    6+1    9+4
    /  \
  1+2  4+3
```

Analoog voor de accent-versies in het pre-order en post-order geval. Waarschuwing: efficiënte definities voor accent-versies zijn lastig (misschien wel te moeilijk in dit stadium).

197. (Zie opg 170.) Functie f reconstrueert een boom uit de pre-order opsomming er van (met $+n$ in de opsomming voor Node-waarden en $-n$ voor Tip-waarden). Voorbeeld:

$$\begin{aligned} f\ [7,6,-1,-4,-9] &= t \\ f\ [7,-9,6,-4,-1] &= \text{de gespiegelde van } t, \quad (= f\ t \text{ van opgave 177}) \end{aligned}$$

Geen accent-versie.

198. (Zie opg 170.) Representeer een pad als een rij nullen en enen: een nul voor 'linksaf' en een één voor 'rechtsaf' in een wandeling vanuit de top naar beneden. Functie f decodeert zo'n representatie, en geeft de labels van de *Nodes* langs het pad. Functie g geeft de label van de *Tip* waarin het pad eindigt. Beide functies geven een foutmelding wanneer de rij nullen en enen geen pad codeert. Voorbeeld:

$$\begin{aligned} & f\ t\ [0,0]\ =\ f\ t\ [0,1]\ =\ [7,6] \\ & g\ t\ [0,0]\ =\ 1 \\ & g\ t\ [0,1]\ =\ 4 \end{aligned}$$

199. (Zie opg 170 en 198.) Functies *fInv* en *gInv* zijn een soort inverse van f en g uit opgave 198: bij een labellijst xs, afkomstig van een pad in boom t, levert $fInv\ t\ xs$ een lijst met alle nul-een-rijen ys waarvoor $f\ t\ ys = xs$; en analoog voor $gInv$. Als in een boom geen label twee- of meermaal voorkomt, levert *fInv* een lijst op die hoogstens twee pad-representaties bevat, bij *gInv* hoogstens één. Voorbeeld:

$$\begin{aligned} & fInv\ t\ [7,6]\ =\ [[0,0],\ [0,1]] \\ & gInv\ t\ 4\ =\ [[0,1]] \end{aligned}$$

Om te onthouden

• Functie op recursieve structuren, zoals bomen, zijn vaak recursief gedefinieerd. De algemene technieken voor recursie spelen ook hier (bij recursieve structuren zoals bomen) een grote rol.

• Met name is het soms nodig om zo'n functie te veralgemenen met een extra parameter. In zo'n extra parameter kan extra informatie "van buiten (de aanroep) naar binnen" worden gebracht.

• Ook is het soms nodig om zo'n functie te veralgemenen met een extra resultaat. In zo'n extra resultaat kan extra informatie "van binnen naar buiten" worden gebracht.

18 Bomen — toepassingen

In "alledaagse" problemen komen bomen in velerlei vermommingen voor. Daarvan geven we hier een paar voorbeelden.

Opgaven

Dit is de inleiding voor alle opgaven in deze serie.
Een mobiel m ziet er bijvoorbeeld als volgt uit:

```
m  =                 |
               5 m   |   3 m
            ---------|------------
            |                    |
       8 m  |   2 m              |
    --------|---------         9 kg
    |                |
    |                |
   1 kg             4 kg
```

De horizontale "armen" en verticale "touwtjes" hebben geen gewicht. Een mobiel hangt in evenwicht precies wanneer voor *iedere arm* geldt:

lengte van de linker arm × totale gewicht onder linker arm =
lengte van de rechter arm × totale gewicht onder rechter arm.

Mobiel m hangt dus niet in evenwicht; de linker sub-mobiel wel.

200. Definieer een type *mobiel* waarmee mobielen weergegeven kunnen worden, en schrijf m als een waarde van dat type:

$$\begin{array}{lll} \mathit{mobiel} & ::= & \ldots \mid \ldots\ldots \\ m & = & \ldots \end{array}$$

201. Veronderstel dat je een functie $\mathit{gew} :: \mathit{mobiel} \rightarrow \mathit{num}$ hebt, die van een mobiel het totale gewicht oplevert. (Die functie komt overeen met de f van opgave 173.) Definieer met behulp van *gew* een functie *isBal* die de gebalanceerdheid van een mobiel geeft:

$$\mathit{isBal} :: \mathit{mobiel} \rightarrow \mathit{bool}$$

Geef ook een definitie van *gew*.

202. Je ziet dat functie *isBal* per arm niet alleen zichzelf recursief aanroept, maar ook de functie *gew* aanroept (en die roept per arm op zijn beurt ook weer *gew* recursief aan). Dat kan efficiënter. Definieer in Miranda een functie *bal′* die de twee berekeningen van *isBal* en *gew* combineert en in één keer doet:

$$\mathit{bal}' :: \mathit{mobiel} \rightarrow (\mathit{bool}, \mathit{num})$$

De linkercomponent in $bal'\ m$ is in feite de waarde $isBal\ m$, de rechtercomponent is $gew\ m$. Met andere woorden:

$$bal'\ m \ = \ (isBal\ m,\ gew\ m)$$

Als je bal' hebt gedefinieerd, is $isBal$ dus makkelijk: $isBal \ = \ fst\ .\ bal'$.

Opgaven

Deze serie opgaven gaat over rekenkundige expressies, zoals `3*((4*5)+6))`.

Een rekenkundige *expressie* heeft één van de volgende drie vormen: het is een getal, of het is een optelling (operator `+` met twee expressies als argument), of het is een vermenigvuldiging (operator `*` met twee expressies als argument).
Zorg ervoor dat je bij alle opgaven uit gaat van het type *expressie* dat in opgave 203 gevraagd (en in de antwoorden gegeven) wordt.

203. Definieer het type *expressie* ter representatie van rekenkundige expressies. Definieer ook een functie *showExpr* die een expressie volledig behaakt toont.
Bijvoorbeeld, laat `3*((4*5)+6))` gerepresenteerd zijn door $e0$ van type *expressie*. Dan geldt: $showExpr\ e0$ = `"(3 * ((4 * 5) + 6))"`. Dus iedere operator wordt omsloten door een haakjespaar.

204. Definieer de functie *showExprZ* die *Z*uinig is met het tonen van haakjes. Laat expressie `3*((4*5)+6)` gerepresenteerd zijn door $e0$. Dan geldt: $showExpr\ e0$ = `"3 * (4 * 5 + 6)"`. Er moeten haakjes komen om een optelling wanneer die een argument is van een vermenigvuldiging; en nergens anders.
Wenk. Gebruik voor de overzichtelijkheid deze hulpfuncties:

$$\begin{aligned} behaakt\ xs &= \texttt{"("} +\!\!+\ xs\ +\!\!+\ \texttt{")"} \\ onhaakt\ xs &= \texttt{" "} +\!\!+\ xs\ +\!\!+\ \texttt{" "} \end{aligned}$$

205. Definieer de functie *showExprP* die de Poolse notatie van expressie toont. Bijvoorbeeld, laat `3*((4*5)+6)` gerepresenteerd zijn door $e0$. Dan geldt: $showExprP\ e0$ = `"* 3 + * 4 5 6"`. De Poolse notatie heeft steeds het operatie-symbool vóóraan, gevolgd door de Poolse notatie van de twee argumenten. Er zijn dan geen haakjes nodig.

206. Definieer de functie *exprVal* die de waarde van een expressie oplevert. Bijvoorbeeld, laat `3*((4*5)+6)` gerepresenteerd zijn door $e0$. Dan geldt: $exprVal\ e0 = 78$.

207. Definieer de functie $stdVorm\ ::\ expressie \rightarrow expressie$ die een 'standaard vorm' van een expressie oplevert. Bijvoorbeeld, laat `3*((4*5)+6)` gerepresenteerd zijn door $e0$. Dan geldt: $showExprZ\ (stdVorm\ e0)$ = `"3 * 4 * 5 + 3 * 6"`. We zeggen dat een expressie 'in standaard vorm' staat als elke vermenigvuldiging géén optellingen meer bevat. Door herhaaldelijk de volgende wetten toe te passen (van links naar rechts) kan zo'n standaard vorm bereikt worden:

$$\begin{aligned} x \times (y + z) &= x \times y \ + \ x \times z \\ (x + y) \times z &= x \times z \ + \ y \times z \end{aligned}$$

Wenk. Gebruik de infix notatie voor *Plus* en *Maal* en eigen hulpfuncties zodat de overeenkomst met bovenstaande wetten duidelijk is:

$$\begin{array}{lcl} x\ \underline{Maal}\ (y\ \underline{Plus}\ z) & = & (x\ \underline{Maal}\ y)\ \underline{Plus}\ (x\ \underline{Maal}\ z) \\ (x\ \underline{Plus}\ y)\ \underline{Maal}\ z & = & (x\ \underline{Maal}\ z)\ \underline{Plus}\ (y\ \underline{Maal}\ z) \end{array}$$

208. Hetzelfde als in de vorige opgaven, 203 t/m 207, maar nu met een extra expressie-vorm: een expressie kan ook een variabele zijn. Definieer wederom een type *expressie* ter representatie van rekenkundige expressies, en definieer de functies *showExpr*, *showExprZ*, *showExprP*, *exprVal* en *stdVorm*. Functie *exprVal* heeft een extra parameter nodig die voor iedere variabele zijn waarde geeft (noem die parameter de 'omgeving' *omg*).
Voorbeeld: Laat de rekenkundige expressie `3*((a*5)+b))` gerepresenteerd zijn door $e1$ van het nieuwe type *expressie*. Laat *omg* een functie zijn die aan (de representatie van) variabele `a` de waarde 4 toekent, en aan (de representatie van) de variabele `b` de waarde 6. Dan:

showExpr $e1$	=	`"(3 * ((a * 5) + b))"`
showExprZ $e1$	=	`" 3 * ( a * 5  + b) "`
showExprP $e1$	=	`"* 3 + * a 5 b"`
exprVal omg $e1$	=	78
showExprZ(*stdVorme*1)	=	`"3 * a * 5 + 3 * b"`

Wenk 1: representeer een variabele binnen *expressie* door een string; de functie *omg* uit het voorbeeld heeft dan de eigenschap dat *omg* "*a*" = 4, en *omg* "*b*" = 5.
Wenk 2: geef alleen de aanpassingen in de definities van opgaven 203 t/m 207.

209. Een rekenkundige *expressie* heeft één van de volgende twee vormen: het is een getal, of het is een bewerking (met twee *expressie*s en een *operator* als onderdeel). Een *operator* heeft één van de volgende vormen: het is een `*`, `/`, `+` of een `-`. Doe nu hetzelfde als in opgaven 203 t/m 207.
Net als in Miranda stellen we dat `*` en `/` voorrang hebben op `+` en `-`; dus *showExprZ* toont de expressie `a+(b*c)` als `a+b*c`. Net als in Miranda stellen we dat `*`, `/`, `+` en `-` links associatief zijn; dat wil zeggen, voor zo'n operator $\oplus$ geldt dat $a \oplus b \ldots \oplus c$ staat voor $((a \oplus b) \ldots \oplus) c$, met de haakjes naar links toe genest. Dus *showExprZ* toont `(a/b)/(c/d)` als `a/b/(c/d)`.
De standaard vorm van een expressie is, ruwweg gezegd, een gelijkwaardige expressie die bij het tonen zo weinig mogelijk haakjes heeft. (Deze eis legt de standaard vorm nog niet geheel vast; probeer een verstandige keuze te maken.)

Opgaven

We gaan eenvoudige propositie-logica modelleren. De proposities die wij beschouwen zijn opgebouwd uit 'propositionele constanten' $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \ldots$ middels de twee 'connectieven' ¬... en ...⇒... . Bijvoorbeeld, $\mathbf{a}_{17}$ en $(\mathbf{a}_7 \Rightarrow (\neg \mathbf{a}_3))$ en $((\mathbf{a}_0 \Rightarrow (\neg \mathbf{a}_1)) \Rightarrow ((\neg \mathbf{a}_1) \Rightarrow \mathbf{a}_0))$ zijn proposities.

210. Definieer een type *propositie* ter representatie van proposities, en representeer daarmee de gegeven voorbeeld-proposities.
Wenk. Gebruik (waar mogelijk en zinvol) de infix notatie.

211. Een valuatie (ofwel: waardering) is een functie die aan proposities een waarde (0 of 1) toekent, en voldoet aan de volgende eigenschap:

$val(p \Rightarrow q) = 0$ precies wanneer: $val\ p\ =\ 1$ en tevens $val\ q\ =\ 0$
$val(\neg p) = 0$ precies wanneer: $val\ p\ =\ 1$

Laat v een functie zijn die aan (de representatie van) de constanten $\mathbf{a}_i$ een waarde (0 of 1) toekent. Definieer de valuatie *val* die aan willekeurige propositie een waarde toekent en daarbij aan (de representatie van) $\mathbf{a}_i$ (alle i) dezelfde waarde toekent als de gegeven functie v . Geef ook het type van *val* en v.

212. Veralgemeen de functie *val* van de vorige door de gegeven functie v tot parameter van *val* te maken: geef de herziene definitie en het type.
213. Definieer een functie *consts* met de volgende eigenschap:

   consts p = een lijst, zonder duplicaten, van alle propositionele constanten die in propositie p voorkomen.

   Geef ook het type van de functie *consts*.
214. Een propositie p heet tautologie als alle mogelijke valuaties allemaal de waarde 1 aan p toekennen. Definieer de functie *isTaut* :: *propositie* $\rightarrow$ *bool* met de eigenschap:

   isTaut p = 'propositie p is een tautologie'.

   Maak desgewenst gebruik van functie *vs* (zonder hem te definiëren):

   vs cs = een lijst van alle functies v die aan de propositionele constanten in de lijst *cs* een waarde (0 of 1) toekennen.
215. Geef het type van functie *vs* uit de vorige opgave, en geef er een definitie van in Miranda.

19 Afscherming van de representatie en implementatie

> Voor grootschalige programmatuur is het nodig om *keuzen* die gemaakt worden in de representatie van gegevens en in de implementatie van functies *af te schermen.* Het doel daarvan is dat een verandering in die representatie en implementatie géén verandering noodzakelijk maakt in het gebruik (van de gegevens en functies).

§19.1 Abstype. Neem als voorbeeld een *bag* (engels voor: zak), dat is: een lijst waarin de volgorde niet terzake doet, ofwel een verzameling waarin de elementen meerdere voorkomens kunnen hebben. Een bag kunnen we representeren als een lijst, en de bijhorende functies luiden als volgt:

$$
\begin{aligned}
&\mathit{bag}\ \alpha\ \equiv\ [\alpha]\\
&\mathit{nbr}\ \mathit{xs}\ x\ =\ \#\mathit{filter}\ (=x)\ \mathit{xs}\\
&\mathit{add}\ \mathit{xs}\ x\ =\ x : \mathit{xs}\\
&\mathit{del}\ \mathit{xs}\ x\ =\ \mathit{xs}\ \texttt{--}\ [x]\\
&\mathit{empty}\ =\ []
\end{aligned}
$$

Het zou nu prettig zijn wanneer de correctheid van toekomstige definities niet afhangt van de keuze die we hierboven gedaan hebben voor de representatie voor bags. Want dan kunnen we nog een andere keuze maken voor de representatie en bovenstaande functies daaraan aanpassen. Te denken valt aan: een gesorteerde lijst of zoekboom (zodat *nbr* veel efficiënter geïmplementeerd kan worden), een lijst van (element, aantal voorkomens)-paren (zodat de opslagruimte efficiënter benut wordt), enzovoorts.

Welnu, de volgende verklaring 'schermt de representatie-keuze af':

$$
\begin{aligned}
&\mathit{abstype}\ \mathit{bag}\ \alpha\ \mathit{with}\\
&\quad \mathit{nbr}\ ::\ \mathit{bag}\ \alpha \to \alpha \to \mathit{num}\\
&\quad \mathit{add},\ \mathit{del}\ ::\ \mathit{bag}\ \alpha \to \alpha \to \mathit{bag}\ \alpha\\
&\quad \mathit{empty}\ ::\ \mathit{bag}\ \alpha
\end{aligned}
$$

(De naam *abstype* komt van 'abstract type'.) Deze verklaring heeft alleen effect op de type-controle: de gelijkheid $\mathit{bag}\ \alpha\ \equiv\ \ldots$ is nu uitsluitend geldig *binnen* de definities van de grootheden die achter *with* staan genoemd. *Buiten* die definities wordt de type-controle uitgevoerd met de type-specificaties die in de *abstype*-verklaring zijn gegeven. Met andere woorden, buiten de definities van *nbr*, *add*, *del* en *empty* gedraagt $\mathit{bag}\ \alpha$ zich als een 'elementair' type, net als *num* en *bool* en *char*.

Dus door de abstype-verklaring is de uitdrukking $\mathit{hd}\ (\mathit{empty}\ \underline{\mathit{add}}\ 3)$ niet type-correct, en Miranda zal deze uitdrukking niet accepteren. Gelukkig maar, want als we de representatie wijzigen tot zoekbomen, dan is die uitdrukking onzinnig: een bag heeft geen 'head'.

§19.2 Include en export. Het is wenselijk om gescheiden taken ook gescheiden te kunnen programmeren. (Een programmeertaak kan dan over verscheidene personen verdeeld worden;

hergebruik van alreeds geschreven definities wordt vergemakkelijkt; enzovoorts.) Denk bijvoorbeeld aan enerzijds een stel functies om 'plaatjes' samen te stellen en af te drukken, en anderzijds het gebruik ervan om plaatjes van bomen te maken, of plaatjes van een kalender, enzovoorts.

Welnu, in Miranda kunnen we in een script, zeg `plaatje.m`, alle definities zetten die betrekking hebben op 'alleen plaatjes'. Die definities worden in een ander script door de volgende *aanwijzing* zichtbaar gemaakt:

$\%include$ "*plaatje.m*"

Het zou nu prettig zijn wanneer de correctheid van andere scripts (die `plaatje.m` insluiten) niet afhangt van de *hulp*functies die in `plaatje.m` zijn gebruikt. Want dan kunnen we te zijner tijd de definities voor de plaatjes-functies nog wijzigen (met name: in een of ander opzicht efficiënter maken en daarbij *andere* hulpfuncties gebruiken).

Welnu, stel dat de volgende aanwijzing voorkomt in `plaatje.m`:

$\%export\ + -h1\ -h2\ -h3$

Dan is de zichtbaarheid van identifiers $h1$, $h2$ en $h3$ beperkt tot uitsluitend script `plaatje.m`. Voor een gebruiker (insluiter) van `plaatje.m` lijkt het alsof $h1$, $h2$ en $h3$ gewoonweg niet gedefinieerd zijn in `plaatje.m`.

Met %*include* en %*export* kan de zichtbaarheid van identifiers uitvoeriger geregeld worden dan we hier hebben beschreven. Het volgende is ook mogelijk:

selectief insluiten (slechts sommige identifiers van een script importeren)
hernoemen van identifiers (bij het insluiten van een script)
zichtbaar maken van alle identifiers van een included script (bij %*export*)

Je kunt zelfs een heel script parametriseren: door middel van een %*free* aanwijzing. Alle informatie hierover staat in de on-line handleiding, in het hoofdstukje over de *Miranda Library Mechanism*.

Opgaven

216. In de context van §19.1, definieer een functie voor de 'vereniging' van twee bags en geef z'n type. Suggereer oplossingen, en alternatieve oplossingen, voor eventuele problemen.
217. In de context van §19.1, geef de representaties van een bag als een zoekboom, en pas de definities van de functies hieraan aan.
Welke definitie van 'vereniging' (gegeven in opgave 216) behoeft nu geen verandering?
218. In de context van §19.1, geef de representaties van een bag als een geordende lijst, en pas de definities van de functies hieraan aan.
Definieer een efficiënte 'vereniging' van twee aldus gerepresenteerde bags. Kunt u deze definitie correct typeren, gegeven de abstype-verklaring?
Wat is uw conclusie?

- Stacks
- Queues
- Pictures

20 Interactie

Het gebruik van een computer beperkt zich niet tot het uitrekenen van een Miranda-uitdrukking en het tonen van de uitkomst op het beeldscherm. Veel vaker is er sprake van een *interactie* tussen computer en gebruiker: afwisselend geeft de gebruiker toetsaanslagen en reageert de computer daarop (bijvoorbeeld met een tekst op het beeldscherm, of met kleurverandering of een piepje, enzovoorts).

Wij laten hier zien hoe interactie uitgedrukt kan worden.

§20.1 Statische interactie. Hier is een interactie in de vorm van *dialoog* met een vaste, 'voorgeprogrammeerde', input/output-verwerking: het spel *Hoog-Laag*. In dit spel tussen computer en gebruiker heeft de computer een getal 'in gedachte', laten we zeggen: 34, en is het de taak van de gebruiker om het getal te raden. Op ieder getal dat de gebruiker in tikt, verschijnt er de melding 'te hoog', 'te laag', of 'geraden'. Deze dialoog kunnen we heel eenvoudig met een functie modelleren. Afgezien van de input/output-verwerking wordt de dialoog gegeven door de functie *hooglaag*:

$$\begin{aligned}
&\mathit{hooglaag}\ (x : \mathit{xs}) \\
&= \texttt{"}\mathit{te\ hoog}\texttt{"} : \mathit{hooglaag}\ \mathit{xs},\ \mathit{if}\ 34 < x \\
&= \texttt{"}\mathit{te\ laag}\texttt{"} : \mathit{hooglaag}\ \mathit{xs},\ \mathit{if}\ 34 > x \\
&= \texttt{"}\mathit{geraden}\texttt{"} : [\,],\ \mathit{if}\ 34 = x
\end{aligned}$$

De input/output-verwerking zelf bestaat uit de transformatie van de toetsaanslagen tot een getallenrij, en de presentatie van de meldingen op het beeldscherm. De input, dat is de *char*-lijst met toetsaanslagen, wordt in Miranda aangeduid met `$-`; de inputverwerking is *map numval . lines*. De output-verwerking is eenvoudigweg *concat*. Dus de dialoog inclusief de IO-verwerking luidt:

$$\mathit{spel}\ =\ (\mathit{concat}\ .\ \mathit{hooglaag}\ .\ \mathit{map\ numval}\ .\ \mathit{lines})\ \$-$$

Voor veel dialogen gaat het net zo; enerzijds de functie die de dialoog modelleert waarbij wordt afgezien van de IO-verwerking, en anderzijds de IO-verwerking zelf als een voorbewerking op het argument en nabewerking op het functieresultaat.

Het wezenlijke hier is dat deze voor- en na-bewerking vantevoren vast staat, dus statisch is en niet afhangt van tussenresultaten van de voorafgaande berekening. Voor interacties met dit soort IO-verwerking is het stel functies dat wij hieronder geven *niet* nodig.

§20.2 Dynamische interactie. Hier is nogmaals het spel Hoog-Laag, maar nu uitgedrukt met behulp van de IO-verwerkingsfuncties *wrStr* en *rdNum* die we straks zullen definiëren:

$$\begin{aligned}
&\mathit{hooglaag}' \\
&= \mathit{rdNum}\ \underline{\mathit{sq}}\ f \\
&\quad \mathit{where} \\
&\quad f\ x \\
&\quad = \mathit{wrStr}\ \texttt{"}\mathit{te\ hoog}\texttt{"}\ \underline{\mathit{sqn}}\ \mathit{hooglaag}',\ \mathit{if}\ 34 < x
\end{aligned}$$

$$
\begin{aligned}
&= \ wrStr\ \texttt{"}te\ laag\texttt{"}\ \underline{sqn}\ hooglaag',\ if\ 34 > x \\
&= \ wrStr\ \texttt{"}geraden\texttt{"},\ otherwise \\
spel &= \ run\ hooglaag'
\end{aligned}
$$

Parameter x van *hooglaag* wordt nu bij $hooglaag'$ door *rdNum* van de invoer gehaald (en door *sq* "aan de locale functie f gegeven"), en de melding "*te hoog*" in het resultaat van *hooglaag* wordt nu bij $hooglaag'$ expliciet door *wrStr* aan de uitvoer toegevoegd. Functies *wrStr*, *rdNum* en $hooglaag'$ hebben alle een verborgen parameter en resultaat, namelijk het tweetal van de nog resterende input en de al geproduceerde output, genaamd $(input,\ output)$. Operaties *sqn* en *sq* combineren dit soort functies, en *run* zorgt voor de daadwerkelijke koppeling van het toetsenbord aan de input-parameter en van de output-parameter aan het beeldscherm. Een toepassing van $hooglaag'$ op de 'verborgen' $(input,\ output)$ verloopt als volgt, onder aanname dat *input* begint met het getal 99, dat wil zeggen $input = \texttt{"}99\backslash n\texttt{"}\ +\!\!+\ input'$:

$$
\begin{aligned}
& hooglaag'\ (input,\ output) \\
= & \qquad \{ \text{ definitie } hooglaag' \ \} \\
& (rdNum\ \underline{sq}\ f)\ (input,\ output) \\
= & \qquad \{ \text{ definitie } rdNum \text{ en } sq \text{ en aanname } input = \texttt{"}99\backslash n\texttt{"} +\!\!+ input' \ \} \\
& f\ 99\ (input',\ output) \\
= & \qquad \{ \text{ definitie } f\text{, conditie } 34 < 99 \text{ is vervuld } \} \\
& (wrStr\ \texttt{"}te\ hoog\texttt{"}\ \underline{sqn}\ hooglaag')\ (input',\ output) \\
= & \qquad \{ \text{ definitie } wrStr \text{ en } sqn \ \} \\
& hooglaag'\ (input',\ output +\!\!+ \texttt{"}te\ hoog\texttt{"})
\end{aligned}
$$

Dus er is één getal van de invoer gelezen, en één melding op de uitvoer geschreven, en met deze gewijzigde invoer en uitvoer wordt weer $hooglaag'$ aangeroepen.

Omdat het lezen van de invoer nu binnen $hooglaag'$ geprogrammeerd staat, is het mogelijk de bewerking op het restant van de invoer te laten afhangen van de alreeds verkregen tussenresultaten. Met andere woorden, deze opzet maakt dynamische IO-verwerking mogelijk.

§20.3 Systeembesturing. Soms is het gewenst dat de toetsaanslagen van de gebruiker niet op het beeldscherm getoond worden, zoals een wachtwoord of de 'te raden code' bij een spel zoals Hoog-Laag. Soms is het gewenst dat de kleur van het beeldscherm wijzigt, of dat de computer anderszins een reactie vertoont (anders dan alleen maar het tonen van een tekst op het beeldscherm). Vanuit een programmeursstandpunt beschouwd, wil je in deze gevallen niet zozeer een waarde berekenen en tonen, maar *een commando-reeks berekenen en door het onderliggende operating system laten uitvoeren.* Dat kan vanuit Miranda middels het standaard type *sys_message*.

In Miranda heeft type *sys_message* een bijzondere status: een output van het type [*sys_message*] wordt niet op het beeldscherm getoond, maar als commando-reeks naar het onderliggende operating system gestuurd. Bijvoorbeeld, de volgende lijst is van type [*sys_message*]:

$$[System\ \texttt{"}ls\ -l\texttt{"},\ Stdout\ \texttt{"}abc\texttt{"},\ System\ \texttt{"}date\texttt{"}]$$

Wanneer dit het resultaat van een berekening is, en dus niet 'getoond' wordt maar naar het operating system gestuurd wordt, gebeurt er het volgende:

het commando `ls -l` wordt uitgevoerd: op het scherm verschijnt een listing;
de tekst `abc` wordt naar de standaard output (het beeldscherm) gestuurd;
het commando `date` wordt uitgevoerd: op het scherm verschijnt de datum.

Het tonen van een willekeurige Miranda-waarde w kan dus ook: $Stdout\ (show\ w)$. We zullen daarom, zonder verlies van algemeenheid, alleen output van type *sys_message* beschouwen.

§20.4 IO-handlers. Een functie die de in- en uitvoer wijzigt, noemen we een *io-handler*. In het algemeen kan zo'n functie ook nog een resultaat opleveren waarvan de verdere berekening afhangt:

$$iohandler\ \alpha \ \equiv\ ([char],\ [sys\_message]) \to (([char],\ [sys\_message]),\ \alpha)$$

De definities van de basisfuncties liggen nu voor de hand; functies voor het produceren van output, voor het consumeren van input, en voor het koppelen van de input en output aan het operating system:

$$\begin{array}{l}
wrStr,\ do\ ::\ [char] \to iohandler\ ()\\
wrStr\ xs\ (input,\ output)\ =\ ((input,\ output +\!\!+ [Stdout\ xs]),\ ())\\
do\ xs\ (input,\ output)\ =\ ((input,\ output +\!\!+ [System\ xs]),\ ())\\
\\
rdChar\ ::\ iohandler\ char\\
rdChar\ (x : xs,\ output)\ =\ ((xs,\ output),\ x)\\
\\
run\ ::\ iohandler\ () \to [char]\\
run\ f\ =\ output' \quad where\ ((input',\ output'),\ x)\ =\ f\ (\$-, \texttt{""})
\end{array}$$

De operatie die twee io-handlers in sequentie aaneen schakelt, noemen we *sq*:

$$\begin{array}{l}
(f\ \underline{sq}\ g)\ (input,\ output)\\
=\ ((input2,\ output2),\ result2)\\
\quad where\\
\quad ((input1,\ output1),\ result1)\ =\ f\ (input,\ output)\\
\quad ((input2,\ output2),\ result2)\ =\ g\ result1\ (input1,\ output1)
\end{array}$$

Het α-resultaat van f wordt door sq aan g doorgegeven als zijn α-argument; het β-resultaat van g wordt het β-resultaat van de samenstelling $f\ \underline{sq}\ g$:

$$sq\ ::\ iohandler\ \alpha \to (\alpha \to iohandler\ \beta) \to iohandler\ \beta$$

Helaas, de definitie is *qua waarde* wel correct, maar niet *qua timing*: niets van f's output, $output1$, kan op het beeldscherm verschijnen voordat g's berekening voltooid is. Dus wanneer f een vraag-om-invoer op de output zet, en g van de de invoer leest, dan verschijnt de vraag-om-invoer pas (lang) nadat de invoer gelezen werd. Met de volgende definitie is dit euvel verholpen (alleen de =-regel is anders):

$$\begin{aligned}
&(f \ \underline{sq}\ \ g)\ (input,\ output)\\
&= ((input2,\ output1\ +\!\!+\ (output2 \mathbin{--} output1)),\ result2)\\
&\quad\ where\\
&\quad\ ((input1,\ output1),\ result1)\ =\ f\ (input,\ output)\\
&\quad\ ((input2,\ output2),\ result2)\ =\ g\ result1\ (input1,\ output1)
\end{aligned}$$

(Bedenk dat iedere io-handler de output alleen maar verlengt; dus $output2$ begint sowieso met $output1$, dat wil zeggen $output2 = output1\ +\!\!+\ (output2\ \mathbin{--}\ output1)$. Deze overweging suggereert bovendien een andere opzet voor de io-handlers; die zullen we in opgave 221 bespreken.) Miranda kan en zal $output1$ al tonen nog voordat de berekening van g gestart wordt.

Hier is nog een eenvoudige maar nuttige io-handler:

$$\begin{aligned}
&return\ ::\ \alpha \rightarrow iohandler\ \alpha\\
&return\ x\ (input,\ output)\ =\ ((input,\ output),\ x)
\end{aligned}$$

Hiermee hebben we de basis io-handlers gegeven; andere io-handlers zoals $rdStr$, $rdNum$ en sqn zijn hiermee uit te drukken. Die zullen we verderop bespreken.

Om de representatie-keuze af te schermen geven we nog deze verklaring:

$$\begin{aligned}
&abstype\ iohandler\ \alpha\ with\\
&\quad return\ ::\ \alpha \rightarrow iohandler\ \alpha\\
&\quad do,\ wrStr\ ::\ [char] \rightarrow iohandler\ ()\\
&\quad rdChar\ ::\ iohandler\ char\\
&\quad sq\ ::\ iohandler\ \alpha \rightarrow (\alpha \rightarrow iohandler\ \beta) \rightarrow iohandler\ \beta\\
&\quad run\ ::\ iohandler\ () \rightarrow [sys_message]
\end{aligned}$$

Zoals uitgelegd in §19.1 is de definitie van *iohandler* nu *buiten* de definities van *return*, ..., *run niet* bekend.

§20.5 Nog meer io-handlers. In de uitdrukking $f\ \underline{sq}\ g$ moet f een resultaat opleveren dat dan door *sq* als argument aan g doorgegeven wordt. Functies zoals *wrStr* leveren eigenlijk geen resultaat, en functies zoals *rdChar* verwachten geen argument. Voor de sequentiële aaneenschakeling van dit soort io-handlers definiëren we *sqn* (de 'n' staat voor 'no intermediate result'):

$$\begin{aligned}
&sqn\ \ ::\ iohandler\ () \rightarrow iohandler\ \alpha \rightarrow iohandler\ \alpha\\
&f\ \ \underline{sqn}\ \ g\ =\ f\ \ \underline{sq}\ \ const\ g
\end{aligned}$$

(Hierin is *const* de standaard functie met definitie: $const\ g\ x\ =\ g$.) Het lezen van een string bestaat uit het herhaaldelijk lezen van een karakters totdat $'\backslash n'$ is bereikt; de gelezen karakters worden door $rdStr'$ verzameld in een accumulatie-parameter:

$$\begin{aligned}
&rdStr\ ::\ iohandler\ [char]\\
&rdStr\ =\ rdStr'\ []
\end{aligned}$$

$$\begin{array}{l}
rdStr'\ xs \\
= rdChar\ \underline{sq}\ f \\
\quad where \\
\quad f\ x \\
\quad\ = return\ xs,\ if\ x = '\backslash n' \\
\quad\ = rdStr'\ (xs +\!\!+ [x]),\ otherwise
\end{array}$$

Het lezen van een getal (in z'n eentje genoteerd op één regel) bestaat uit het lezen van een string, en de toepassing van *numval* daarop:

$$rdNum = rdStr\ \underline{sq}\ (return\ .\ numval)$$

Op soortgelijke manier kunnen *rdBool*, *rdDigit* etcetera gedefinieerd worden, en *wrNum*, *wrBool* etcetera.

Om de echoing aan en uit te zetten, zijn er deze:

$$\begin{array}{l}
echoOff = do\ \texttt{"}stty\ -icanon\ -echo\ min\ 1\texttt{"} \\
echoOn = do\ \texttt{"}stty\ icanon\ echo\texttt{"}
\end{array}$$

De *stty*-commando's zijn specifiek voor het operating system dat de auteur gebruikt.

§20.6 IO-handling idioom. Bij bijna ieder voorkomen van $\underline{sq}$ komt er een where-part; kijk maar naar de tot nu toe gegeven voorbeelden. Aldus ontstaat er bij een reeks van $\underline{sq}$ operatoren een nesting van where-parts; schematisch:

$$\begin{array}{l}
linkerlid \\
= xxx1\ \underline{sqn}\ \ xxx2\ \underline{sq}\ f \\
\quad where \\
\quad f\ x \\
\quad\ = xxx3\ \underline{sqn}\ \ xxx4\ \underline{sq}\ f \\
\quad\quad where \\
\quad\quad f\ y \\
\quad\quad\ = xxx5\ \underline{sqn}\ \ xxx6\ \underline{sq}\ f \\
\quad\quad\quad where \\
\quad\quad\quad f\ z \\
\quad\quad\quad\ = xxx7a\ \underline{sqn}\ xxx7b\ \underline{sqn}\ xxx7c,\ if\ \ldots \\
\quad\quad\quad\ = xxx8a\ \underline{sqn}\ xxx8b\ \underline{sqn}\ xxx8c,\ if\ \ldots \\
\quad\quad\quad\ = xxx9a\ \underline{sqn}\ xxx9b\ \underline{sqn}\ xxx9c,\ if\ \ldots
\end{array}$$

De xxx... zijn io-handlers die door $\underline{sqn}$ en $\underline{sq}$ aaneen geschakeld zijn. Het resultaat van $xxx2$ wordt in de eerste f met x benoemd; die x kan verderop in $xxx3$ t/m $xxx9c$ gebruikt worden. Het resultaat van $xxx4$ wordt in de tweede f met y benoemd; die y kan verderop in $xxx5$ t/m $xxx9c$ gebruikt worden. Het resultaat van $xxx6$ wordt in de derde f met z benoemd; die z kan verderop in $xxx7$ t/m $xxx9$ gebruikt worden. Deze vorm kunnen we op een heel ongebruikelijke maar o zo handige manier opschrijven. Dat ziet er dan zó uit:

$$
\begin{array}{l}
\mathit{linkerlid} \\
= \mathit{xxx}1\ \underline{\mathit{sqn}} \\
\quad \mathit{xxx}2\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ x\ = \\
\quad \mathit{xxx}3\ \underline{\mathit{sqn}} \\
\quad \mathit{xxx}4\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ y\ = \\
\quad \mathit{xxx}5\ \underline{\mathit{sqn}} \\
\quad \mathit{xxx}6\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ z \\
\qquad = \mathit{xxx}7a\ \underline{\mathit{sqn}}\ \mathit{xxx}7b\ \underline{\mathit{sqn}}\ \mathit{xxx}7c,\ \mathit{if}\ \ldots \\
\qquad = \mathit{xxx}8a\ \underline{\mathit{sqn}}\ \mathit{xxx}8b\ \underline{\mathit{sqn}}\ \mathit{xxx}8c,\ \mathit{if}\ \ldots \\
\qquad = \mathit{xxx}9a\ \underline{\mathit{sqn}}\ \mathit{xxx}9b\ \underline{\mathit{sqn}}\ \mathit{xxx}9c,\ \mathit{if}\ \ldots
\end{array}
$$

Nu wordt duidelijk gesuggereerd dat $\mathit{xxx}1$ t/m $\mathit{xxx}6$ en dan nog één van $\mathit{xxx}7$..., $\mathit{xxx}8$... of $\mathit{xxx}9$... na elkaar uitgevoerd worden. Ook is duidelijk dat het resultaat van $\mathit{xxx}2$ met x benoemd wordt, het resultaat van $\mathit{xxx}4$ met y en het resultaat van $\mathit{xxx}6$ met z. Ieder deel '$\underline{\mathit{sq}}\ f\ \mathit{where}\ f\ v\ = \ldots$' kan gelezen worden als: 'en vervolgens, met v als naam voor het resultaat van de voorgaande io-handler, ...' Een voorbeeld volgt hieronder.

§20.7 Voorbeeld. Als voorbeeld van een echte interactie (met onder andere een regeling van de echoing) en van ons io-handling idioom geven we nu de definitieve versie van Hoog-Laag. In deze variant verschijnt er steeds op het scherm éérst een vraag, vóórdat de gebruiker wat moet intikken. Bovendien verschijnt het te raden getal niet op het beeldscherm wanneer dat wordt ingetikt:

$$
\begin{array}{l}
\mathit{hooglaag} \\
= \mathit{wrStr}\ \texttt{"}\mathit{Getal:}\ \texttt{"}\ \underline{\mathit{sqn}} \\
\quad \mathit{rdNumS}\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ k\ = \\
\quad \mathit{wrStr}\ \texttt{"}\mathit{Ra\ ra}\ldots\ \texttt{"}\ \underline{\mathit{sqn}} \\
\quad \mathit{hooglaag}'\ k \\
\\
\mathit{hooglaag}'\ x \\
= \mathit{wrStr}\ \texttt{"}\mathit{Gok:}\ \texttt{"}\ \underline{\mathit{sqn}} \\
\quad \mathit{rdNum}\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ n \\
\qquad = \mathit{wrStr}\ \texttt{"}\mathit{Te\ hoog.}\ \texttt{"}\ \underline{\mathit{sqn}}\ \mathit{hooglaag}'\ x,\ \mathit{if}\ x < n \\
\qquad = \mathit{wrStr}\ \texttt{"}\mathit{Te\ laag.}\ \texttt{"}\ \underline{\mathit{sqn}}\ \mathit{hooglaag}'\ x,\ \mathit{if}\ x > n \\
\qquad = \mathit{wrStr}\ \texttt{"}\mathit{Geraden!}\texttt{"},\ \mathit{if}\ x = n \\
\\
\mathit{spel}\ =\ \mathit{run\ hooglaag}
\end{array}
$$

De gebruikte hulpfunctie *rdNumS* (*S* staat voor 'Stil') luidt:

$$
\begin{array}{l}
\mathit{rdNumS} \\
= \mathit{echoOff}\ \underline{\mathit{sqn}} \\
\quad \mathit{rdNum}\ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ k\ = \\
\quad \mathit{wrStr}\ \texttt{"}\backslash n\texttt{"}\ \underline{\mathit{sqn}} \\
\quad \mathit{echoOn}\ \underline{\mathit{sqn}} \\
\quad \mathit{return}\ k
\end{array}
$$

Door de *wrStr* "\n" wordt als het ware de 'newline' van de gebruiker wél zichtbaar.

Opgaven

219. Pas de definitieve *hooglaag* aan zó dat bij het intikken van het te raden getal de toetsaanslagen met een streepje of sterretje getoond worden.
220. Pas de definitieve *hooglaag* aan zó dat er, na de vraag om een getal, een melding komt wanneer er niet-cijfers worden ingetikt, en opnieuw een getal wordt gevraagd en (op deze manier) wordt ingelezen.
221. Wijzig de representatie van io-handlers als volgt:

$$iohandler\ \alpha\ \equiv\ [char] \rightarrow ([char],\ [sys\_message]),\ \alpha)$$

Een io-handler krijgt nu niet meer de alreeds geproduceerde output mee om die te verlengen met nog meer output, maar levert, wat de output betreft, alleen "z'n eigen" output op. De operator $\underline{sq}$ moet de outputs van linker en rechter argument aaneen schakelen. Zie de opmerking bij de definitie van *sq* in §20.4.
Ga na dat er, dankzij de *abstype iohandler* verklaring, geen andere definitie dan die van *iohandler*, *wrStr*, *do*, *rdChar* en *run* gewijzigd hoeft te worden.
222. Stel dat we in de uitwerking van de vorige opgave 221 ook *rdStr* in de *abstype iohandler* verklaring hadden opgenomen, en gedefinieerd hadden:

$$\begin{aligned}
rdStr\ input\ &=\ ((dropline\ input,\ []),\ takeline\ input)\\
takeline\ xs\ &=\ takewhile\ (\neq\ '\backslash n')\ xs\\
dropline\ xs\ &=\ drop\ 1\ (dropwhile\ (\neq\ '\backslash n')\ xs)
\end{aligned}$$

Qua waarde (en typering) is deze definitie correct, maar niet qua timing! Wat gaat er mis? Hoe kan dit euvel verholpen worden?
223. Hier staat nogmaals ons io-handler idioom:

$$\begin{array}{l}
linkerlid\\
=\ xxx1\ \ \underline{sqn}\\
\quad xxx2\ \ \underline{sq}\ \ f\ \ where\ f\ x\ =\\
\quad xxx3\ \ \underline{sqn}\\
\quad xxx4\ \ \underline{sq}\ \ f\ \ where\ f\ y\ =\\
\quad xxx5\ \ \underline{sqn}\\
\quad xxx6\ \ \underline{sq}\ \ f\ \ where\ f\ z\\
\qquad =\ xxx7a\ \ \underline{sqn}\ \ xxx7b\ \ \underline{sqn}\ \ xxx7c,\ if\ \ldots\\
\qquad =\ xxx8a\ \ \underline{sqn}\ \ xxx8b\ \ \underline{sqn}\ \ xxx8c,\ if\ \ldots\\
\qquad =\ xxx9a\ \ \underline{sqn}\ \ xxx9b\ \ \underline{sqn}\ \ xxx9c,\ if\ \ldots
\end{array}$$

Zet vóór ieder van *xxx*1 t/m *xxx*6 een openingshaakje. Waar komen bijhorende sluithaakjes te staan?
224. Reconstrueer aan de hand van §20.3 een gedeeltelijke definitie van het type *sys_message*.

Opgaven

Deze serie opgaven is voor een practikum bedoeld.

225. Hangman. Er zijn twee spelers, waarvan één (de computer) een geheim woord kent, en de andere het woord moet raden door steeds één letter in te tikken. Na iedere gok wordt de score getoond: het woord waarin de nog niet geraden letters als een streepje worden getoond.

- ◦ Licht aan.
- ◦ Master Mind.
- ◦ Nim.
- ◦ Zet over: Boer Wolf Geit Kool.
- ◦ Calculator.

21 Nog te doen . . .

Wat ik nog had willen doen . . . .

§21.1 Ordening. De ordening op lijsten en tupels is *lexicografisch*, dat wil zeggen, net zo als in het woordenboek: het eerste verschil —van links naar rechts gelezen— tussen twee dingen bepaalt de onderlinge ordening. Bijvoorbeeld:

$$\begin{array}{l}
[] \;<\; ['a','b','c'] \;<\; ['a','c'] \;<\; ['a','c','d','e'] \\
[] \;<\; [1,2,3] \;<\; [1,3] \;<\; [1,3,4,5] \\
[] \;<\; [\mathit{True},\ \mathit{False}] \;<\; [\mathit{True},\ \mathit{True}] \;<\; [\mathit{True},\ \mathit{True},\ \mathit{False}] \\
(1,'b') \;<\; (2,'a') \;<\; (2,'b') \\
[] \;<\; [[1,2,3],\ [6,7,8],\ [9]] \;<\; [[1,3],\ [6,7,8],\ [9]]
\end{array}$$

De standaard functie *sort* sorteert zijn argumentlijst volgens de standaardordening $<$:

$$\begin{array}{lcl}
\mathit{sort}\ [3,\ 2,\ 4,\ 1] & = & [1,\ 2,\ 3,\ 4] \\
\mathit{sort}\ [[1,3],\ [1,2,3],\ [1,3,4,5],\ []] & = & [[],\ [1,2,3],\ [1,3],\ [1,3,4,5]] \\
\mathit{sort}\ [(2,'a'),\ (1,'b'),\ (2,'b')] & = & [(1,'b'),\ (2,'a'),\ (2,'b')]
\end{array}$$

De standaard functies *min* en *max* leveren het minimum en maximum op van hun argumentlijst:

$$\begin{array}{lcl}
\mathit{min}\ [3,\ 2,\ 4,\ 1] & = & 1 \\
\mathit{min}\ [[1,3],\ [1,2,3],\ [1,3,4,5],\ []] & = & [] \\
\mathit{min}\ [(2,'a'),\ (1,'b'),\ (2,'b')] & = & (1,'b')
\end{array}$$

Om te onthouden

• In Miranda zijn alle waarden geordend met $<$ etc. Functie *sort* levert de gesorteerde versie op van zijn argumentlijst, en *min* en *max* het minimum en maximum.

Oneindige lijsten

Recursieve lijstdefinities.
De oneindige lijst van alle faculteiten:

$$\mathit{facs} \;=\; 1 \;:\; [(n{+}1)\times\mathit{fn} \mid (n,\mathit{fn}) \leftarrow \mathit{zip}\ ([0..],\ \mathit{facs})\ ]$$

De oneindige lijst van alle Fibonacci-getallen:

$$\mathit{fibs} \;=\; 0 \;:\; 1 \;:\; \mathit{zipwith}\ (+)\ (\mathit{fibs},\ \mathit{tl}\ \mathit{fibs})$$

Opgaven

226. Priemgetallen op de manier van Eratosthenes.

227. Hamming: de rij van alle verschillende getallen die uitsluitend 2, 3 en 5 als factor hebben.
228. Geef een definitie van *regels* zó dat *lay regels* het volgende toont:

```
Regel 1 luidt:
"Regel 1 luidt:"
Regel 2 luidt:
""Regel 1 luidt:""
Regel 3 luidt:
"Regel 2 luidt:"
Regel 4 luidt:
"""Regel 1 luidt:"""
Regel 5 luidt:
"Regel 3 luidt:"
Regel 6 luidt:
""Regel 2 luidt:""
...
```

Doe het ook eens zó dat iedere regel van een regelnummer wordt voorzien.

Efficiëntie

Een paar woorden over efficiëntie.
Het precieze aantal rekenstappen is in Miranda (*lazy evaluation*) moeilijk te overzien.
Redeneer met grootte-orden van de gebruikte lijsten.

Backtracking

Veel leuke voorbeelden/opgaven!

Diverse opgaven

Opgaven

- Er geldt: $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. Geef een efficiënte uitdrukking voor de lijst van benaderingen:

$$[\ \sum_{i=0}^{n} \frac{x^i}{i!} \mid n \leftarrow [0, 1 \,..]\ ]$$

- Alle permutaties, op verschillende manieren.
- De lexicografisch eerstvolgende permutatie.

A Antwoorden

1. $gemiddelde\ x\ y\ =\ (x{+}y)/2$

   Nu geldt:

   $$\begin{array}{lll} gemiddelde\ 3\ 7 & = & 5 \\ gemiddelde\ 3\ 2{+}5 & = & (gemiddelde\ 3\ 2) + 5 \quad = \quad 7.5 \\ gemiddelde\ (3,\ 7) & & \text{is een foutieve uitdrukking.} \end{array}$$

2. $saldo\ r\ n\ b\ =\ (1 + 0.01{\times}r)\hat{}\,n \times b$

3. $hypotenusa\ a\ b\ =\ sqrt\ (a\hat{}2 + b\hat{}2)$

4. $blokOpp\ h\ d\ b$
   $= 2{\times}voorkant + 2{\times}zijkant + 2{\times}onderkant$
   $\quad where$
   $\quad voorkant\ =\ h{\times}b$
   $\quad zijkant\ =\ h{\times}d$
   $\quad onderkant\ =\ b{\times}d$

5. $cirkelOpp\ r\ =\ pi \times r\hat{}2$

   Voorbeeld aanroepen:
   $cirkelOpp\ (3{+}4)$
   $cirkelOpp\ 7$
   $cirkelOpp\ 3{+}4\ \|\ dit\ is: \quad (cirkelOpp\ 3) + 4$

6. $bolVol\ r\ =\ 4/3 \times pi \times r\hat{}3$

7. $tan\ x\ =\ sin\ x\ /\ cos\ x$

8. $somRR\ a\ v\ n\ =\ n{\times}a + v{\times}n{\times}(n{-}1)/2 \qquad \|\ =\ n \times 1/2 \times (a + (n{-}1){\times}v + a)$

   Voorbeelden van gebruik:
   $somRR\ 1\ 1\ 10\quad \|\ =\ 1{+}2{+}\ldots{+}10\ =\ 55$
   $somRR\ 23\ 1\ 21\quad \|\ =\ 23{+}24{+}\ldots{+}43\ =\ 693$
   $somRR\ 1\ 33\ 53\quad \|\ =\ 1{+}34{+}\ldots{+}1717\ =\ 45527$

9. $lengteRR\ a\ b\ z\ =\ (z{-}a)/(b{-}a) + 1$
   $somRR'\ a\ b\ z\ =\ somRR\ a\ (b{-}a)\ (lengteRR\ a\ b\ z)$

   Inderdaad, $somRR\ 1\ (34{-}1)\ (lengteRR\ 1\ 34\ 1717)$ levert 45527.

10. We geven verscheidene varianten, genaamd $max3$, $max3'$, $max3''$, enzovoorts. De eerste variant heeft onze voorkeur.
    $max3\ x\ y\ z\ =\ x\ \underline{max2}\ y\ \underline{max2}\ z$
    $max3'\ x\ y\ z\ =\ max2\ (max2\ x\ y)\ z$

$max3''\ x\ y\ z$
$= \ x,\ if\ x \geq y\ \wedge\ x \geq z$
$= \ y,\ if\ y \geq x\ \wedge\ y \geq z$
$= \ z,\ if\ z \geq x\ \wedge\ z \geq y$
$max3'''\ x\ y\ z$
$= \ x,\ if\ x \geq y\ \wedge\ x \geq z$
$= \ y,\ if\ y \geq z$
$= \ z,\ otherwise$

Bij het uitrekenen van $max3'''$ worden de drie alternatieven in de opgegeven volgorde "geprobeerd". Dus in het tweede alternatief geldt dat de bewakingsconditie van het eerste alternatief niet vervuld is; x is dus niet de grootste, en y of z wel. Net zo geldt in het derde alternatief dat beide voorafgaande bewakingscondities niet vervuld zijn, en dus x en y niet de grootste zijn.

$max4\ w\ x\ y\ z\ =\ w\ \underline{max2}\ x\ \underline{max2}\ y\ \underline{max2}\ z$
$max4'\ w\ x\ y\ z$
$= \ w,\ if\ w \geq x\ \wedge\ w \geq y\ \wedge\ w \geq z$
$= \ \ldots\ etc\ \ldots$

Deze laatste definitie-vorm kost erg veel schrijfwerk.

11. $sign\ x$
$= \ -1,\ if\ x < 0$
$= \ 0,\ if\ x = 0$
$= \ 1,\ if\ x > 0$
$sign'\ x$
$= \ 0,\ if\ x = 0$
$= \ entier\ (x\ /\ abs\ x),\ otherwise$

12. $eenheid\ n\ =\ n\ mod\ 10$

13. Met hulpfuncties $vrnl'$, $vrnl''$, $vrnl'''$ die net als $vrnl$ werken maar alleen voor getallen met 1, respectievelijk 2 en 3, cijfers:

$vrnl\ n$
$= \ vrnl'\ n,\ if\ 0 \leq n < 10$
$= \ vrnl''\ n,\ if\ 10 \leq n < 100$
$= \ vrnl'''\ n,\ if\ 100 \leq n < 1000$
$vrnl'\ n\ \|\ aanname:\ 0 \leq n < 10$
$= \ n$
$vrnl''\ n\ \|\ aanname:\ 10 \leq n < 100$
$= \ (n\ mod\ 10){\times}10 + vrnl'\ (n\ div\ 10)$
$vrnl'''\ n\ \|\ aanname:\ 100 \leq n < 1000$
$= \ (n\ mod\ 10){\times}100 + vrnl''\ (n\ div\ 10)$

Merk op dat de aannamen over de parameters bij elk gebruik van de functie vervuld zijn. Functie $vrnl'$ kan weggewerkt worden, maar brengt de systematiek duidelijk naar voren.
Een andere variant; nu met $vrnl''$ die voor getallen met ten hoogste twee cijfers werkt, en $vrnl'$ voor getallen met ten hoogste één cijfer:

$vrnl\ n$

$= \ vrnl''\ n,\ \mathit{if}\ 0 \le n < 100$
$= \ (n\ mod\ 10) \times 100 + vrnl''\ (n\ div\ 10),\ \mathit{if}\ 100 \le n < 1000$
$vrnl''\ n$
$= \ vrnl'\ n,\ \mathit{if}\ 0 \le n < 10$
$= \ (n\ mod\ 10) \times 10 + vrnl'\ (n\ div\ 10),\ \mathit{if}\ 10 \le n < 100$
$vrnl'\ n \ = \ n,\ \mathit{if}\ 0 \le n < 10$

Alweer kan *vrnl′* weggewerkt worden.

14. $isCijfer\ x \ = \ '0' \le x \le '9'$
$hLetter\ x$
$= \ x,\ \mathit{if}\ ishLetter\ x$
$= \ decode\ (code\ x + code\ 'A' - code\ 'a'),\ \mathit{if}\ iskLetter\ x$

Twee lelijke alternatieven voor *isCijfer*:

$isCijfer\ x$
$= \ True,\ \mathit{if}\ '0' \le x \le '9'$
$= \ False,\ otherwise$
$isCijfer\ x \ = \ code\ '0' \le code\ x \le code\ '9'$

15. $cijferNgetal\ x$
$= \ code\ x - code\ '0',\ \mathit{if}\ isCijfer\ x$
$= \ error\ "cijferNgetal:\ argument\ is\ geen\ cijfer",\ otherwise$

16. $getalNcijfer\ n$
$= \ decode\ (code\ '0' + n),\ \mathit{if}\ 0 \le n < 10$
$= \ error\ "getalNcijfer:\ argument\ niet\ in\ 0 \,..\, 9",\ otherwise$

17. $letterNa\ x$
$= \ 'A',\ \mathit{if}\ x \ = \ 'Z'$
$= \ 'a',\ \mathit{if}\ x \ = \ 'z'$
$= \ decode\ (code\ x + 1),\ \mathit{if}\ 'A' \le x < 'Z' \ \vee\ 'a' \le x < 'z'$

De aanroep $letterNa\ '\#'$ resulteert nu in een foutstop. Vervangen we de laatste bewaking door *otherwise*, dan heeft $letterNa\ x$ voor alle letters x nog steeds de goede uitkomst, maar wordt er bij $x = '\#'$ ook iets opgeleverd.

18. $piep\ n$
$= \ n\ mod\ 7 = 0 \ \vee$
$\quad n\ mod\ 10 = 7 \ \vee\ (n\ div\ 10)\ mod\ 10 = 7 \ \vee\ n\ div\ 100 = 7$

De condities testen achtereenvolgens of n een zevenvoud is, of het eenheden-cijfer van n zeven is, of het tientallen-cijfer van n zeven is, en of het hondertallen-cijfer van n (met $n < 1000$) zeven is.

19. a. $\#[] \ = \ 0$; de lege lijst heeft geen elementen, de lengte ervan is 0.
b. $\#[[1,2],[3],[4,5,6]] = 3$; er zijn hier drie elementen, namelijk de lijsten $[1,2]$ en $[3]$ en $[4,5,6]$. Vergis je dus niet in “de getallen die te zien zijn” en de *elementen* van de lijst.
c. $\#[[]] = 1$; er is geen getal te zien, maar toch heeft de lijst $[[]]$ één element, namelijk de lege lijst $[]$.
d. $\#[[[]]] = 1$; er is weer één element, namelijk $[[]]$.

20. $somRR'\ a\ b\ z\ =\ sum\ [a,\ b\ ..\ z]$

21. $somRR\ a\ v\ n\ =\ sum\ (take\ n\ [a,\ a{+}v\ ..])$

22. $fac\ n\ =\ product\ [1\,..\,n]$
Er geldt nu $fac\ 0\ =\ 1$.

23. $index\ xs\ =\ take\ (\#xs)\ [0..]$

24. $hd\ xs\ =\ xs!0$
$tl\ xs\ =\ drop\ 1\ xs,\ if\ xs \neq []$
$init\ xs\ =\ take\ (\#xs{-}1)\ xs,\ if\ xs \neq []$
$last\ xs\ =\ xs!(\#xs{-}1)$
Zonder de bewakingen zouden *tl* [] en *init* [] niet in een foutstop resulteren.

25. Drie alternatieven voor *segment*; de eerste heeft onze voorkeur:
$segment\ xs\ i\ j\ =\ drop\ i\ (take\ j\ xs)$
$segment\ xs\ i\ j\ =\ take\ (j{-}i)\ (drop\ i\ xs)$
$segment\ xs\ i\ j\ =\ [xs!k \mid k \leftarrow [i\ ..\ j{-}1]]$

26. Met gebruik van *segment*:
$swap\ xs\ i\ j$
$=\ segment\ xs\ 0\ i'\ +\!\!+\ [xs!j']\ +\!\!+\ segment\ xs\ (i'{+}1)\ j'\ +\!\!+\ [xs!i']\ +\!\!+$
$\quad segment\ xs\ (j'{+}1)\ (\#xs)$
$\quad where$
$\quad (i',\ j')\ =\ (min2\ i\ j,\ \ max2\ i\ j)$
Het is goed te zien dat alle elementen van *xs* ook voorkomen in *swap xs i j*. We hadden de eerste en laatste aanroep van segment ook gemakkelijk met *take* en *drop* uit kunnen schrijven. Zonder *segment* kunnen we *swap* ook definiëren door:
$swap\ xs\ i\ j$
$=\ [\ xs\ !\ f\ k \mid k \leftarrow index\ xs]$
$\quad where$
$\quad f\ k$
$\quad =\ i,\ if\ k = j$
$\quad =\ j,\ if\ k = i$
$\quad =\ k,\ otherwise$

27. $rotatieL\ xs\ =\ drop\ 1\ xs\ +\!\!+\ take\ 1\ xs$
$rotatieR\ xs\ =\ drop\ (n{-}1)\ xs\ +\!\!+\ take\ (n{-}1)\ xs\ \ where\ n = \#xs$
Deze functies leveren ook voor de lege lijst een resultaat op (de lege lijst). Met onderstaande definities resulteert er voor de lege lijst een foutstop:
$rotatieL\ xs\ =\ tl\ xs\ +\!\!+\ [hd\ xs]$
$rotatieR\ xs\ =\ [last\ xs]\ +\!\!+\ init\ xs$
Let er op dat je $hd\ xs$ ($= xs!0$) en $last\ xs$ ($= xs!(\#xs{-}1)$) eerst in een lijst zet, alvorens ze aan $+\!\!+$ te onderwerpen.

28. In de eerste uitdrukking staat: '… $x-y=6;\ y \leftarrow [1..32]$ …'. Maar de 'y' uit '$y \leftarrow [1..32]$' mag uitsluitend rechts ervan gebruikt worden (en hélemaal links in de lijstcomprehensie).
In de tweede uitdrukking staat: '$y \leftarrow x-6$'. Dat is niet geoorloofd, want rechts van de '$y \leftarrow$ ' moet een lijst staan, desnoods een lijst met maar één element, zoals $[x-6]$.
In de derde uitdrukking staat: '$y = x-6$'. Dat is een conditie, een test; en in deze uitdrukking is 'y' nog niet geïntroduceerd. Een introductie van 'y' luidt bijvoorbeeld: '$y \leftarrow [x-6]$'.
29. Ja, de lijsten hebben dezelfde drietallen, maar in een andere volgorde. In de lijst met '$x, y, z \leftarrow [1..32]$' staat het drietal met de kleinste x voorop; in de lijst met '$z, y, x \leftarrow [1..32]$' staat het drietal met de kleinste z voorop.
30. $\#\,[(x,y,z) \mid x,y,z \leftarrow [1..32];\ x-y=6;\ z+x=17;\ y\times z=18]$
Alternatief (minder makkelijk te veralgemenen tot varianten van de puzzel):
2

31. $$\begin{aligned}&\mathit{opln}\ n\ (a,b,c)\\ &= [(x,y,z) \mid x \leftarrow [1..n];\ y \leftarrow [x-a];\ 1 \le y \le n;\ z \leftarrow [b-x];\\ &\qquad 1 \le z \le n;\ y\times z = c]\end{aligned}$$
Zonder de condities $1 \le y \le n$ en $1 \le z \le n$ worden er soms verkeerde resultaten opgeleverd. Bijvoorbeeld, wanneer $a,b,c = 2,1,0$ dan wordt ook $(x,y,z) = (1,-1,0)$ opgeleverd, en dat is niet goed volgens de specificatie.
Alle oplossingen van de puzzel: $\mathit{opln}\ 32\ (6,17,18)$.
32. $$\begin{aligned}&\mathit{puzzels}\ k\\ &= [(n,(a,b,c)) \mid\\ &\qquad n\leftarrow[0..15];\ a\leftarrow[0..n];\ b\leftarrow[0..2\times n];\ c\leftarrow[0..n\times n];\\ &\qquad \#\mathit{opln}\ n\ (a,b,c)\ =\ k]\\ &\mathit{puzzels}'\ k\ \mathit{nMax}\\ &= [(n,(a,b,c)) \mid\\ &\qquad n\leftarrow[0..\mathit{nMax}];\ a\leftarrow[0..n];\ b\leftarrow[0..2\times n];\ c\leftarrow[0..n\times n];\\ &\qquad \#\mathit{opln}\ n\ (a,b,c) = k]\end{aligned}$$

33. Twee alternatieven; de eerste heeft onze voorkeur:
$$\begin{aligned}&x\ \underline{\mathit{aantal}}\ xs\ =\ \#\,[1 \mid x' \leftarrow xs;\ x' = x]\\ &x\ \underline{\mathit{aantal}}\ xs\ =\ \mathit{sum}\ [1 \mid x' \leftarrow xs;\ x' = x]\end{aligned}$$
In het eerste alternatief doet het er niet toe wat er op de plaats van de '1' staat; het mag bijvoorbeeld ook x' of x zijn.
34. Drie alternatieven:
$$\begin{aligned}&\mathit{member}\ xs\ x\ =\ [1 \mid x'\leftarrow xs;\ x' = x]\ \neq\ []\\ &\mathit{member}\ xs\ x\ =\ \#\,[1 \mid x' \leftarrow xs;\ x' = x]\ >\ 0\\ &\mathit{member}\ xs\ x\ =\ x\ \underline{\mathit{aantal}}\ xs\ >\ 0\end{aligned}$$

35. $\mathit{delers}\ n\ =\ [d \mid d\leftarrow[2..n-1];\ n\ \mathit{mod}\ d\ =\ 0]$
Bij argument 0 en 1 wordt de lege lijst opgeleverd.
Alle positieve delers van n: $[1]\ +\!\!+\ \mathit{delers}\ n\ +\!\!+\ [n]$; voor $n=1$ is dit de lijst $[1,1]$.
36. $\mathit{priem}\ n\ =\ n \ge 2\ \wedge\ \mathit{delers}\ n\ =\ []$

37. $x \ \underline{ggd}\ y \ = \ max\ [d \mid d \leftarrow [1\,..\ min2\ x\ y];\ x\ mod\ d \ = \ 0;\ y\ mod\ d \ = \ 0]$

Het genereren van kandidaat-delers, $d \leftarrow [1\,..\ min2\ x\ y]$, kan nog ingeperkt worden. Bijvoorbeeld, als d een deler is van x, dan geldt $d \leq \sqrt{x} \vee d = x$. Dus:

$\ldots\ d \leftarrow [1\ ..\ min2\ (sqrt\ x)\ (sqrt\ y)]\ +\!\!+\ [min2\ x\ y]\ \ldots$

We kunnen de berekening van max vermijden door de kandidaat-delers van groot naar klein te genereren, en dan het eerste element ervan te nemen:

$$
\begin{aligned}
& x\ \underline{ggd}\ y \\
= \ & hd\ [d \mid d{\leftarrow}[m,\ m{-}1\ ..\ 1];\ x\ mod\ d = 0;\ y\ mod\ d = 0] \\
& \quad where \\
& \quad m \ = \ min2\ x\ y
\end{aligned}
$$

38. $pythagoras\ n \ = \ [(x,y,z) \mid x{\leftarrow}[1\,..\,n];\ y{\leftarrow}[x\,..\,n];\ z{\leftarrow}[y\,..\,2{\times}n];\ x\hat{\ }2{+}y\hat{\ }2 = z\hat{\ }2]$

De lijst van mogelijke waarden voor z is ruim genoeg: bij maximale x en y is bijhorende z gelijk aan $\sqrt{2n^2}$ en dat is kleiner dan $2n$. De lijst voor z kan verder ingeperkt worden, zelfs tot een singleton lijst: bij gegeven x en y is $\sqrt{x^2{+}y^2}$, afgekapt tot een geheel getal, de enige mogelijkheid voor z:

$$
\begin{aligned}
& pythagoras\ n \\
= \ & [(x,y,z) \mid x{\leftarrow}[1\,..\,n];\ y{\leftarrow}[x\,..\,n];\ z{\leftarrow}[entier(sqrt(x\hat{\ }2{+}y\hat{\ }2))]; \\
& \quad x\hat{\ }2{+}y\hat{\ }2 = z\hat{\ }2]
\end{aligned}
$$

De dubbele berekening van $x\hat{\ }2{+}y\hat{\ }2$ kan desgewenst ook vermeden worden:

$$
\begin{aligned}
& pythagoras\ n \\
= \ & [(x,y,z) \mid x{\leftarrow}[1\,..\,n];\ y{\leftarrow}[x\,..\,n];\ k{\leftarrow}[x\hat{\ }2{+}y\hat{\ }2]; \\
& \quad z{\leftarrow}[entier\ (sqrt\ k)];\ k = z\hat{\ }2]
\end{aligned}
$$

39. Benoem de onbekenden van links naar rechts en van boven naar beneden met $a, \ldots, f$.

$$
\begin{aligned}
& opl \\
= \ & [(a,b,c,d,e,f) \mid a,b,c,d,e,f \leftarrow [1\,..\,9]; \\
& \quad 10{\times}a{+}b \ < \ 20; \\
& \quad c{+}d{+}e \ = \ 10{\times}a{+}b; \\
& \quad 6 \times (100{\times}a{+}10{\times}d{+}f) \ = \ 100{\times}c{+}10{\times}d{+}e; \\
& \quad 10{\times}b{+}e + 10{\times}e{+}b \ = \ 100{\times}a{+}10{\times}d{+}f\ ]
\end{aligned}
$$

Door de voorwaarden "zo vroeg mogelijk" te vermelden, wordt de berekeningstijd aanzienlijk verkort:

$$
\begin{aligned}
& opl \\
= \ & [(a,b,c,d,e,f) \mid \\
& \quad a \leftarrow [1];\ b,c,d,e \leftarrow [1\,..\,9];\ c{+}d{+}e \ = \ 10{\times}a{+}b;\ f \leftarrow [1\,..\,9]; \\
& \quad 6 \times (100{\times}a{+}10{\times}d{+}f) \ = \ 100{\times}c{+}10{\times}d{+}e;\ 10{\times}b{+}e + 10{\times}e{+}b \ = \ 100{\times}a{+}10{\times}d{+}f\ ]
\end{aligned}
$$

40. Twee alternatieven voor $dias$:

$$
\begin{aligned}
& dias \ = \ [(i,j) \mid n \leftarrow [0.\,.];\ i \leftarrow [0\,..\,n];\ j \leftarrow [n{-}i]] \\
& dias \ = \ [(i,n{-}i) \mid n \leftarrow [0.\,.];\ i \leftarrow [0\,..\,n]] \\
& dias100 \ = \ take\ 100\ dias
\end{aligned}
$$

41. $inMunten\ x$

$= \ [(s,d,k,g) \mid$
$\quad s{\leftarrow}[0\,..\,x/5];\ d{\leftarrow}[0\,..\,x/10];\ k{\leftarrow}[0\,..\,x/25];\ g{\leftarrow}[0\,..\,x/100];$
$\quad 5{\times}s + 10{\times}d + 25{\times}k + 100{\times}g \ = \ x]$

Als x geen vijfvoud is, wordt de lege lijst opgeleverd omdat de laatste conditie dan niet vervuld kan zijn. Een andere manier is om éérst de guldens te kiezen, dán de kwartjes, etc. Dan is het restbedrag zeker in munten uit te betalen, als het oorspronkelijke bedrag wel een vijfvoud is. We doen dit nu, en beperken tevens steeds de mogelijke keuzen voor de munten voor het restbedrag (als er al guldens, kwartjes, etc zijn gekozen):

$$
\begin{array}{l}
\mathit{inMunten}\ x \\
\quad = \ [\,],\ \mathit{if}\ x\ \mathit{mod}\ 5\ \neq\ 0 \\
\quad = \ [(s,d,k,g) \mid \\
\qquad g \leftarrow [0\ ..\ x/100]; \\
\qquad k \leftarrow [0\ ..\ (x{-}100{\times}g)/25]; \\
\qquad d \leftarrow [0\ ..\ (x{-}100{\times}g{-}25{\times}k)/10]; \\
\qquad s \leftarrow [(x{-}100{\times}g{-}25{\times}k{-}10{\times}d)\ \mathit{div}\ 5]],\ \mathit{otherwise}
\end{array}
$$

Om zo min mogelijk munten te krijgen, moeten we zoveel mogelijk van de waardevolste munten kiezen:

$$
\begin{array}{l}
\mathit{inMunten'}\ x \\
\quad = \ \mathit{error}\ \texttt{"}\mathit{inMunten'}:\ \mathit{argument\ geen\ vijfvoud}\texttt{"}, \\
\qquad \mathit{if}\ x\ \mathit{mod}\ 5\ \neq\ 0 \\
\quad = \ (s,d,k,g),\ \mathit{otherwise} \\
\qquad \mathit{where} \\
\qquad g \ = \ x\ \mathit{div}\ 100 \\
\qquad k \ = \ (x\ -\ 100{\times}g)\ \mathit{div}\ 25 \\
\qquad d \ = \ (x\ -\ 100{\times}g\ -\ 25{\times}k)\ \mathit{div}\ 10 \\
\qquad s \ = \ (x\ -\ 100{\times}g\ -\ 25{\times}k\ -\ 10{\times}d)\ \mathit{div}\ 5
\end{array}
$$

42. Met gebruik van functie *vrnl* van opgave 13 (of de variant die argumenten van 0 tot en met 99 aan kan):

$$[(i,j) \mid i,j \leftarrow [0\,..\,99];\ i \times j \ = \ \mathit{vrnl}\ i \times \mathit{vrnl}\ j]$$

Een andere manier is om de cijferwaarden apart te genereren, en daaruit de getallen i, i', j, j' op te bouwen:

$$
\begin{array}{l}
[(i,j) \mid \\
\quad a,b \leftarrow [0\,..\,9];\ i \leftarrow [10{\times}a{+}b];\ i' \leftarrow [10{\times}b{+}a]; \\
\quad c,d \leftarrow [0\,..\,9];\ j \leftarrow [10{\times}c{+}d];\ j' \leftarrow [10{\times}d{+}c]; \\
\quad i{\times}j = i'{\times}j']
\end{array}
$$

43.

$$
\begin{array}{l}
\mathit{tels} \ = \ [\mathit{snd}\ x \mid x \leftarrow \mathit{telboek}] \\
\mathit{tel}\ \mathit{nm} \ = \ [\mathit{snd}\ x \mid x \leftarrow \mathit{telboek};\ \mathit{fst}\ x \ = \ \mathit{nm}] \\
\mathit{abos} \ = \ [\mathit{fst}\ x \mid x \leftarrow \mathit{telboek}] \\
\mathit{abo}\ \mathit{nr} \ = \ [\mathit{fst}\ x \mid x \leftarrow \mathit{telboek};\ \mathit{snd}\ x \ = \ \mathit{nr}]
\end{array}
$$

In de lijsten *tels* en *abos* kunnen elementen dubbel voorkomen (als meerdere personen eenzelfde telefoonnummer hebben, of een persoon meerdere telefoonnummers heeft). We kunnen de definities iets mooier formuleren:

$$\mathit{tel}\ \mathit{nm} \ = \ [\mathit{nr} \mid (\mathit{nm'}, \mathit{nr}) \leftarrow \mathit{telboek};\ \mathit{nm'} = \mathit{nm}]$$

$abo\ nr\ =\ [nm \mid (nm, nr') \leftarrow telboek;\ nr' = nr]$
$tels\ =\ [nr \mid (nm, nr) \leftarrow telboek]$
$abos\ =\ [nm \mid (nm, nr) \leftarrow telboek]$

44. $stdDeviatie\ xs$
$= sqrt\ (sum\ [(x{-}xgem)\hat{}2 \mid x \leftarrow xs]\ /\ (n{-}1))$
$where$
$xgem\ =\ sum\ xs\ /\ n$
$n\ =\ \#xs$

45. a. [] b. $[0, 1]$ c. []
46. De lijst $[x{\times}y \mid x \leftarrow [1\,..\,4];\ y \leftarrow [3\,..\,8]]$ heeft lengte 24.
Er geldt: $\#[expr \mid x \leftarrow xs;\ y \leftarrow ys] = \#xs \times \#ys$.
Wanneer xs en/of ys leeg is, geldt $[expr \mid x \leftarrow xs;\ y \leftarrow ys] = []$.
47. $\#mannen$
$\#vrouwen$

48. $moeders\ x\ =\ [v \mid v \leftarrow vrouwen;\ kinderen\ v\ \underline{member}\ x]$
$ouders\ x\ =\ vaders\ x\ +\!\!+\ moeders\ x$
$oudertal\ x\ =\ \#\ ouders\ x$
Voor *dochters* liggen twee alternatieven voor de hand:
$dochters\ x\ =\ [v \mid v \leftarrow vrouwen;\ kinderen\ x\ \underline{member}\ v]$
$dochters\ x\ =\ [k \mid k \leftarrow kinderen\ x;\ vrouwen\ \underline{member}\ k]$
Net zo voor *zonen*:
$zonen\ x\ =\ [m \mid m \leftarrow mannen;\ kinderen\ x\ \underline{member}\ m]$
$zonen\ x\ =\ [k \mid k \leftarrow kinderen\ x;\ mannen\ \underline{member}\ k]$
Voor *zussen* zijn er verscheidene alternatieven:
$zussen\ x\ =\ [d \mid m \leftarrow moeders\ x;\ d \leftarrow dochters\ m;\ d \neq x]$
$zussen\ x\ =\ [d \mid v \leftarrow vaders\ x;\ d \leftarrow dochters\ v;\ d \neq x]$
$zussen\ x\ =\ [d \mid o \leftarrow ouders\ x;\ d \leftarrow dochters\ o;\ d \neq x]$
$zussen\ x\ =\ [v \mid v \leftarrow vrouwen;\ v \neq x;\ ov \leftarrow ouders\ v;\ ouders\ x\ \underline{member}\ ov]$
$zussen\ x\ =\ [v \mid v \leftarrow vrouwen;\ v \neq x;\ ox \leftarrow ouders\ x;\ ouders\ v\ \underline{member}\ ox]$
Bij de eerste twee alternatieven worden geen halfzussen opgeleverd; toevallig zijn die er ook niet in ons gegevensbestand. Bij het derde alternatief kunnen er zussen dubbel voorkomen. Bijvoorbeeld, "*Margriet*" is een zus van "*Beatrix*" omdat ze een dochter is van Bea's moeder, maar ook van Bea's vader. In de laatste twee alternatieven worden weer geen halfzussen opgeleverd.
De definities voor *broers* zijn analoog aan die voor *zussen*.
$brussen\ x\ =\ broers\ x\ +\!\!+\ zussen\ x$
$tantes\ x\ =\ [z \mid o \leftarrow ouders\ x;\ z \leftarrow zussen\ o]$
$tantes\ x\ =\ [v \mid v \leftarrow vrouwen;\ o \leftarrow ouders\ x;\ zussen\ o\ \underline{member}\ v]$
$kleinkinderen\ x\ =\ [kk \mid k \leftarrow kinderen\ x;\ kk \leftarrow kinderen\ k]$
$grootouders\ x\ =\ [go \mid o \leftarrow ouders\ x;\ go \leftarrow ouders\ o]$
$x\ \underline{isVaderVan}\ y\ =\ vaders\ y\ \underline{member}\ x$
$x\ \underline{isBrusVan}\ y\ =\ brussen\ x\ \underline{member}\ y$

Voor sommige functies zijn er nog andere alternatieven.
Twee uitdrukkingen voor 'Jorge is een vader van Jaime':

$vaders$ "$Jaime$" $\underline{member}$ "$Jorge$"
"$Jorge$" $\underline{isVaderVan}$ "$Jaime$"

49. $concat\ xss \ = \ [x \mid xs \leftarrow xss;\ x \leftarrow xs]$

50. $inits\ xs \ = \ [take\ i\ xs \mid i \leftarrow [0\ldots \#xs]]$

Er geldt: $\#inits\ xs = 1 + \#xs$.

51. $tails\ xs \ = \ [drop\ i\ xs \mid i \leftarrow [0\ldots \#xs]]$

Er geldt: $\#tails\ xs = 1 + \#xs$.

52. Twee alternatieven voor *group*:

$group\ n\ xs \ = \ [take\ n\ (drop\ j\ xs) \mid j \leftarrow [0,\ n\ \ldots\ \#xs{-}1]\ ]$
$group\ n\ xs \ = \ [take\ n\ (drop\ (i{\times}n)\ xs) \mid i \leftarrow [0\ \ldots\ (\#xs{-}1)\ div\ n]]$

Merk op dat de 'j' in de eerste definitie exact overeenkomt met de '$i{\times}n$' in de tweede. Er geldt altijd $concat\ (group\ 3\ xs) \ = \ xs$, en alleen wanneer xss uitsluitend lijsten ter lengte 3 bevat, geldt ook $group\ (concat\ xss) \ = \ xss$.

En twee alternatieven voor *ggroup*:

$ggroup\ m\ n\ xs \ = \ group\ n\ (group\ m\ xs)$
$ggroup\ m\ n\ xs \ = \ [group\ m\ xs' \mid xs' \leftarrow group\ (m{\times}n)\ xs]$

53. We geven een paar alternatieven. Functies *index* en *concat* zijn standaard functies (zie opgaven 23 en 49). Functies *inits*, *tails*, en *segment* staan in opgaven 50, 51, en 25.

$segs\ xs \ = \ [zs \mid ys \leftarrow inits\ xs;\ zs \leftarrow tails\ ys]$
$segs\ xs \ = \ concat\ [tails\ ys \mid ys \leftarrow inits\ xs]$
$segs\ xs \ = \ [segment\ xs\ i\ j \mid i \leftarrow index\ xs;\ j \leftarrow [i\ldots \#xs]]$

De uitkomsten verschillen in de volgorde waarin de segmenten opgesomd staan, en in het aantal keren dat het lege segment voorkomt. Desgewenst kun je '*inits xs*' herschrijven volgens z'n definitie. Net zo voor *tails*, *segment* en *index*.

Het aantal malen dat *xs* voorkomt als segment in *ys* is:

$xs\ \underline{aantal}\ segs\ ys \qquad \|\ ofwel$
$\#[1 \mid zs \leftarrow segs\ ys;\ xs = zs]$

54. Twee alternatieven; de eerste is het efficiëntst:

$segs'\ n\ xs \ = \ [take\ n\ (drop\ i\ xs) \mid i \leftarrow [0\ldots \#xs{-}n]\ ]$
$segs'\ n\ xs \ = \ [ys \mid ys \leftarrow segs\ xs;\ \#ys = n]$

Het aantal keren dat *xs* als segment voorkomt in *ys*:

$xs\ \underline{aantal}\ segs'\ (\#xs)\ ys$

55. $reverse\ xs \ = \ [xs!(\#xs{-}1{-}i) \mid i \leftarrow index\ xs]$
$reverse\ xs \ = \ [xs!i \mid i \leftarrow [\#xs{-}1,\ \#xs{-}2\ \ldots\ 0]]$
$tl\ xs \ = \ reverse\ (init\ (reverse\ xs))$
$tails\ xs \ = \ reverse\ [reverse\ ys \mid ys \leftarrow inits\ (reverse\ xs)\ ]$

Als in de definitie van *tails* de meest linker '*reverse*' ontbreekt, staan er wel de juiste staartstukken in de resultaatlijst, maar niet in de gewenste volgorde (zie de specificatie van *tails* in opgave 51).

56. Twee alternatieven voor *or*; de eerste is het efficiëntst:

$$\begin{array}{l} or\ xs\ =\ [\texttt{"}iets\texttt{"} \mid x \leftarrow xs;\ x]\ \neq\ [] \\ or\ xs\ =\ \#\,[\texttt{"}iets\texttt{"} \mid x \leftarrow xs;\ x]\ >\ 0 \\ member\ xs\ x\ =\ or\ [\,x' = x \mid x' \leftarrow xs] \end{array}$$

57. Vier alternatieven voor *and*; de eerste twee zijn het efficiëntst:

$$\begin{array}{l} and\ xs\ =\ \neg\ or\ [\neg x \mid x \leftarrow xs] \\ and\ xs\ =\ [\texttt{"}iets\texttt{"} \mid x \leftarrow xs;\ \neg x]\ =\ [] \\ and\ xs\ =\ \#\,[\texttt{"}iets\texttt{"} \mid x \leftarrow xs;\ \neg x]\ =\ 0 \\ and\ xs\ =\ [x \mid x \leftarrow xs;\ x]\ =\ xs \\ allPos\ xs\ =\ and\ [x > 0 \mid x \leftarrow xs] \end{array}$$

58.
$$\begin{array}{l} zip\ (xs, ys) \\ \ =\ [(xs!i,\ ys!i) \mid i \leftarrow [0\ ..\ k{-}1]] \\ \quad\ where \\ \quad\ k\ =\ (\#xs)\ \underline{min2}\ (\#ys) \end{array}$$

59.
$$unzip\ xys\ =\ ([x \mid (x,y) \leftarrow xys],\ [y \mid (x,y) \leftarrow xys])$$

60. Mét en zónder indicering:

$$\begin{array}{l} isStijgend\ xs\ =\ and\ [xs!i\ \leq\ xs!(i{+}1) \mid i \leftarrow init\ (index\ xs)] \\ isStijgend\ xs\ =\ and\ [fst\ z\ \leq\ snd\ z \mid z \leftarrow zip\ (xs,\ tl\ xs)] \end{array}$$

De laatste kan nog iets duidelijker worden opgeschreven:

$$isStijgend\ xs\ =\ and\ [x \leq y \mid (x,y) \leftarrow zip\ (xs,\ tl\ xs)]$$

61. Twee alternatieven, de tweede heeft onze voorkeur:

$$\begin{array}{l} isPalindroom\ xs\ =\ and\ [xs!i\ =\ xs!(\#xs{-}1{-}i) \mid i \leftarrow index\ xs] \\ isPalindroom\ xs\ =\ xs\ =\ reverse\ xs \end{array}$$

Er hoeven geen haakjes om het rechterlid. Wanneer er ook leestekens in het argument staan:

$$isPalindroom'\ \ =\ isPalindroom\ ([kLetter\ x \mid x \leftarrow xs;\ isLetter\ x])$$

De functies *kLetter* en *isLetter* zijn in §3.4 besproken.

62. Met indicering:

$$f\ xs\ =\ \#[\texttt{"}iets\texttt{"} \mid i \leftarrow init\ (index\ xs);\ xs!i\ <\ xs!(i{+}1)] + \#[\texttt{"}iets\texttt{"} \mid \#xs = 1]$$

De eerste term in het rechterlid telt het aantal "sprongen" in xs. Deze term levert 0 op als $\#xs = 1$; vandaar de laatste term: die levert 1 op als $\#xs = 1$, en anders 0. We hadden ook gevalsonderscheid kunnen gebruiken. Zonder gebruik van de indiceringsoperatie !:

$$f\ xs\ =\ \#[\texttt{"}iets\texttt{"} \mid (x,y) \leftarrow zip\ (xs,\ tl\ xs);\ x < y] + \#[\texttt{"}iets\texttt{"} \mid \#xs = 1]$$

Een methode om g te definiëren; tel elke x in xs "die verderop niet meer voorkomt":

$$g\ xs\ =\ \#[\texttt{"}iets\texttt{"} \mid ys \leftarrow tails\ xs;\ ys \neq [];\ \neg member\ (tl\ ys)\ (hd\ ys)]$$

De conditie $ys \neq []$ is nodig, omdat $tl\ []$ en $hd\ []$ niet bestaan. We hadden ook $ys \leftarrow init\ (tails\ xs)$ kunnen nemen; dan is de conditie $ys \neq []$ sowieso vervuld.

Nog een methode; neem $\#xs$ verminderd met 1 voor elke x in xs “die verderop nog eens voorkomt”.

$g\ xs\ =\ \#xs - \#[\texttt{"}iets\texttt{"} \mid ys \leftarrow tails\ xs;\ ys \neq [\,];\ member\ (tl\ ys)\ (hd\ ys)]$

63. Er zijn steeds verscheidene alternatieven:

$verti\ xs\ =\ lay\ [[x] \mid x{\leftarrow}xs]$
$\|\ =\ concat\ [[x,\ '\backslash n'] \mid x{\leftarrow}xs]$
$\|\ =\ [y \mid x{\leftarrow}xs;\ y \leftarrow [x,\ '\backslash n']]$
$sqHor\ xs\ =\ lay\ [xs \mid x{\leftarrow}xs]$
$\|\ =\ concat\ [xs \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"} \mid x{\leftarrow}xs]$
$\|\ =\ [y \mid x{\leftarrow}xs;\ y \leftarrow xs \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"}]$
$sqVer\ xs\ =\ lay\ [\ [x|y{\leftarrow}xs] \mid x{\leftarrow}xs]$
$\|\ =\ lay\ [rep\ (\#xs)\ x \mid x \leftarrow xs]$
$\|\ =\ lay\ [f\ x \mid x \leftarrow xs]\quad where\ f\ x\ =\ [x \mid y{\leftarrow}xs]$
$\|\ =\ concat\ [[x|\ y{\leftarrow}xs] \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"} \mid x{\leftarrow}xs]$
$\|\ =\ [z \mid x{\leftarrow}xs;\ z \leftarrow [x \mid i \leftarrow index\ xs] \mathbin{+\!\!+} \texttt{"}\backslash n\texttt{"}]$
$diaNW\ xs\ =\ lay\ [\ spaces\ i \mathbin{+\!\!+} [xs!i] \mid i \leftarrow index\ xs]$
$\|\ =\ lay\ [take\ (i{+}1)\ (spaces\ i \mathbin{+\!\!+} drop\ i\ xs) \mid i \leftarrow index\ xs]$
$\|\ =\ lay\ [rjustify\ (i{+}1)\ [xs!i] \mid i \leftarrow index\ xs]$
$diaNO\ xs\ =\ lay\ [spaces\ (\#xs{-}1{-}i) \mathbin{+\!\!+} [xs!i] \mid i \leftarrow index\ xs]$
$\|\ =\ lay\ [take\ (\#xs{-}i)\ (spaces\ (\#xs{-}1{-}i) \mathbin{+\!\!+} drop\ i\ xs) \mid i \leftarrow index\ xs]$
$\|\ =\ lay\ [rjustify\ (\#xs{-}i)\ [xs!i] \mid i \leftarrow index\ xs]$
$triaNW\ xs\ =\ lay\ [take\ i\ xs \mid i \leftarrow [\#xs,\ \#xs{-}1\ ..\ 0]]$
$triaNO\ xs\ =\ lay\ [spaces\ i \mathbin{+\!\!+} drop\ i\ xs \mid i \leftarrow [0\,..\,\#xs]]$
$triaZW\ xs\ =\ lay\ [take\ i\ xs \mid i \leftarrow [0\,..\,\#xs]]$
$triaZO\ xs\ =\ lay\ [spaces\ j \mathbin{+\!\!+} drop\ j\ xs \mid j \leftarrow [\#xs,\ \#xs{-}1\ ..\ 0]]$

64. $tafel\ n\ =\ lay\ ([kop\ n] \mathbin{+\!\!+} [regel\ n\ i \mid i{\leftarrow}[1\,..\,10]])$
$kop\ n\ =\ \texttt{"}Vermenigvuldigingstafel\ voor\ \texttt{"} \mathbin{+\!\!+} show'\ n \mathbin{+\!\!+} \texttt{":"}$
$regel\ n\ i\ =\ show'\ n \mathbin{+\!\!+} \texttt{"}\times\texttt{"} \mathbin{+\!\!+} show'\ i \mathbin{+\!\!+} \texttt{" = "} \mathbin{+\!\!+} show'\ (n{\times}i)$
$show'\ x\ =\ rjustify\ 2\ (shownum\ x)$

65. Zónder de regelnummers:

$histogram\ xs\ =\ lay\ [show'\ x \mathbin{+\!\!+} \texttt{" | "} \mathbin{+\!\!+} balk\ x \mid x \leftarrow xs]$
$show'\ y\ =\ rjustify\ 3\ (shownum\ y)$
$balk\ z\ =\ ['\#' \mid i \leftarrow [1\,..\,z]]$

Je kunt desgewenst ook bij de definitie van $show'$ en $balk$ variabele x als parameter gebruiken (in plaats van de y en de z). Desgewenst kan je de hulpfuncties $show'$ en $balk$ wegwerken:

$histogram\ xs$
$=\ lay\ [rjustify\ 3\ (shownum\ x) \mathbin{+\!\!+} \texttt{" | "} \mathbin{+\!\!+} ['\#' \mid i \leftarrow [1\,..\,x]] \mid x \leftarrow xs]$

Mét regelnummers:

$histogram\ xs$
$=\ lay\ [show'\ i \mathbin{+\!\!+} \texttt{" : "} \mathbin{+\!\!+} show'\ x \mathbin{+\!\!+} \texttt{" | "} \mathbin{+\!\!+} balk\ x \mid (i,x) \leftarrow zip\ ([0..],\ xs)]$

Met behulp van de standaard functie *rep* (van: repetition) kunnen we *balk* schrijven als:

$balk\ x\ =\ rep\ x\ '{-}'$

66. $showUitslagen\ xs$
$= lay\ ([streep]\ +\!\!+\ [regel\ x \mid x \leftarrow xs]\ +\!\!+\ [streep])$
$where$
$naamlengte\ =\ 10$ $\|$ *langste naamlengte*
$vl\ =\ $ "| " $\|$ v : *verticaal streepje plus spaties*
$vc\ =\ $ " | "
$vr\ =\ $ " |"
$h\ =\ $ " − " $\|$ h : *horizontaal streepje plus spaties*
$regel\ ((x,y),m,n)$
$=\ vl\ +\!\!+\ show''\ x\ +\!\!+\ h\ +\!\!+\ show''\ y\ +\!\!+\ vc\ +\!\!+\ show'\ m\ +\!\!+\ h\ +\!\!+\ show'\ n\ +\!\!+\ vr$
$show'\ a\ =\ rjustify\ 2\ (shownum\ a)$
$show''\ a\ =\ ljustify\ naamlengte\ a$
$tabelbreedte$
$=\ \#vl + naamlengte + \#h + naamlengte + \#vc + 2 + \#h + 2 + \#vr$
$streep\ =\ [\text{'}-\text{'} \mid i \leftarrow [1\,..\,tabelbreedte]]$

Met behulp van de standaard functie *rep* (van: repetition) kunnen we *streep* schrijven als:
$streep\ =\ rep\ tabelbreedte\ \text{'}-\text{'}$

67. $showUitslagen\ datum\ xs$
$=\ lay\ ([kop,\ streep]\ +\!\!+\ [regel\ x \mid x \leftarrow xs]\ +\!\!+\ [streep])$
$where$
...
$kop\ =\ $ "*Uitslagen van* " $+\!\!+\ datum\ +\!\!+$ " : "
...

68. $chain\ sep\ xss\ =\ drop\ (\#sep)\ (concat\ [sep +\!\!+ xs \mid xs \leftarrow xss])$
$train\ sep\ xss\ =\ concat\ [xs +\!\!+ sep \mid xs \leftarrow xss]$

Alternatieven voor *chain*:
$chain\ sep\ xss\ =\ drop\ (\#sep)\ (train\ sep\ xss)$
$chain\ sep\ xss\ =\ concat\ (take\ 1\ xss\ +\!\!+\ [sep +\!\!+ xs \mid xs \leftarrow drop\ 1\ xss])$
$chain\ sep\ xss$
$=\ [],\ if\ xss = []$
$=\ concat\ ([hd\ xss]\ +\!\!+\ [sep +\!\!+ xs \mid xs \leftarrow tl\ xss]),\ otherwise$

Als in de laatste definitie de regel ‘ $= [],\ if\ xss = []$’ wordt weggelaten, dan resulteert er een foutstop bij *chain sep* []. Dat is niet het geval bij de voorlaatste definitie. Nog een alternatief:
$chain\ sep\ xss$
$=\ [],\ if\ xss = []$
$=\ concat\ (\ [train\ sep\ (init\ xss)]\ +\!\!+\ [last\ xss]),\ if\ xss \neq []$

69. $zin\ xs\ =\ chain$ " " xs
$zinnen\ zss\ =\ train$ ".\n" $[zin\ xs \mid xs \leftarrow zss]$
$zinnen'\ zss\ =\ train$ ". " $[chain$ "; " $[chain$ " " $zs \mid zs \leftarrow zss]]$

De aanroep *zinnen xs* leidt tot een foutstop: het argument *xs* is een woordenlijst, terwijl *zinnen* een lijst van woordenlijsten verwacht.

70. $allen\ =\ mannen +\!\!\!+\ vrouwen$
$afdruk$
$=\ lay\ [x\ +\!\!\!+\ \texttt{"}\backslash n\ \ \texttt{"}\ +\!\!\!+\ concat\ [ljustify\ 20\ k\ |\ k \leftarrow kinderen\ x]\ |\ x \leftarrow allen]$
$afdruk'$
$=\ lay\ [x\ +\!\!\!+\ \texttt{"}\backslash n\texttt{"}\ +\!\!\!+\ lay\ [rjustify\ 23\ k\ |\ k \leftarrow kinderen\ x]\ |\ x \leftarrow allen]$

71. De belangrijkste hulpfunctie is deze:
$weken\ (x, y)\ =\ group\ 7\ (take\ 42\ (take\ x\ nullen\ +\!\!\!+\ [1\mathinner{..}y]\ +\!\!\!+\ nullen))$
$nullen\ =\ [0,\ 0\mathinner{..}]$

Bijvoorbeeld, *weken* $(6, 31)$ levert:

```
[[ 0, 0, 0, 0, 0, 0, 1],
 [ 2, 3, 4, 5, 6, 7, 8],
 [ 9,10,11,12,13,14,15],
 [16,17,18,19,20,21,22],
 [23,24,25,26,27,28,29],
 [30,31, 0, 0, 0, 0, 0]]
```

Nu luidt de definitie van *showMaand* als volgt:
$showMaand\ mnd$
$=\ lay\ ([\texttt{"}\ ma\ di\ wo\ do\ vr\ za\ zo\texttt{"}]\ +\!\!\!+\ [regel\ wk\ |\ wk \leftarrow weken\ mnd])$
$regel\ wk\ =\ concat\ [show'\ d\ |\ d \leftarrow wk]$
$show'\ d$
$=\ rjustify\ 3\ \texttt{""},\ if\ \ d = 0$
$=\ rjustify\ 3\ (shownum\ d),\ \ if\ \ d > 0$

De lijst [" *ma di wo do vr za zo*"] hadden we ook uit *week* kunnen vormen:
$[concat\ [rjustify\ 3\ d\ |\ d \leftarrow week]]$

72. $pi\ ::\ num$
$sqrt\ pi\ ::\ num$
$take\ ::\ num \rightarrow [\alpha] \rightarrow [\alpha]$
$member\ ::\ [\alpha] \rightarrow \alpha \rightarrow bool$
$oplLijst\ ::\ [(num, num, num)]$
$opln\ ::\ num \rightarrow (num, num, num) \rightarrow [(num, num, num)]$
$puzzels\ ::\ num \rightarrow [(num, (num, num, num))]$

73. We definiëren een type-synoniem voor de duidelijkheid:
$naam\ \equiv\ [char]$

Je kunt desgewenst hieronder overal *naam* door [*char*] vervangen.
$mannen\ ::\ [naam]$
$kinderen,\ vaders,\ broersEnZussen\ ::\ naam \rightarrow [naam]$
$isBroerOfZusVan\ ::\ naam \rightarrow naam \rightarrow bool$

74. $show\ ::\ \alpha \rightarrow [char]$

$shownum\ ::\ num \rightarrow [char]$
$lay,\ layn\ ::\ [[char]] \rightarrow [char]$
$error\ ::\ [char] \rightarrow \alpha$

75. $student\ \equiv\ ([char],\ num,\ [char],\ num)\ \|\ (naam,\ geb\,jaar,\ faculteit,\ SP)$
$bestand\ \equiv\ [student]$
$studenten\ ::\ bestand \rightarrow [[char]]$
$gemSP\ ::\ bestand \rightarrow num$
$faculteiten\ ::\ bestand \rightarrow [[char]]$

76. $patient\ \equiv\ ([char],\ bool,\ num,\ [([char],\ num)])$
$\|\ (naam,\ sekse,\ geb\,jaar,\ [(afdeling,\ jaar\ van\ opname)])$
$\|\ sekse\ =\ m/v\ =\ False/True$
$bestand\ \equiv\ [patient]$
$namen,\ namenVr\ ::\ bestand \rightarrow [[char]]$
$opnamesVoor\ ::\ bestand \rightarrow num \rightarrow [([char],\ num)]$

77. $(\#)\ ::\ [\alpha] \rightarrow num$
$take,\ drop\ ::\ num \rightarrow [\alpha] \rightarrow [\alpha]$
$(!)\ ::\ [\alpha] \rightarrow num \rightarrow \alpha$
$or,\ and\ ::\ [bool] \rightarrow bool$
$min,\ max\ ::\ [\alpha] \rightarrow \alpha$
$member\ ::\ [\alpha] \rightarrow \alpha \rightarrow bool$
$concat\ ::\ [[\alpha]] \rightarrow [\alpha]$
$hd,\ last\ ::\ [\alpha] \rightarrow \alpha$
$init,\ tl,\ reverse\ ::\ [\alpha] \rightarrow [\alpha]$
$inits,\ tails,\ segs\ ::\ [\alpha] \rightarrow [[\alpha]]$
$segment\ ::\ num \rightarrow num \rightarrow [\alpha] \rightarrow [\alpha]$
$segs'\ ::\ num \rightarrow [\alpha] \rightarrow [[\alpha]]$
$zip\ ::\ ([\alpha],\ [\beta]) \rightarrow [(\alpha,\ \beta)]$

78. $partitie\ van\ een\ getallijst\ ::\ [[num]]$
$partitie\ van\ een\ karakterlijst\ ::\ [[char]]$
$partitie\ van\ een\ lijst\ van\ getallijsten\ ::\ [[[num]]]$
$partitions\ ::\ [\alpha] \rightarrow [[[\alpha]]]$

Als $xs :: [t]$, dan is $[layn\ xss \mid xss \leftarrow partitions\ xs]$ typeerbaar indien $t\ \equiv\ char$; het type is dan: $[[t]]$ (of te wel $[[char]]$).
De uitdrukking $layn\ (partitions\ xss)$ is niet typeerbaar: $layn$ verwacht een argument van type $[[char]]$, terwijl $partitions\ xss$ iets van type $[[[\alpha]]]$ oplevert. Voor geen enkele keuze van α is $[[char]]$ gelijk aan $[[[\alpha]]]$. Miranda klaagt bij de uitdrukking $layn\ (partitions\ xss)$:

```
type error in expression
cannot unify [[[*]]] with [[char]]
```

79. $letterNa\ 'Z'\ =\ 'A'$
$letterNa\ 'z'\ =\ 'a'$
$letterNa\ x\ =\ decode\ (code\ x + 1)$

80. $f\ (n{+}101)\ =\ n{+}91$
$f\ n\ =\ 91$

81. $hd\ (x : xs)\ =\ x$
$tl\ (x : xs)\ =\ xs$

82. $True\ \underline{xor}\ False\ =\ True$
$False\ \underline{xor}\ True\ =\ True$
$x\ \underline{xor}\ y\ =\ False$

Het laatste alternatief wordt alleen gekozen wanneer de argumenten niet in de patronen passen van de voorgaande alternatieven, dus wanneer x en y beide *True* zijn of beide *False*. Overigens, een veel kortere definitie van *xor* is:

$x\ \underline{xor}\ y\ =\ (x \neq y)$

83. Eerste manier:

$product\ []\ =\ 1$
$product\ [x]\ =\ x$
$product\ (xs + \!\!\! + ys)\ =\ product\ xs \times product\ ys$

De laatste clausule is in Miranda niet geoorloofd als definitie vanwege de $+\!\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$product\ zs$
$\quad =\ product\ xs \times product\ ys$
$\qquad where$
$\qquad (xs, ys)\ =\ (take\ n\ zs,\ drop\ n\ zs)$
$\qquad n\ =\ \#zs\ div\ 2$

Tweede manier:

$product\ []\ =\ 1$
$product\ (xs +\!\!\!+ [x])\ =\ product\ xs \times x$

De laatste clausule in in Miranda niet geoorloofd als definitie vanwege de $+\!\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$product\ zs\ =\ product\ xs \times x\ \ where\ (xs,\ x)\ =\ (init\ zs,\ last\ zs)$

Derde manier:

$product\ []\ =\ 1$
$product\ (x : xs)\ =\ x \times product\ xs$

Met behulp van *product* kunnen we *fac* als volgt definiëren:

$fac\ n\ =\ product\ [1\ ..\ n]$

84. Eerste manier:

$max\ [x]\ =\ x$
$max\ (xs +\!\!\!+ ys)\ =\ max\ xs\ \ \underline{max2}\ \ max\ ys,\ if\ xs \neq []\ \wedge\ ys \neq []$

De laatste clausule is in Miranda niet geoorloofd als definitie vanwege de $+\!\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$max\ zs$

$= max\ xs\ \underline{max2}\ max\ ys,\ if\ zs \neq []$
$\quad where$
$\quad (xs,\ ys)\ =\ (take\ n\ zs,\ drop\ n\ zs)$
$\quad n\ =\ \#zs\ div\ 2$

Er geldt dat $max\ []$ in een foutstop resulteert (want $[]$ past niet in het patroon $[x]$, en past wel in het patroon zs maar daar is de bewaking $zs \neq []$ niet vervuld). Zonder de bewaking '$zs \neq []$' resulteert $max\ []$ in een niet-eindigende berekening: het enige wat je met $max\ []$ kunt doen, volgens de definitie, is te herschrijven tot $max\ []\ \underline{max2}\ max\ []$, en dan staan er al twéé aanroepen van $max\ []$.

Tweede manier:

$max\ [x]\ =\ x$
$max\ (xs +\!\!+ [x])\ =\ max\ xs\ \underline{max2}\ x$

De laatste clausule is in Miranda niet geoorloofd als definitie vanwege de $+\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$max\ zs\ =\ max\ xs\ \underline{max2}\ x\quad where\ (xs,\ x)\ =\ (init\ zs,\ last\ zs)$

De uitdrukking $max\ []$ resulteert in een foutstop, omdat $init\ []$ en $last\ []$ beide in een foutstop resulteren.

Derde manier:

$max\ [x]\ =\ x$
$max\ (x : xs)\ =\ x\ \underline{max2}\ max\ xs$

Nu resulteert $max\ []$ in een foutstop omdat $[]$ niet past in de patronen van de linkerleden.

85. $init\ [x]\ =\ []$
$init\ (x : xs)\ =\ x :\ init\ xs$
$last\ [x]\ =\ x$
$last\ (x : xs)\ =\ last\ xs$

86. $(x : xs)\ !\ 0\ =\ x$
$(x : xs)\ !\ (n{+}1)\ =\ xs\ !\ n$

De uitdrukking $xs\ !\ n$ met $n \geq \#xs$ resulteert in een foutstop omdat het uitrekenen daarvan uiteindelijk leidt tot het uitrekenen van $[]\ !...$, en daarvoor is geen definitie gegeven.

87. Eerste manier:

$reverse\ []\ =\ []$
$reverse\ [x]\ =\ [x]$
$reverse\ (xs +\!\!+ ys)\ =\ reverse\ ys\ +\!\!+\ reverse\ xs$

De laatste clausule is in Miranda niet geoorloofd als definitie vanwege de $+\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$reverse\ zs$
$=\ reverse\ ys\ +\!\!+\ reverse\ xs$
$\quad where$
$\quad (xs,\ ys)\ =\ (take\ n\ zs,\ drop\ n\ zs)$
$\quad n\ =\ \#zs\ div\ 2$

Tweede manier:

$reverse\ []\ =\ []$
$reverse\ (xs +\!\!+ [x])\ =\ [x]\ +\!\!+\ reverse\ xs$

De laatste clausule is in Miranda niet geoorloofd als definitie vanwege de $+\!\!+$ in het linkerlid; een toegestane Miranda formulering ervan luidt:

$reverse\ zs\ =\ [x]\ +\!\!+\ reverse\ xs\ \ \ where\ (xs,\ x)\ =\ (init\ zs,\ last\ zs)$

De regel hierboven mag ook als volgt luiden:

$reverse\ zs\ =\ x : reverse\ xs\ \ \ where\ \ldots$

Derde manier:

$reverse\ []\ =\ []$
$reverse\ (x : xs)\ =\ reverse\ xs\ +\!\!+\ [x]$

88. Met recursie (inductie naar de prefix opbouw van het argument):

$member\ []\ x\ =\ False$
$member\ (x' : xs)\ x\ =\ x' = x\ \vee\ member\ xs\ x$

Bij het uitrekenen van $\ldots \vee \ldots$, rekent Miranda het tweede argument ($member\ xs\ x$) niet uit als het eerste argument ($x' = x$) de uitkomst al bepaalt. De tweede clausule kan ook met gevalsonderscheid geformuleerd worden (maar dat is minder mooi):

$member\ []\ x\ =\ False$
$member\ (x' : xs)$
$=\ True,\ if\ x' = x$
$=\ member\ xs\ x,\ otherwise$

Inductie naar de postfix of $+\!\!+$ opbouw is ook mogelijk.

89. De eerste manier, voor elk van de vier functies. We gebruiken $+\!\!+$ in het linkerlid; dat is in Miranda niet toegestaan, en moet dus nog herschreven worden volgens het schema:

$\ldots\ zs\ =\ \ldots xs \ldots ys \ldots\ \ \ where\ (xs, ys)\ =\ (take\ n\ zs,\ drop\ n\ zs);\ n\ =\ \#zs\ div\ 2$

Let in het bijzonder op de clausules van *and* en *or* bij argument $[]$ en $[x]$:

$and\ []\ =\ True$
$and\ [x]\ =\ x$
$and\ (xs +\!\!+ ys)\ =\ and\ xs\ \wedge\ and\ ys \qquad \| \leftarrow nog\ herschrijven$
$or\ []\ =\ False$
$or\ [x]\ =\ x$
$or\ (xs +\!\!+ ys)\ =\ or\ xs\ \vee\ or\ ys \qquad \| \leftarrow nog\ herschrijven$
$concat\ []\ =\ []$
$concat\ [xs]\ =\ xs$
$concat\ (xss +\!\!+ yss)\ =\ concat\ xss\ +\!\!+\ concat\ yss \qquad \| \leftarrow nog\ herschrijven$
$\#\ []\ =\ 0$
$\#\ [x]\ =\ 1$
$\#\ (xs +\!\!+ ys)\ =\ \#xs\ +\ \#ys \qquad \| \leftarrow nog\ herschrijven$

De tweede manier:

$and\ []\ =\ True$
$and\ (xs +\!\!+ [x])\ =\ and\ xs\ \wedge\ x \qquad \| \leftarrow nog\ herschrijven$
$or\ []\ =\ False$
$or\ (xs +\!\!+ [x])\ =\ or\ xs\ \vee\ x \qquad \| \leftarrow nog\ herschrijven$
$concat\ []\ =\ []$
$concat\ (xss +\!\!+ [xs])\ =\ concat\ xss\ +\!\!+\ xs \qquad \| \leftarrow nog\ herschrijven$
$\#\ []\ =\ 0$

$\#\,(xs +\!\!+ [x]) \;=\; \#xs \;+\; 1 \qquad\qquad \| \leftarrow \textit{nog herschrijven}$

De derde manier:

$and\ [\,] \;=\; True$
$and\ (x : xs) \;=\; x \;\wedge\; and\ xs$
$or\ [\,]\ False$
$or\ (x : xs) \;=\; x \;\vee\; or\ xs$
$concat\ [\,] \;=\; [\,]$
$concat\ (xs : xss) \;=\; xs \;+\!\!+\; concat\ xss$
$\#\,[\,] \;=\; 0$
$\#\,(x : xs) \;=\; 1 + \#xs$

Tenslotte functie f. Eén definitie ervoor luidt:

$f\ [\,] \;=\; 1$
$f\ (x : xs) \;=\; x\ \hat{}\ f\ xs$

Definities met inductie naar de postfix of $+\!\!+$-opbouw lukken niet (of nauwelijks), omdat operatie ˆ niet associatief is: 5ˆ3ˆ2 = 5ˆ(3ˆ2) = 1953125, maar (5ˆ3)ˆ2 = 15625.

90. Inductie naar de postfix opbouw van het argument is het eenvoudigst:

$inits\ [\,] \;=\; [[\,]]$
$inits\ (xs +\!\!+ [x]) \;=\; inits\ xs \;+\!\!+\; [xs +\!\!+ [x]] \qquad \| \leftarrow \textit{nog herschrijven}$

Het kan ook met inductie naar de prefix opbouw:

$inits\ [\,] \;=\; [[\,]]$
$inits\ (x : xs) \;=\; [[\,]] \;+\!\!+\; [x : us \mid us \leftarrow inits\ xs]$

en zelfs met inductie naar de $+\!\!+$ opbouw van het argument:

$inits\ [\,] \;=\; [[\,]]$
$inits\ [x] \;=\; [[\,],\ [x]]$
$inits\ (xs +\!\!+ ys) \;=\; inits\ xs \;+\!\!+\; [xs +\!\!+ us \mid us \leftarrow inits\ ys] \qquad \| \leftarrow \textit{nog herschrijven}$

Voor *tails* luidt de definitie met inductie naar de prefix opbouw:

$tails\ [\,] \;=\; [[\,]]$
$tails\ (x : xs) \;=\; [x : xs] \;+\!\!+\; tails\ xs \;\; \| \;=\; (x : xs) \;:\; tails\ xs$

Inductie naar de postfix opbouw:

$tails\ [\,] \;=\; [[\,]]$
$tails\ (xs +\!\!+ [x]) \;=\; [us +\!\!+ [x] \mid us \leftarrow tails\ xs] \;+\!\!+\; [[\,]]$

Inductie naar de $+\!\!+$-opbouw:

$tails\ [\,] \;=\; [[\,]]$
$tails\ [x] \;=\; [[x],\ [\,]]$
$tails\ (xs +\!\!+ ys) \;=\; [us +\!\!+ ys \mid us \leftarrow tails\ xs] \;+\!\!+\; tails\ ys \;\| \; \textit{nog herschrijven}$

Functie *subs* gedefinieerd met inductie naar de prefix opbouw van het argument:

$subs\ [\,] \;=\; [[\,]]$
$subs\ (x : xs) \;=\; [x : ys \mid ys \leftarrow subs\ xs] \;+\!\!+\; subs\ xs$

De inductie naar de postfix opbouw verloopt analoog. Een definitie met inductie naar de $+\!\!+$-opbouw:

$subs\ [\,] \;=\; [[\,]]$
$subs\ [x] \;=\; [[\,], [x]]$
$subs\ (xs +\!\!+ ys) \;=\; [vs +\!\!+ ws \mid vs \leftarrow subs\ xs;\ ws \leftarrow subs\ ys]$

Ga na dat *subs xs* precies 2ˆn elementlijsten bevat, waarbij $n = \#xs$.

91. Functie *segs* kán zonder recursie gedefinieerd worden:

$segs\ xs \ = \ [zs \mid ys \leftarrow inits\ xs;\ zs \leftarrow tails\ ys]$
$segs\ xs \ = \ concat\ [tails\ ys \mid ys \leftarrow inits\ xs]$

Mét inductie naar de prefix opbouw kan het ook:

$segs\ [] \ = \ [[]]$
$segs\ ([x] +\!\!+\ xs) \ = \ [[]] \ +\!\!+ \ [x : us \mid us \leftarrow inits\ xs] \ +\!\!+ \ segs\ xs$

Inductie naar de postfix opbouw gaat analoog. Als [] slechts éénmaal in het resultaat hoeft voor te komen, kan in de laatste twee clausules het deel [[]] (met bijhorende $+\!\!+$) worden weggelaten. Zelfs inductie naar de $+\!\!+$opbouw lukt, zij het enigszins gekunsteld. Hier is zo'n definitie voor *segs*0 die de lege lijst in het geheel niet bevat:

$segs0\ [] \ = \ []$
$segs0\ [x] \ = \ [[x]]$
$segs0\ (xs +\!\!+ ys)$
$= \ segs0\ xs \ +\!\!+ \ segs0\ ys\ +\!\!+$
$\quad [vs +\!\!+ ws \mid vs \leftarrow tails\ (tl\ xs);\ vs \neq [];\ ws \leftarrow inits\ (init\ ys);\ ws \neq []]$

92. $fib\ 0 \ = \ 0$
$fib\ 1 \ = \ 1$
$fib\ (n{+}2) \ = \ fib\ n + fib\ (n{+}1)$

93. $binom\ (n{+}1)\ (k{+}1) \ = \ binom\ n\ k + binom\ n\ (k{+}1)$
$binom\ n\ 0 \ = \ 1$
$binom\ 0\ k \ = \ 0,\ if\ 0 < k$

Met bovenstaande volgorde van de clausules is de bewaking '*if* $0 < k$' overbodig. Maar als de derde clausule vóór de tweede wordt geplaatst, is die bewaking wel nodig. De laatste clausule kan ook geschreven worden als:

$binom\ 0\ (k{+}1) \ = \ 0$

94. We volgen de definitievorm van *binom* uit opgave 93 nauwgezet:

$subs'\ (x : xs)\ (k{+}1) \ = \ [x : ys \mid ys \leftarrow subs\ xs\ k] \ +\!\!+ \ subs'\ xs\ (k{+}1)$
$subs'\ xs\ 0 \ = \ [[]]$
$subs'\ []\ k \ = \ [],\ if\ 0 < k$

Net als bij *binom* geldt dat met bovenstaande volgorde van de clausules de bewaking '*if* $0 < k$' overbodig is. Maar als de derde clausule vóór de tweede wordt geplaatst, is die bewaking wel nodig. De laatste clausule kan ook geschreven worden als:

$subs'\ []\ (k{+}1) \ = \ []$

95. De eigenschap $ggd\ m\ 0 \ = \ ggd\ 0\ m \ = \ m$ wordt niet gebruikt, de andere wel:

$ggd\ m\ n$
$= \ ggd\ (m{-}n)\ m,\ if\ m > n$
$= \ m,\ if\ m = n$
$= \ ggd\ m\ (n{-}m),\ if\ m < n$

96. Eerst een hulpfunctie, die bij een cijfer de waarde ervan geeft:

$digitval\ x \ = \ code\ x - code\ '0',\ if\ '0' \ \leq \ x \ \leq \ '9'$

Hier is een mogelijke definitie van *numval*:

$numval\ [\,] \ = \ 0$
$numval\ (x : xs) \ = \ digitval\ x \times 10\hat{}(\#xs) + numval\ xs$

Een andere manier (inductie naar de postfix opbouw van het argument):

$numval\ [\,] \ = \ 0$
$numval\ (xs +\!\!+ [x]) \ = \ numval\ xs \times 10 + digitval\ x$

De laatste clausule is in Miranda niet geoorloofd; een Miranda formulering ervan luidt:

$numval\ zs \ = \ numval\ xs \times 10 + digitval\ x \ \ where\ (xs,\ x) \ = \ (init\ zs,\ last\ zs)$

97. $binval\ [\,] \ = \ 0$
$binval\ (x : xs) \ = \ x{\times}2\hat{}\#xs + binval\ xs$

Dat was met inductie naar de prefix opbouw; nu met inductie naar de postfix opbouw:

$binval\ [\,] \ = \ 0$
$binval\ zs \ = \ binval\ xs \times 2 \ + \ x \ \ \ where\ (xs,\ x) \ = \ (init\ zs,\ last\ zs)$

En tenslotte, inductie naar de ++ opbouw van het argument:

$binval\ [\,] \ = \ 0$
$binval\ [x] \ = \ x$
$binval\ zs$
$= \ binval\ xs \times 2\hat{}\#ys \ + \ binval\ ys$
$\quad where$
$\quad (xs,\ ys) \ = \ (take\ n\ zs,\ drop\ n\ zs)$
$\quad n \ = \ \#zs\ div\ 2$

98. $cijfers\ n$
$= \ [n],\ if\ n < 10$
$= \ cijfers\ (n\ div\ 10) \ +\!\!+ \ [n\ mod\ 10],\ if\ n \geq 10$
$csom\ n$
$= \ n,\ if\ n < 10$
$= \ csom\ (sum\ (cijfers\ n)),\ if\ n \geq 10$

We kunnen desgewenst het opbouwen en weer afbreken van de lijst *cijfers* vermijden:

$csom\ n$
$= \ n,\ if\ n < 10$
$= \ csom\ (csom\ (ndiv\ 10) + n\ mod\ 10),\ if\ n \geq 10$

99. Met gebruik van de functie *cijfers* uit opgave 98 is het eenvoudig:

$piep\ n \ = \ n\ mod\ 7 \ = \ 0 \ \vee \ member\ (cijfers\ n)\ 7$

Het opbouwen en doorlopen van de cijferlijst van n kan vermeden worden:

$piep\ n \ = \ n\ mod\ 7 \ = \ 0 \ \vee \ piep'\ n$
$piep'\ n$
$= \ False,\ if\ n = 0$
$= \ n\ mod\ 10 \ = \ 7 \ \vee \ piep'\ (n\ div\ 10),\ if\ n > 0$

De definitie van *piep′* kunnen we mooier en korter formuleren:

$piep'\ n \ = \ n \neq 0 \ \wedge \ (n\ mod\ 10 \ = \ 7 \ \vee \ piep'\ (n\ div\ 10))$

Wanneer het linkerlid van $\wedge$ al faalt, wordt het rechterlid niet meer uitgerekend.

100. $f'\ 1\ =\ [1]$
$f'\ n$
$=\ [n]\ +\!\!+\ f'\ (n\ div\ 2),\ if\ n\ mod\ 2\ =\ 0$
$=\ [n]\ +\!\!+\ f'\ (3{\times}n{+}1),\ otherwise$

De berekening van f voor alle positieve getallen:

$[f'\ n\ |\ n{\leftarrow}[1..]]$

Hetzelfde, maar met een mooiere opmaak:

$layn\ [show\ (f'\ n)\ |\ n{\leftarrow}[1..]]$

101. $nakomelingen\ 0\ p\ =\ [p]$
$nakomelingen\ (n{+}1)\ p\ =\ concat\ [nakomelingen\ n\ k\ |\ k \leftarrow kinderen\ p]$
$voorouders\ n\ p\ =\ [p]$
$voorouders\ (n{+}1)\ p\ =\ concat\ [voorouders\ n\ o\ |\ o \leftarrow ouders\ p]$
$ouders\ p\ =\ [o\ |\ o \leftarrow personen;\ kinderen\ o\ \underline{member}\ p]$

Desgewenst kan *concat* weggewerkt worden:

$nakomelingen\ (n{+}1)\ p\ =\ [nk\ |\ k \leftarrow kinderen\ p;\ nk \leftarrow nakomelingen\ n\ k]$
$voorouders\ (n{+}1)\ p\ =\ [vo\ |\ o \leftarrow ouders\ p;\ vo \leftarrow voorouders\ n\ o]$

102. 1) $(0\ =)\ .\ (mod\ 2)\ ::\ num \rightarrow bool$
2) $(0\ =)\ .\ (n\ mod)\ ::\ num \rightarrow bool$
3) $('a'\ \neq)\ ::\ char \rightarrow bool$
4) $(4\ =)\ .\ (\#)\ ::\ [\alpha] \rightarrow bool$
5) $(5\ =)\ .\ hd\ ::\ [num] \rightarrow bool$
6) $(\underline{member}\ 6)\ ::\ [num] \rightarrow bool$

103. $filter\ ::\ (\alpha{\rightarrow}bool) \rightarrow [\alpha] \rightarrow [\alpha]$
$filter\ p\ []\ =\ []$
$filter\ p\ (x : xs)$
$=\ x :\ filter\ p\ xs,\ if\ p\ x$
$=\ filter\ p\ xs,\ if\ \neg\ p\ x$

Dit is inductie naar de prefix opbouw van het argument. Inductie naar de postfix en $+\!\!+$ opbouw is ook eenvoudig. De tweede clausule kan geschreven worden zonder expliciet gevalsonderscheid:

$filter\ p\ (x : xs)\ =\ [x\ |\ p\ x]\ +\!\!+\ filter\ p\ xs$

Nu de gevraagde uitdrukkingen:

1) $filter\ ((0\ =)\ .\ (mod\ 2))\ [0..] \qquad \|\ =\ [0,\ 2\ ..]$
2) $filter\ ((0\ =)\ .\ (n\ mod))\ [1\ ..\ n]$
3) $filter\ ('a'\ \neq)\ xs$
4) $filter\ ((4\ =)\ .\ (\#))\ xss$
5) $filter\ ((5\ =)\ .\ hd))\ yss$
6) $filter\ (\underline{member}\ 6)\ zss$

104. $map\ f\ []\ =\ []$
$map\ f\ (x : xs)\ =\ f\ x :\ map\ f\ xs$

Dit is inductie naar de prefix opbouw van het argument. Inductie naar de postfix en $+\!\!+$ opbouw is ook eenvoudig.

105. $unzip\ xys\ =\ (map\ fst\ xys,\ map\ snd\ xys)$

106. $all\ p\ =\ and\ .\ map\ p$

107. $takewhile\ p\ []\ =\ []$
$takewhile\ p\ (x : xs)$
$=\ x :\ takewhile\ p\ xs,\ if\ p\ x$
$=\ [],\ if\ \neg\ p\ x$

Niet-recursief:
$takewhile\ p\ =\ last\ .\ filter\ (all\ p)\ .\ inits$

108. $dropwhile\ p\ []\ =\ []$
$dropwhile\ p\ (x : xs)$
$=\ dropwhile\ p\ xs,\ if\ p\ x$
$=\ x : xs,\ if\ \neg\ p\ x$

Niet-recursief:
$dropwhile\ p\ xs\ =\ xs$ -- $takewhile\ p\ xs$

109. $mkset\ []\ =\ []$
$mkset\ (x : xs)\ =\ x :\ mkset\ (filter\ (x \neq)\ xs)$

De tweede clausule kan ook anders (minder efficiënt) opgeschreven worden:
$mkset\ (x : xs)\ =\ x :\ filter\ (x \neq)\ (mkset\ xs)$

110. $lay\ []\ =\ []$
$lay\ (xs : xss)\ =\ xs\ +\!\!+\ "\backslash n"\ +\!\!+\ lay\ xss$

Alternatieve definities:
$lay\ xss\ =\ concat\ [xs +\!\!+ "\backslash n"\ |\ xs \leftarrow xss]$
$lay\ =\ concat\ .\ map\ (+\!\!+"\backslash n")$

111. $[\ (-b\ \underline{pm}\ sqrt\ (b\hat{}2 - 4{\times}a{\times}c))\ /\ (2{\times}a)\ \ |\ \ pm \leftarrow [(+),\ (-)]\ ]$

112. ops
$=\ [(o1, o2, o3, o4)\ |$
$o1, o2, o3, o4 \leftarrow [+, -, \times, /];$
$((((2\ \underline{o1}\ 4)\ \underline{o2}\ 5)\ \underline{o3}\ 2)\ \underline{o4}\ 9)\ =\ 0]$

In Miranda worden functies en operaties afgedrukt als `<function>`. Daarom passen we een truc toe:

$showO\ o$
$=\ "+",\ if\ r\ =\ 6{+}2$
$=\ "-",\ if\ r\ =\ 6{-}2$
$=\ "\times",\ if\ r\ =\ 6{\times}2$
$=\ "/",\ if\ r\ =\ 6/2$

$\quad\quad \mathit{where}\ r\ =\ 6\ \underline{o}\ 2$
$\mathit{uitvoer}$
$=\ \mathit{layn}$
$\quad [\texttt{"("} \mathbin{+\!\!+} \mathit{showO}\ o1 \mathbin{+\!\!+} \texttt{","} \mathbin{+\!\!+} \mathit{showO}\ o2 \mathbin{+\!\!+} \texttt{","} \mathbin{+\!\!+} \mathit{showO}\ o3 \mathbin{+\!\!+} \texttt{","} \mathbin{+\!\!+} \mathit{showO}\ o4 \mathbin{+\!\!+} \texttt{")"}\ |$
$\quad (o1, o2, o3, o4) \leftarrow \mathit{ops}]$

Er zijn drie oplossingen.

Een andere manier is:

ops
$=\ [(o1, o2, o3, o4)\ |$
$\quad o1, o2, o3, o4 \leftarrow [\texttt{"+"}, \texttt{"-"}, \texttt{"}\times\texttt{"}, \texttt{"/"}];$
$\quad (\mathit{doe}\ o4\ (\mathit{doe}\ o3\ (\mathit{doe}\ o2\ (\mathit{doe}\ o1\ 2\ 4)\ 5)\ 2)\ 9)\ =\ 0]$
$\mathit{doe}\ \texttt{"+"}\ =\ (+)$
$\mathit{doe}\ \texttt{"-"}\ =\ (-)$
$\mathit{doe}\ \texttt{"}\times\texttt{"}\ =\ (\times)$
$\mathit{doe}\ \texttt{"/"}\ =\ (/)$

113. $\mathit{vrnl}\ =\ \mathit{numval}\ .\ \mathit{reverse}\ .\ \mathit{shownum}$

114. $\mathit{tel}\ \mathit{nm}\ =\ \mathit{map}\ \mathit{snd}\ (\mathit{filter}\ ((\mathit{nm}\ =)\ .\ \mathit{fst})\ \mathit{telboek})$
$\mathit{abb}\ \mathit{nr}\ =\ \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ ((\mathit{nr}\ =)\ .\ \mathit{snd})\ \mathit{telboek})$
$\mathit{tels}\ =\ \mathit{map}\ \mathit{snd}\ \mathit{telboek}$
$\mathit{abbs}\ =\ \mathit{map}\ \mathit{fst}\ \mathit{telboek}$

115. Functie *lines* blijkt niet nodig te zijn. Zónder gebruik van *lines* kan het als volgt:
$\mathit{format}, \mathit{formatpara}\ ::\ \mathit{num}{\rightarrow}[\mathit{char}]{\rightarrow}[\mathit{char}]$
$\mathit{format}\ n\ =\ \mathit{concat}\ .\ \mathit{map}\ (\mathit{formatpara}\ n)\ .\ \mathit{paras}$
$\mathit{formatpara}\ n\ =\ \mathit{para}\ .\ \mathit{map}\ (\mathit{line}\ n)\ .\ \mathit{wordgrps}\ n\ .\ \mathit{words}$

Mét gebruik van *lines*:
$\mathit{format}, \mathit{formatpara}\ ::\ \mathit{num}{\rightarrow}[\mathit{char}]{\rightarrow}[\mathit{char}]$
$\mathit{format}\ n\ =\ \mathit{concat}\ .\ \mathit{map}\ (\mathit{formatpara}\ n)\ .\ \mathit{paras}$
$\mathit{formatpara}\ n\ =\ \mathit{para}\ .\ \mathit{map}\ (\mathit{line}\ n)\ .\ \mathit{wordgrps}\ n\ .\ \mathit{concat}\ .\ \mathit{map}\ \mathit{words}\ .\ \mathit{lines}$

116. $\mathit{nakomelingen}\ 0\ p\ =\ [p]$
$\mathit{nakomelingen}\ (n{+}1)\ p\ =\ \mathit{concat}\ (\mathit{map}\ (\mathit{nakomelingen}\ n)\ (\mathit{kinderen}\ p))$

De tweede clausule kan iets mooier met functiecompositie geformuleerd worden:
$\mathit{nakomelingen}\ (n{+}1)\ =\ \mathit{concat}\ .\ \mathit{map}\ (\mathit{nakomelingen}\ n)\ .\ \mathit{kinderen}$

Net zo voor *voorouders*:
$\mathit{voorouders}\ n\ p\ =\ [p]$
$\mathit{voorouders}\ (n{+}1)\ =\ \mathit{concat}\ .\ \mathit{map}\ (\mathit{voorouders}\ n)\ .\ \mathit{ouders}$
$\mathit{ouders}\ p\ =\ \mathit{filter}\ ((\underline{\mathit{member}}\ p)\ .\ \mathit{kinderen})\ \mathit{personen}$

117. $\mathit{lijst}\ =\ (\mathit{concat}\ .\ \mathit{map}\ f)\ [0..]\quad \mathit{where}\ f\ n\ =\ \mathit{zip}\ ([0, 1\ ..\ n],\ [n, n{-}1\ ..\ 0])$

118. $f\ n\ =\ \mathit{hd}\ .\ (\mathbin{+\!\!+}[0])\ .\ \mathit{map}\ g\ .\ \mathit{filter}\ (p\ n)$

119. $diagonaal\ [\,]\ =\ [\,]$
$diagonaal\ (xs : xss)\ =\ hd\ xs\ :\ diagonaal\ (map\ tl\ xss)$

120. Met inductie naar de prefix opbouw van het argument:
$transpose\ [xs]\ =\ map\ f\ xs \quad where\ f\ x\ =\ [x]$
$transpose\ (xs : xss)\ =\ zipwith\ (:)\ (xs,\ transpose\ xss)$

121. $scalprod\ a\ =\ map\ (a\times) \qquad \|\ ofwel:\ scalprod\ a\ xs\ =\ map\ (a\times)\ xs$
$xs\ \underline{inprod}\ ys\ =\ sum\ (zipwith\ (\times)\ (xs, ys))$

Een recht-toe recht-aan definitie van *matprod* is niet moeilijk. In het resultaat *matprod xss yss* is het element op rij i en kolom j het inproduct van rij i van xss met kolom j van yss:

$matprod\ xss\ yss$
$=\ [[(xss\ \underline{rij}\ i)\ \underline{inprod}\ (yss\ \underline{kolom}\ j) \mid j \leftarrow [0\,..\,n{-}1]] \mid i \leftarrow [0\,..\,k{-}1]]$
$\quad where$
$\quad k\ =\ \#xss$
$\quad n\ =\ \#(hd\ yss)$
$xss\ \underline{rij}\ i\ =\ xss\ !\ i$
$yss\ \underline{kolom}\ j\ =\ map\ (!j)\ yss$

In Miranda is dit niet efficiënt (en niet elegant) vanwege de vele indiceringen (operatie '!'). Een efficiënte (en elegante) definitie construeren we als volgt.

De achtereenvolgende rijen van *xss* kunnen we gemakkelijk bereiken met *map ... xss*. Om gemakkelijk de kolommen van *yss* te bereiken, met een *map*, gebruiken we functie *transpose* van opgave 120. In onderstaande definitie van *matprod* produceert een aanroep '*mkrij xs*' de rij van de resultaatmatrix die dezelfde index heeft als *xs* in de argumentmatrix *xss*:

$xss\ \underline{matprod}\ yss$
$=\ map\ mkrij\ xss$
$\quad where$
$\quad mkrij\ xs\ =\ map\ (xs\ \underline{inprod})\ (transpose\ yss)$

122. Het schrappen van rij i, of kolom j, van een matrix xss, luidt:
$schrapR\ i\ xss\ =\ schrap\ i\ xss$
$schrapK\ j\ xss\ =\ map\ (schrap\ j)\ xss$
$schrap\ i\ xs\ =\ take\ i\ xs\ +\!\!+\ drop\ (i{+}1)\ xs$

Het schrappen van rij 0 of kolom 0 kan eenvoudiger worden uitgedrukt:
$schrapR\ 0\ (xs : xss)\ =\ xss$
$schrapK\ 0\ xss\ =\ map\ tl\ xss$

Uit de definitie volgt dat een lege matrix precies één greep toelaat: een leeg stel elementen (met grootte $m = 0$). Moeten de voorste elementen (van de grepen uit een niet-lege matrix) alle uit één rij komen, dan definiëren we:

$grepen\ [\,]\ =\ [\ [\,]\ ]$
$grepen\ (xs : xss)$
$\|\ \ [xs!i\ :\ g \mid i \leftarrow index\ xs;\ g \leftarrow grepen\ (schrapR\ 0\ (schrapK\ i\ (xs : xss)))]$
$=\ [xs!i\ :\ g \mid i \leftarrow index\ xs;\ g \leftarrow grepen\ (map\ (schrap\ i)\ xss)]$

Moeten de voorste elementen alle uit één kolom komen, dan:
$grepen\ [\,]\ =\ [[\,]]$

$\quad grepen\ xss$
$\quad = [hd\ (xss!i)\ :\ g\ \mid i \leftarrow index\ xss;\ g \leftarrow grepen\ (schrapR\ i\ (schrapK\ 0\ xss))]$

123. Definieer:
$\quad grepental\ xss\ =\ \#\ grepen\ xss$
Dan geldt voor het lege argument:
$\quad grepental\ []\ =\ \#\ grepen\ []\ =\ \#\ [[]]\ =\ 1$
En voor een matrix $xs : xss$ ter grootte $m{+}1$:
$\quad grepental\ (xs : xss)$
$\quad = \#\ grepen\ (xs : xss)$
$\quad = \#\ [xs!i\ :\ g \mid i \leftarrow index\ xs;\ g \leftarrow grepen\ (map\ (schrap\ i)\ xss)]$
$\quad = \#\ index\ xs\ \times\ \#\ grepen\ (map\ (schrap\ i)\ xss)$
$\quad = (m{+}1)\ \times\ grepental\ xss'$
waarbij xss' een matrix ter grootte m is. Geven we alleen de grootte van de matrix aan de functie mee, dan krijgen we:
$\quad grepental'\ 0\ =\ 1$
$\quad grepental'\ (m{+}1)\ =\ (m{+}1) \times grepental'\ m$
Een ouwe bekende: de faculteitsfunctie. De efficiëntie-winst van $grepental'$ ten opzichte van $grepental$ is vooral gelegen in het feit dat de opbouw van hulplijsten, en het doorlopen daarvan, nu achterwege blijft.

124. $\quad gauss'\ []\ =\ [1]$
$\quad gauss'\ (xs : xss)\ =\ xs\ \underline{bij}\ (gauss'\ (map\ (xs\ \underline{elim})\ xss))$
$\quad gauss\ xss\ =\ init\ (gauss'\ xss)$
In plaats van de eerste clausule kun je desgewenst ook definiëren:
$\quad gauss'\ [xs]\ =\ xs\ \underline{bij}\ [1]$

125. Hulpfunctie $sort'$ her-arrangeert een rij van rijen zó dat degene met het grootste eerste element (in absolute waarde) helemaal achteraan komt te staan.
$\quad sort'\ []\ =\ []$
$\quad sort'\ [xs]\ =\ [xs]$
$\quad sort'\ (xs : ys : zss)$
$\quad = xs :\ sort'\ (ys :\ zss),\ if\ abs\ (hd\ xs)\ \leq\ abs\ (hd\ ys)$
$\quad = ys :\ sort'\ (xs :\ zss),\ if\ abs\ (hd\ ys)\ \leq\ abs\ (hd\ xs)$
$\quad pivot\ xss\ =\ last\ yss\ :\ init\ yss\ \ where\ yss\ =\ sort'\ xss$

126. $\quad f'\ n\ []\ =\ []$
$\quad f'\ i\ (x : xs)\ =\ spaces\ i\ +\!\!+\ show\ x\ +\!\!+\ \texttt{"}\backslash n\texttt{"}\ +\!\!+\ f'\ (i{+}1)\ xs$
$\quad f\ =\ f'\ 0$

127. $\quad unzip\ []\ =\ ([],\ [])$
$\quad unzip\ ((x, y) :\ xys)\ =\ (x : xs,\ y : ys)\ \ where\ (xs,\ ys)\ =\ unzip\ xys$

128. Hulpfunctie $isGoed'$ heeft de extra parameter; $isGoed'\ n\ xs$ geeft aan of `"(((...("` $+\!\!+\ xs$ goedgevormd is (met n openingshaakjes):

$isGoed\ xs\ =\ isGoed'\ 0\ xs$
$isGoed'\ n\ []\ =\ n = 0$
$isGoed'\ n\ ('(' : xs)\ =\ isGoed'\ (n{+}1)\ xs$
$isGoed'\ n\ (')' : xs)\ =\ n > 0\ \wedge\ isGoed'\ (n{-}1)\ xs$

Een minder mooie formulering van $isGoed'$ luidt als volgt:

$isGoed'\ 0\ []\ =\ True$
$isGoed'\ (n{+}1)\ []\ =\ False$
$isGoed'\ n\ (x : xs)$
$=\ isGoed'\ (n{+}1)\ xs,\ if\ x\ =\ '('$
$=\ isGoed'\ (n{-}1)\ xs,\ if\ x\ =\ ')'\ \wedge\ n > 0$
$=\ False,\ if\ x\ =\ ')'\ \wedge\ 0 = n$

129. $isGoed\ xs\ =\ isGoed'\ \texttt{""}\ xs$
$isGoed'\ ys\ []\ =\ ys = []$
$isGoed'\ ys\ ('(' : xs)\ =\ isGoed'\ (')' : ys)\ xs$
$isGoed'\ ys\ ('[' : xs)\ =\ isGoed'\ (']' : ys)\ xs$
$isGoed'\ ys\ ('\{' : xs)\ =\ isGoed'\ ('\}' : ys)\ xs$
$isGoed'\ ys\ (')' : xs)\ =\ ys \neq []\ \wedge\ hd\ ys\ =\ ')'\ \wedge\ isGoed'\ (tl\ ys)\ xs$
$isGoed'\ ys\ (']' : xs)\ =\ ys \neq []\ \wedge\ hd\ ys\ =\ ']'\ \wedge\ isGoed'\ (tl\ ys)\ xs$
$isGoed'\ ys\ ('\}' : xs)\ =\ ys \neq []\ \wedge\ hd\ ys\ =\ '\}'\ \wedge\ isGoed'\ (tl\ ys)\ xs$

Ietwat korter en makkelijker te veralgemenen:

$isGoed'\ ys\ []\ =\ ys = []$
$isGoed'\ ys\ (x : xs)$
$=\ isGoed'\ (sluit\ x :\ ys)\ xs,\ if\ isOpen\ x$
$=\ ys \neq []\ \wedge\ hd\ ys\ =\ x\ \wedge\ isGoed'\ (tl\ ys)\ xs,\ if\ isSluit\ x$
$isOpen\ =\ member\ \texttt{"([\{"}$
$isSluit\ =\ member\ \texttt{")]\}"}$
$sluit\ x\ =\ hd\ [z \mid [y, z] \leftarrow [\texttt{"()"},\ \texttt{"[]"},\ \texttt{"\{\}"}];\ y = x]$

130. Allereerst de test of iets een regelscheider (line separator) is:

$isLsep\ x\ =\ \texttt{"}\backslash n\texttt{"}\ \underline{member}\ x$

Hierdoor is het eenvoudig om ook form feeds, enzovoorts, als regelscheiders te nemen.

$lines\ \texttt{""}\ =\ []$
$lines\ (x : xs)$
$=\ \texttt{""} :\ lines\ xs,\ if\ isLsep\ x$
$=\ (x : ys) : yss,\ otherwise$
$\quad where\ (ys : yss)\ =\ lines\ xs$

Met deze definitie resulteert *lines xs* in een foutstop wanneer *xs* niet op een newline karakter eindigt. Met een extra clausule kunnen we *lines xs* toch voor zulke *xs* definiëren:

$lines\ \texttt{""}\ =\ []$
$lines\ (x : xs)$
$=\ \texttt{""} :\ lines\ xs,\ if\ isLsep\ x$
$=\ (x : \texttt{""}) :\ [],\ if\ xs\ =\ \texttt{""}$
$=\ (x : ys) :\ yss,\ otherwise$
$\quad where\ (ys : yss)\ =\ lines\ xs$

De definitie met behulp van *takewhile* etc:

$lines$ `""` $=$ $[]$
$lines\ xs$
$=\ takewhile\ ((\neg).isLsep)\ xs\ :\ lines\ (drop\ 1\ (dropwhile\ ((\neg).isLsep)\ xs))$

Nu is $lines\ xs\ =\ lines\ (xs \mathbin{+\!\!+}$ `"\n"`$)$ als xs niet op een newline karakter eindigt.
De uitkomst van $lay\ xss$ eindigt voor niet-lege xss altijd met een newline-karakter (en $lay\ [] =$ `""`). Dus met alle definities hierboven geldt $lines\ (lay\ xss) = xss$. De gelijkheid $lay\ (lines\ xs)$ $= xs$ geldt alleen wanneer xs op een newline-karakter eindigt.

131. Met inductie naar de prefix opbouw (en met een hulpfunctie $isWsep$):

$isWsep\ x\ =$ `" \n"` $\underline{member}\ x$
$words$ `""` $=\ []$
$words\ (x : xs)$
$=\ words\ ys,\ if\ isWsep\ x$
$=\ x\ \underline{cons1}\ words\ xs,\ if\ \neg isWsep\ x$
$x\ \underline{cons1}\ (ys : yss)\ =\ (x : ys) :\ yss$
$x\ \underline{cons1}\ []\ =\ (x :$ `""`$) :\ []$

Dus, zoals gevraagd, geen enkel element in *words xs* is leeg.
Met behulp van *takewhile* etc:

$words\ =\ words'\ .\ dropwhile\ isWsep$
$words'$ `""` $=\ []$
$words'\ xs\ =\ takewhile\ ((\neg).isWsep)\ xs\ :\ words\ (dropwhile\ ((\neg).isWsep)\ xs)$

Analoog aan *lines*:

$words\ =\ filter\ (\neq$ `""`$)\ .\ words''$

waarbij $words''$ exact zo gedefinieerd wordt als *lines* (met inductie, of met *takewhile*) behalve dat *isLsep* overal vervangen wordt door *isWsep*.

132.
$f\ =\ map\ words\ .\ lines$
$g\ =\ concat\ .\ f$
$\|\ =\ concat\ .\ map\ words\ .\ lines$
$h\ =\ (\#)\ .\ g$
$\|\ =\ (\#)\ .\ concat\ .\ map\ words\ .\ lines$
$\|\ =\ sum\ .\ map\ (\#)\ .\ map\ words\ .\ lines$
$fmt\ =\ lay\ .\ map\ (drop\ 1\ .\ concat\ .\ map\ ([\text{' '}]\mathbin{+\!\!+})\ .\ words)\ .\ lines$

133.
$f\ xs\ =\ [(xs!i)\times i \mid i \leftarrow index\ xs]$
$f\ xs\ =\ zipwith\ (\times)\ (xs,\ [0..])$

134. Functie fib' combineert de twee aanroepen $fib\ n$ en $fib\ (n{+}1)$ van het rechterlid van functie fib; preciezer gezegd, $fib'\ n\ =\ (fib\ n,\ fib\ (n{+}1))$:

$fib'\ 0\ =\ (0,\ 1)$
$fib'\ (n{+}1)\ =\ (b,\ a{+}b)\ \ where\ (a, b)\ =\ fib'\ n$
$fib\ =\ fst\ .\ fib'$

Functie fib'' is zó dat $fib''\ (a, b)\ i$ de gevraagde uitkomst $fib\ n$ levert, mits a, b de twee "eerst benodigde" Fibonacci-getallen $fib\ i$ en $fib\ (i{+}1)$ zijn.

$fib\ n$
$=\ fib''\ (0, 1)\ 0$

$\quad where$
$\quad \mathit{fib}''\ (a,b)\ i\ =\ a,\ \mathit{if}\ i=n;\quad = \mathit{fib}''\ (b, a{+}b)\ (i{+}1),\ \mathit{otherwise}$

Merk op dat parameter n van *fib* nu bekend is in de definitie van fib''. Dat komt doordat de definitie van fib'' locaal is in de definitie van *fib*.

We kunnen fib'' ook globaal maken. Daartoe laten we parameter j de rol spelen van $n{-}i$:

$\mathit{fib}\ n\ =\ \mathit{fib}''\ (0,1)\ n$
$\mathit{fib}''\ (a,b)\ 0\ =\ a$
$\mathit{fib}''\ (a,b)\ (j{+}1)\ =\ \mathit{fib}''\ (b,\ a{+}b)\ j$

De specificatie van deze fib'', in termen van al de bekende functie *fib*, luidt:

$\mathit{fib}''\ (a,b)\ j\ =\ \mathit{fib}\ n \qquad \mathit{mits}\ \ (a,b)\ =\ (\mathit{fib}\ (n{-}j),\ \mathit{fib}\ (n{+}1{-}j))$

135. $\mathit{plateaus}\ []\ =\ []$
$\mathit{plateaus}\ (x:xs)\ =\ \mathit{takewhile}\ (x =)\ (x:xs)\ :\ \mathit{plateaus}\ (\mathit{dropwhile}\ (=x)\ xs)$

136. Eén manier is om *max* en *map* (#) in de berekening van *plateaus* te verwerken:

$\mathit{llp}\ []\ =\ 0$
$\mathit{llp}\ (x:xs)\ =\ (\#\mathit{takewhile}\ (=x)\ (x:xs))\ \ \underline{\mathit{max2}}\ \ \mathit{llp}\ (\mathit{dropwhile}\ (=x)\ xs)$

Een andere manier; met parameteraccumulatie. Parameter m geeft de lengte van het langste plateau ys aan "dat al verwerkt is". Preciezer gezegd, als $m\ =\ \mathit{llp}\ ys$ (en $ys \mathbin{+\!\!+} xs$ heeft geen plateau over de grens van ys en xs heen), dan is $\mathit{llp}'\ m\ xs\ =\ \mathit{llp}\ (ys \mathbin{+\!\!+} xs)$:

$\mathit{llp}'\ m\ []\ =\ m$
$\mathit{llp}'\ m\ (x:xs)$
$=\ \mathit{llp}'\ (m\ \ \underline{\mathit{max2}}\ \ n)\ (\mathit{dropwhile}\ (=x)\ xs)$
$\quad \mathit{where}\ \ n\ =\ 1 + \#\mathit{takewhile}(=x)xs$
$\mathit{llp}\ =\ \mathit{llp}'\ 0$

Nu een variatie op voorgaande. Parameter m is zoals hierboven, en hulpparameters n, y "onthouden" de lengte en de hoogte van het laatste plateau in ys voorafgaande aan xs (en dat plateau mag doorlopen in xs):

$\mathit{llp}'\ m\ (n,y)\ []\ =\ m\ \ \underline{\mathit{max2}}\ \ n$
$\mathit{llp}'\ m\ (n,y)\ (x:xs)$
$=\ \mathit{llp}'\ m\ (n{+}1, y)\ xs,\ \mathit{if}\ \ y = x$
$=\ \mathit{llp}'\ (m\ \ \underline{\mathit{max2}}\ \ n)\ (1,x)\ xs,\ \mathit{if}\ \ y \neq x$
$\mathit{llp}\ (x:xs)\ =\ \mathit{llp}'\ 0\ (1,x)\ xs$
$\mathit{llp}\ []\ =\ 0$

Met een kustgreep kunnen we deze laatste twee clausules voor *llp* zelf tot één reduceren:

$\mathit{llp}\ xs\ =\ \mathit{llp}'\ 0\ (n', x')\ xs\ \ \mathit{where}\ (n', x')\ =\ (-999,\ \mathit{hd}\ (xs \mathbin{+\!\!+} [0]) + 1)$

want met deze definitie verschilt x' van het eerste element in xs (als dat bestaat). Het doet er niet toe wat n' is, als-ie maar hoogstens 0 is.

137. $\mathit{max}\ [x]\ =\ x$
$\mathit{max}\ xs\ =\ (\mathit{max}\ .\ \mathit{zipwith}\ \mathit{max2}\ .\ \mathit{split})\ xs$
$\mathit{split}\ (x:y:zs)\ =\ (x:xs,\ y:ys)\ \mathit{where}\ (xs, ys)\ =\ \mathit{split}\ zs$
$\mathit{split}\ (x:[])\ =\ (x:[],\ x:[])$
$\mathit{split}\ []\ =\ ([],\ [])$

De rekenstappen van *zipwith* en *split* kunnen met elkaar verweven worden, waardoor het tussenresultaat (tussen *split* en *zipwith*) niet meer expliciet opgebouwd en weer afgebroken wordt:

$zipwithmax2\ []\ =\ []$
$zipwithmax2\ [x]\ =\ [x]$
$zipwithmax2\ (x:y:zs)\ =\ x\ \underline{max2}\ y\ :\ zipwithmax2\ zs$
$max\ [x]\ =\ x$
$max\ xs\ =\ max\ (zipwithmax2\ xs)$

138. Recursief:

$isKeten\ (xs:ys:zss)\ =\ eenverschil\ xs\ ys\ \ \wedge\ isKeten\ (ys:zss)$
$isKeten\ zss\ =\ True\ \ \|\ er\ geldt:\ \#zss \leq 1$
$eenverschil\ (x:xs)\ (y:ys)\ =\ \ x = y\ \wedge\ eenverschil\ xs\ ys\ \ \ \vee\ \ \ x \neq y\ \wedge\ xs = ys$
$eenverschil\ xs\ ys\ =\ False\ \ \|\ verschillende\ lengte,\ \ of:\ beide\ leeg\ dus\ gelijk$

Met *zip* etc.

$isKeten\ zss\ =\ (and\ .\ zipwith\ eenverschil)\ (zss,\ tl\ zss)$
$eenverschil\ xs\ ys\ =\ \#xs = \#ys\ \ \wedge\ \ 1\ =\ \#filter\ (= False)\ (zipwith\ (=)\ (xs, ys))$

139. Eerst de hulp-functie:

$v\ \ \underline{magVoor}\ \ w$
$=\ or\ [drop\ m\ v\ =\ take\ (\#w{-}n)\ w]\ |$
$\quad m \leftarrow [0\ ..\ maxm];\ n \leftarrow [minn\ m\ \ ..\ maxn];\ abs\ (m{-}n)\ \leq\ 2]$
$\quad where$
$\quad maxm\ =\ max\ [5,\ \#v]$
$\quad maxn\ =\ max\ [5,\ \#w]$
$\quad minn\ m\ =\ 1,\ if\ \#v = m;\ \ =\ 0,\ otherwise$

Wanneer m groter is dan $\#v$ duidt-ie niet meer *precies* het aantal letters aan dat van v wordt geschrapt; analoog voor n. De minimale waarde van n is zódanig dat w niet leeg is.
Nu de recursieve definitie van *isKetting*:

$isKetting\ []\ =\ []$
$isKetting\ [v]\ =\ []$
$isKetting\ (v:w:ws)\ =\ \ v\ \ \underline{magVoor}\ \ w\ \wedge\ isKetting\ (w:ws)$

En de niet-recursieve definitie:

$isKetting\ ws\ =\ and\ (zipwith\ magVoor\ (ws,\ tl\ ws))$

140. $merge\ (x:xs)\ (y:ys)$
$=\ x:\ merge\ xs\ (y:ys),\ if\ x \leq y$
$=\ y:\ merge\ (x:xs)\ ys,\ if\ x \geq y$
$merge\ xs\ ys\ =\ xs\ +\!\!+\ ys\qquad \|\ een\ van\ xs\ of\ ys\ is\ leeg$

141. $merge'\ (x:xs)\ (y:ys)$
$=\ x:\ merge'\ xs\ ys,\ if\ x = y$
$=\ x:\ merge'\ xs\ (y:ys),\ if\ x < y$
$=\ y:\ merge'\ (x:xs)\ ys,\ if\ x > y$
$merge'\ xs\ ys\ =\ xs\ +\!\!+\ ys\qquad \|\ een\ van\ xs\ of\ ys\ is\ leeg$

142. $\mathit{vereniging}\ (x : \mathit{xs})\ (y : \mathit{ys})$
$= x :\ \mathit{vereniging}\ \mathit{xs}\ \mathit{ys},\ \mathit{if}\ x = y$
$= x :\ \mathit{vereniging}\ \mathit{xs}\ (y : \mathit{ys}),\ \mathit{if}\ x < y$
$= y :\ \mathit{vereniging}\ (x : \mathit{xs})\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{vereniging}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs}\ +\!\!+\ \mathit{ys}\ \ \|\ \mathit{een\ van\ beide\ is\ leeg}$
$\mathit{doorsnee}\ (x : \mathit{xs})\ (y : \mathit{ys})$
$= x :\ \mathit{doorsnee}\ \mathit{xs}\ \mathit{ys},\ \mathit{if}\ x = y$
$= \mathit{doorsnee}\ \mathit{xs}\ (y : \mathit{ys}),\ \mathit{if}\ x < y$
$= \mathit{doorsnee}\ (x : \mathit{xs})\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{doorsnee}\ \mathit{xs}\ \mathit{ys}\ =\ []\qquad \|\ \mathit{een\ van\ beide\ is\ leeg}$
$\mathit{symverschil}\ (x : \mathit{xs})\ (y : \mathit{ys})$
$= \mathit{symverschil}\ \mathit{xs}\ \mathit{ys},\ \mathit{if}\ x = y$
$= x :\ \mathit{symverschil}\ \mathit{xs}\ (y : \mathit{ys}),\ \mathit{if}\ x < y$
$= y :\ \mathit{symverschil}\ (x : \mathit{xs})\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{symverschil}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs}\ +\!\!+\ \mathit{ys}\ \ \|\ \mathit{een\ van\ beide\ is\ leeg}$
$\mathit{zonder}\ (x : \mathit{xs})\ (y : \mathit{ys})$
$= \mathit{zonder}\ \mathit{xs}\ \mathit{ys},\ \mathit{if}\ x = y$
$= x :\ \mathit{zonder}\ \mathit{xs}\ (y : \mathit{ys}),\ \mathit{if}\ x < y$
$= \mathit{zonder}\ (x : \mathit{xs})\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{zonder}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs}\ \ \|\ \mathit{een\ van\ beide\ is\ leeg}$
$\mathit{isDeel}\ (x : \mathit{xs})\ (y : \mathit{ys})$
$= \mathit{isDeel}\ \mathit{xs}\ \mathit{ys},\ \mathit{if}\ x = y$
$= \mathit{False},\ \mathit{if}\ x < y$
$= \mathit{isDeel}\ (x : \mathit{xs})\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{isDeel}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs} = []\ \ \|\ \mathit{een\ van\ beide\ is\ leeg}$
$\mathit{isIn}\ x\ (y : \mathit{ys})$
$= \mathit{True},\ \mathit{if}\ x = y$
$= \mathit{False},\ \mathit{if}\ x < y$
$= \mathit{isIn}\ x\ \mathit{ys},\ \mathit{if}\ x > y$
$\mathit{isIn}\ x\ \mathit{ys}\ =\ \mathit{False}\ \ \|\ \mathit{ys\ is\ leeg}$

De definities van *isDeel* en *isIn* kunnen iets korter:

$\mathit{isDeel}\ (x : \mathit{xs})\ (y : \mathit{ys})\ =\ x = y \wedge \mathit{isDeel}\ \mathit{xs}\ \mathit{ys}\quad \vee\quad x > y \wedge \mathit{isDeel}\ (x : \mathit{xs})\ \mathit{ys}$
$\mathit{isDeel}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs} = []$
$\mathit{isIn}\ x\ (y : \mathit{ys})\ =\ x = y\quad \vee\quad x > y \wedge \mathit{isIn}\ x\ \mathit{ys}$
$\mathit{isIn}\ x\ \mathit{ys}\ =\ \mathit{False}$

Natuurlijk zijn sommige functies ook uitdrukbaar in andere. Bijvoorbeeld:

$\mathit{symverschil}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{vereniging}\ \mathit{xs}\ \mathit{ys}\ \underline{\mathit{zonder}}\ \mathit{doorsnee}\ \mathit{xs}\ \mathit{ys}$
$\mathit{isDeel}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{xs}\ \underline{\mathit{zonder}}\ \mathit{ys}\ =\ []$
$\mathit{isIn}\ x\ \mathit{ys}\ =\ \mathit{isDeel}\ [x]\ \mathit{ys}$

143. We gebruiken *fold* als zowel *foldl* als ook *foldr* goed zijn.

$\mathit{sum}\ =\ \mathit{fold}\ (+)\ 0$
$\mathit{product}\ =\ \mathit{fold}\ (\times)\ 1$
$\mathit{and}\ =\ \mathit{fold}\ (\wedge)\ \mathit{True}$
$\mathit{or}\ =\ \mathit{fold}\ (\vee)\ \mathit{False}$

$(\#) \;=\; foldl\ op\ 0\ \ where\ n\ \underline{op}\ x \;=\; n{+}1$
$(\#) \;=\; foldr\ op\ 0\ \ where\ x\ \underline{op}\ n \;=\; 1{+}n$
$concat \;=\; fold\ ({+\!\!+})\ []$

144. Schets van de definitie:
$reverse\ [x,\ldots,y,z] \;=\; (([]\ \underline{prefix}\ x)\ \ldots\ \underline{prefix}\ y)\ \underline{prefix}\ z$
waarbij *prefix* gedefinieerd is door: $xs\ \underline{prefix}\ x \;=\; x : xs$.
Dus met behulp van *foldl*:
$reverse \;=\; foldr\ prefix\ []\quad where\ xs\ \underline{prefix}\ x \;=\; x : xs$
Met *foldr* gaat het ook:
$reverse\ [x,y,\ldots,z] \;=\; x\ \underline{postfix}\ (y\ \underline{postfix}\ \ldots\ (z\ \underline{postfix}\ []))$
$x\ \underline{postfix}\ xs \;=\; xs\ {+\!\!+}\ [x]$
Functie *postfix* is een standaard functie. Dus:
$reverse \;=\; foldr\ postfix\ []$

145. Schematisch:
$rev\ ys\ [x,y,\ldots,z] \;=\; ((ys\ \underline{prefix}\ x)\ \underline{prefix}\ y)\ \ldots\ \underline{prefix}\ z$
Dus:
$rev\ ys \;=\; foldl\ prefix\ ys\quad where\ xs\ \underline{prefix}\ x \;=\; x : xs$
Met *foldr* lukt het niet.

146. $f\ x$
$=\ x,\ if\ ({>}\ 999)\ x$
$=\ f\ ((2\times)\ x),\ otherwise$
$until\ p\ g\ x$
$=\ x,\ if\ p\ x$
$=\ until\ p\ g\ (g\ x),\ otherwise$
$spatiesWeg \;=\; until\ p\ tl\ \ where\ p\ xs \;=\; xs = []\ \vee\ hd\ xs \neq {'}\ {'}$
$dropwhile\ p \;=\; until\ p'\ tl\ \ where\ p'\ xs \;=\; xs = []\ \vee\ p\ (hd\ xs)$
Merk op dat predicaten p en p' op staartstukken van een lijst werken.

147. $sort\ [] \;=\; []$
$sort\ xs$
$=\ sort\ xs'\ {+\!\!+}\ [x']$
$\quad where$
$\quad (xs',\ x') \;=\; (init\ ys,\ last\ ys)$
$\quad ys \;=\; bubble\ xs$
$bubble\ [x] \;=\; [x]$
$bubble\ (x : y : zs) \;=\; \ldots\quad \|\ zonder\ wijziging$
Functie *bubble* wordt niet aangeroepen met een lege argumentlijst; dus *bubble* [] hoeft niet gedefinieerd te worden. Het is toegestaan, maar onnodig, om *sort* $[x]$ apart te definiëren.
Per recursie-slag kost de berekening van *bubble* een vast aantal vergelijkingen (één, of twee), dus de berekening van de uitkomst van *bubble xs* kost een vast aantal keer $\#xs$ vergelijkingen.
In de berekening van *sort xs* wordt *bubble* op steeds kleinere beginstukken van *xs* aangeroepen; dus die berekening kost een vast aantal keer $n{+}(n{-}1){+}(n{-}2){+}\ldots{+}1 \;=\; \frac{1}{2}{\times}n{\times}(n{+}1)$ vergelijkingen, waarbij $n = \#xs$. Het totaal aantal uitgevoerde vergelijkingen neemt evenre-

dig toe met n^2.
Alternatieve definitie van *bubble*:

$bubble\ [x] \ = \ [x]$
$bubble\ (x : y : zs) \ = \ min2\ x\ y \ : \ bubble\ ((max2\ x\ y) : \ zs)$

148. $sort\ [\,] \ = \ [\,]$
$sort\ (x : xs) \ = \ x \ \underline{insert}\ sort\ xs$
$x \ \underline{insert}\ ys \ = \ takewhile\ (< x)\ ys \ +\!\!+ \ [x] \ +\!\!+ \ dropwhile\ (< x)\ ys$

Een recursieve definitie van *in*:

$x \ \underline{insert}\ [\,] \ = \ [x]$
$x \ \underline{insert}\ (y : ys)$
$= \ x : \ y : \ ys, \ if\ x \leq y$
$= \ y : \ x \ \underline{insert}\ ys, \ if\ x \geq y$

De berekening van $x \ \underline{insert}\ ys$ kost, in grootte-orde, $\#ys$ vergelijkingen. Dus de berekening van $sort\ xs$ met $n = \#xs$ kost, in grootte-orde, $n+(n-1)+(n-2)+\ldots+1$ vergelijkingen, dat is, in grootte-orde, n^2.

149. $sort\ [\,] \ = \ [\,]$
$sort\ [x] \ = \ [x]$
$sort\ zs$
$= \ sort\ xs \ \underline{merge}\ sort\ ys$
$where$
$(xs, \ ys) \ = \ (take\ n\ zs, \ drop\ n\ zs)$
$n \ = \ \#zs\ div\ 2$

De berekening van $xs \ \underline{merge}\ ys$ kost, in grootte-orde, $\#xs + \#ys$ vergelijkingen. Dus de berekening van $sort\ xs$ met $n = \#xs$ kost, in grootte-orde, $n+2\times\frac{1}{2}n+4\times\frac{1}{4}n+\ldots$ vergelijkingen. Iedere term in deze sommatie is gelijk aan n. Het aantal termen is het aantal keren dat n gehalveerd kan worden met een uitkomst groter of gelijk aan 1; dat aantal is ${}^2\log n$. Dus het aantal vergelijkingen is, in grootte-orde, $n \log n$.

150. Alleen de clausule voor $sort\ [\,]$ moet nog toegevoegd worden:

$sort\ [\,] \ = \ [\,]$
$sort\ xs \ = \ m \ : \ sort\ (xs -\!\!- [m]) \ where\ m \ = \ min\ xs$

De berekening van $min\ xs$ kost, in grootte-orde, $\#xs$ vergelijkingen. Dus de berekening van $sort\ xs$ met $n = \#xs$ kost, in grootte-orde, $n+(n-1)+(n-2)+\ldots+1$ vergelijkingen, dat is, in grootte-orde, n^2.

151. We schrijven de definitie van *kleintjes* en *groten* ter plekke uit:

$sort\ [\,] \ = \ [\,]$
$sort\ (x : xs) \ = \ sort\ (filter\ (\leq x)\ xs) \ +\!\!+ \ [x] \ +\!\!+ \ sort\ (filter\ (> x)\ xs)$

De berekening van de kleintjes en de groten van xs kost, in grootte-orde, $\#xs$ vergelijkingen. Dus, als er toevallig steeds evenveel kleintjes als groten zijn, dan kost de berekening van $sort\ xs$ met $n = \#xs$, in grootte-orde, $n+2\times\frac{1}{2}n+4\times\frac{1}{4}n+\ldots$ vergelijkingen, dat is, $n \log n$ (zie het antwoord van opgave 149). Maar als er toevallig steeds één kleintje is of één grote, dan kost de berekening van $sort\ xs$ met $n = \#xs$, in grootte-orde, $n+(n-1)+(n-2)+\ldots+1$ vergelijkingen, dat is, in grootte-orde, n^2.

152. $min'\ n\ (x : xs)$
$= \ min'\ n\ kleintjes, \ if\ k > n$

$= x, \; if \; k = n$
$= min' \; (n{-}k{-}1) \; groten, \; if \; k < n$
$where$
$(kleintjes, \; groten) \; = \; (filter \; (\leq x) \; xs, \; filter \; (> x) \; xs)$
$k \; = \; \#kleintjes$

De berekening van de kleintjes en de groten van xs kost, in grootte-orde, $\#xs$ vergelijkingen. Wanneer er toevallig steeds evenveel kleintjes als groten zijn, dan kost de berekening van $min' \; k \; xs$, met $n = \#xs$, in grootte-orde hoogstens $n{+}\frac{1}{2}n{+}\frac{1}{4}n{+} \ldots = 2n$ vergelijkingen, dat is, in grootte-orde, n. Het algoritme $sort \; xs \; ! \; n$ kost, op z'n best, in grootte-orde n^2 stappen. Nu geldt:

$kleinte \; element \; van \; xs \; = \; min' \; 0 \; xs$
$een \; na \; kleinste \; van \; xs \; = \; min' \; 1 \; xs$
$mediaan \; van \; xs \; = \; min' \; (\#xs \; div \; 2) \; xs$

153. $mkset \; [] \; = \; []$
$mkset \; (x : xs) \; = \; mkset \; (filter \; (< x) \; xs) \; +\!\!+ \; [x] \; +\!\!+ \; mkset \; (filter \; (> x) \; xs)$

Alternatief; eerst heel efficiënt sorteren en dan uitdunnen:

$mkset \; = \; dunUit \; . \; sort$
$dunUit \; (x : y : zs) \; = \; [x | \; x < y] \; +\!\!+ \; dunUit \; (y : zs)$
$dunUit \; zs \; = \; zs \qquad \| \; \#zs \; \leq \; 1$

154. De gemakkelijkste oplossing:

$f \; = \; (\#) \; . \; mkset$

Als de standaard functie $mkset$ zó gedefinieerd is dat de berekening van $mkset \; xs$ in grootte-orde $n \log n$ vergelijkingen kost (met $n \; = \; \#xs$), dan kost de berekening van $f \; xs$ ook $n \log n$ vergelijkingen. Zo'n definitie van $mkset$ is in opgave 109 gegeven. We kunnen desgewenst $f \; = \; (\#) \; . \; mkset$ direct definiëren:

$f \; [] \; = \; 0$
$f \; (x : xs) \; = \; f \; (filter \; (< x) \; xs) + 1 + f \; (filter \; (> x) \; xs)$

Een efficiënte definitie voor de functie g:

$g \; xs \; ys \; = \; \# \; doorsnee \; (sort \; xs) \; (sort \; ys)$

waarbij $sort$ = merge sort, en $doorsnee$ zoals in opgave 142.
In opgave 152 is al een efficiënte definitie van $mediaan$ gegeven:

$mediaan \; xs \; = \; min' \; (\#xs \; div \; 2) \; xs$

Ook goed is: $mediaan \; xs = sort \; xs \; ! \; (\#xs \; div \; 2)$ met een '$n \log n$'-definitie voor $sort$, bijvoorbeeld merge sort.

155. Inductie naar de prefix opbouw van het argument:

$transpose \; [xs] \; = \; map \; f \; xs \; \; where \; f \; x \; = \; [x]$
$transpose \; (xs : xss) \; = \; zipwith \; (:) \; (xs, \; transpose \; xss)$

Inductie naar de prefix opbouw van het resultaat:

$transpose \; xss$
$= [], \; if \; hd \; xss \; = \; [] \quad \| \; dat \; wil \; zeggen: \; xss \; = \; [\,[], \; [], \; [], \; \ldots\,]$
$= [concat \; xss], \; if \; \# \; hd \; xss \; = \; 1 \quad \| \; dwz: \; \; xss \; = \; [[x], \; [y], \; [z], \; \ldots]$
$= map \; hd \; xss \; : \; transpose \; (map \; tl \; xss), \; \; otherwise$

De tweede clausule is overbodig; die gelijkheid volgt al uit de eerste en de derde clausule. Als voor ieder argument $xss = [xs,\ ys,\ zs,\ \ldots]$ van *transpose* geldt dat $xs, ys, zs \ldots$ allemaal niet-leeg zijn, dan mag de eerste clausule weggelaten worden (en is de tweede wel nodig). De standaard functie *transpose* is iets algemener: die werkt ook voor een argument $[xs, ys, \ldots, zs]$ met $\#xs \geq \#ys \geq \ldots \geq \#zs$.

156. $in\ 0 = []$
$in\ 1 = [0]$
$in\ (n{+}2) = in\ n \mathbin{+\!\!+} [n{+}1] \mathbin{+\!\!+} uit\ n \mathbin{+\!\!+} in\ (n{+}1)$

Alternatief:

$in\ n = map\ neg\ (uit\ n)$

Hierbij is *neg* de standaard functie met $neg\ x = -x$. Met de definitie analoog aan *uit* kost de berekening van $in\ n$ precies evenveel stappen als die van $uit\ n$; met de laatste definitie komen daar nog de berekeningsstappen voor *map neg* bij. Beide definities van *in* kosten in grootte-orde evenveel stappen: exponentioneel in de lengte van het argument.

157. $uit'\ m\ n = uit\ n \mathbin{+\!\!+} in\ m$

Deze is correct onder aanname dat bij aanroep de eerste n staafjes er uit zijn.

158. Met parameteraccumulatie; we definiëren $aantal'$ zó dat $aantal'\ (a, b)\ i = aantal\ n$ mits $a, b = aantal\ (n{-}i),\ aantal\ (n{+}1{-}i)$:

$aantal'\ (a, b)\ 0 = a$
$aantal'\ (a, b)\ (i{+}1) = aantal'\ (b,\ a{+}1{+}a{+}b)\ i$
$aantal\ n = aantal'\ (0, 1)\ n$

Met extra resultaten; we definiëren $aantal'$ zó dat $aantal'\ n = (aantal\ n,\ aantal\ (n{+}1))$:

$aantal'\ 0 = (0,\ 1)$
$aantal'\ (n{+}1) = (b,\ a{+}1{+}a{+}b)\ \ where\ (a, b) = aantal'\ n$
$aantal = fst\ .\ aantal'$

Nog iets efficiënter; *aantal*0 doet 't voor even getallen, *aantal*1 voor oneven getallen:

$aantal\ n$
$= aantal0\ n\ ,\ if\ n\ mod\ 2 = 0$
$= aantal1\ n,\ otherwise$
$aantal0\ 0 = 0$
$aantal0\ (n{+}1) = 2 \times aantal1\ n$
$aantal1\ \ (n{+}1) = 2 \times aantal0\ n + 1$

159. $hanoi = hanoi'\ 'A'\ 'B'\ 'C'\ 64$
$hanoi'\ x\ y\ z\ 0 = []$
$hanoi'\ x\ y\ z\ (n{+}1) = hanoi'\ x\ z\ y\ n \mathbin{+\!\!+} [(x, y)] \mathbin{+\!\!+} hanoi'\ z\ y\ x\ n$

Merk op dat de verplaatsing (x, y) de grootste (oorspronkelijk onderste) schijf betreft; de beide recursieve aanroepen van $hanoi'$ betreffen de n bovenste, kleinere, schijven. Een aanroep $hanoi'\ x\ y\ z\ n$ levert een rij correcte verplaatsingen op van de bovenste n schijven van plaats x naar plaats y via plaats z, **indien** (bij aanroep) plaatsen x, y en z geoorloofde stapels bevatten én plaatsen y en z geen kleinere schijven bevatten dan de n bovenste schijven van x. Deze voorwaarde is ook bij de recursieve aanroepen in het rechterlid vervuld, wanneer die al voor het linkerlid geldt, en hiermee is in te zien dat de verplaatsing (x, y) in de definitie van $hanoi'$ correct is.

Een nette afdruk wordt verkregen door:

$showhanoi \ = \ concat\ (map\ f\ hanoi)\ \ where\ f\ (x, y)\ =\ [x,\ y,\ '\ ']$

Dit levert: `AC AB CB AC BA BC AC AB CB CA BA CB AC ...`

Het aantal benodigde verplaatsingen:

$hanoital\ 0\ =\ 0$
$hanoital\ (n{+}1)\ =\ hanoital\ n + 1 + hanoital\ n$

Dit kan efficiënter:

$hanoital\ n\ =\ 2\hat{}\,n - 1$

De verplaatsingen voor een toren van 64 schijven kosten heel wat jaren:

$hanoiduur\ n\ =\ (hanoital\ n \times 10\ )\ div\ (60 \times 60 \times 24 \times 365)$

De aanroep *hanoiduur* 64 levert: $5849424173550 = 5.8\ 10^{12}$ (jaar!). Volgens recente schattingen is ons heelal "tenminste $15{\times}10^9$ jaar" oud.

160. $dagnr\ Ma\ =\ 0$
$dagnr\ Di\ =\ 1$
$\ldots\ etc\ \ldots$

Alternatief:

$dagnr\ x\ =\ \#\ takewhile\ (\neq x)\ [Ma, Di, Wo, Do, Vr, Za, Zo]$

161. $dagNa\ Ma\ =\ Di$
$dagNa\ Di\ =\ Wo$
$\ldots\ etc\ \ldots$
$dagNa\ Zo\ =\ Ma$

Alternatief:

$dagNa\ x\ =\ nrdag\ ((1 + dagnr\ x)\ mod\ 7)$
$nrdag\ n\ =\ [Ma, Di, Wo, Do, Vr, Za, Zo]\ !\ n$

162. $F\ g\ \underline{verhoogdMet}\ n\ =\ F\ (g{+}n)$
$C\ g\ \underline{verhoogdMet}\ n\ =\ C\ (g{+}n)$
$t\ \underline{isEvenwarm}\ t'\ =\ kelvin\ t\ =\ kelvin\ t'$
$t\ \underline{isWarmer}\ t'\ =\ kelvin\ t\ >\ kelvin\ t'$
$kelvin\ (C\ g)\ =\ g{+}273$
$kelvin\ (F\ g)\ =\ (g{-}32){\times}5/9{+}273$

163. $bool\ ::=\ False \mid True$

164. $nat\ ::=\ Nul \mid Opv\ nat$

Nu kunnen patronen *Nul* en *Opv n* net zo gebruikt worden als de patronen 0 en $n{+}1$ bij het standaard type *num*.

$plus,\ mult\ ::\ nat \rightarrow nat \rightarrow nat$
$n\ \underline{plus}\ Nul\ =\ n$
$m\ \underline{plus}\ (Opv\ n)\ =\ Opv\ (m\ \underline{plus}\ n)$
$m\ \underline{mult}\ Nul\ =\ Nul$
$m\ \underline{mult}\ (Opv\ n)\ =\ (m\ \underline{mult}\ n)\ \underline{plus}\ m$

Voor type *num* bestaan er de notaties 0, 1, 2, ... Die moeten we nu schrijven als *Nul*, *Opv Nul*, *Opv* (*Opv Nul*), ...

165. $lijst\ \alpha\ ::=\ Leeg \mid \alpha\ \underline{Cons}\ lijst\ \alpha$
$map\ ::\ (\alpha \to \beta) \to (lijst\ \alpha \to lijst\ \beta)$
$map\ f\ Leeg\ =\ Leeg$
$map\ f\ (x\ \underline{Cons}\ xs)\ =\ f\ x\ \underline{Cons}\ map\ f\ xs$
$pplus\ ::\ lijst\ \alpha \to lijst\ \alpha \to lijst\ \alpha$
$Leeg\ \underline{pplus}\ xs\ =\ xs$
$(x\ \underline{Cons}\ xs)\ \underline{pplus}\ ys\ =\ x\ \underline{Cons}\ (xs\ \underline{pplus}\ ys)$
$concat\ ::\ lijst\ (lijst\ \alpha) \to lijst\ \alpha$
$concat\ Leeg\ =\ Leeg$
$concat\ (xs\ \underline{Cons}\ xss)\ =\ xs\ \underline{plusplus}\ concat\ xss$

Voor type $[\alpha]$ bestaan er de notaties [], $[x]$, $[x, y]$, ... Die moeten we nu schrijven als *Leeg*, $x\ \underline{Cons}\ Leeg$, $x\ \underline{Cons}\ y\ \underline{Cons}\ Leeg$, ...; dus net zo als [], $x : []$, $x : y : []$, ...

166. $tupel3\ \alpha\ \beta\ \beta\ ::=\ Tupel3\ \alpha\ \beta\ \gamma$
$fst3\ ::\ tupel3\ \alpha\ \beta\ \gamma \to \alpha$
$fst3\ (Tupel3\ x\ y\ z)\ =\ x$
$snd3\ ::\ tupel3\ \alpha\ \beta\ \gamma \to \beta$
$snd3\ (Tupel3\ x\ y\ z)\ =\ y$

In Miranda worden 2-tupels, 3-tupels, 4-tupels, ... allemaal met de (, , ...)-notatie geschreven. Die moeten we nu met verschillende identifiers *Tupel*2, *Tupel*3, *Tupel*4 schrijven.

167. $met\ ::\ tabel\ \alpha\ \beta \to (\alpha, \beta) \to tabel\ \alpha\ \beta$
$Klein\ t\ \underline{met}\ (x, y)\ =\ Klein\ ((x, y) : filter\ ((x \neq) .fst)\ t)$
$Groot\ t\ \underline{met}\ (x, y)$
$=\ Groot\ t'$
where
$t'\ x'\ =\ t\ x',\ if\ x' \neq x;\quad =\ y,\ if\ x' = x$

168. $gegeven\ ::=\ Naam\ [char] \mid Geboren\ num \mid Sekse\ sekse$
$sekse\ ::=\ M \mid V$
$showsekse\ M\ =\ "man"$
$showsekse\ V\ =\ "vrouw"$

$vrouwental\ ps\ =\ \#[\ p \mid p{\leftarrow}ps;\ member\ p\ (Sekse\ V)]$

Nu een definitie van *geslacht′* die van één persoon het geslacht geeft:

$geslacht'\ ::\ persoon \to [char]$
$geslacht'\ (Sekse\ mv :\ p)\ =\ showsekse\ mv$
$geslacht'\ (gegeven :\ p)\ =\ geslacht'\ p$
$geslacht'\ []\ =\ "onbekend"$

De functie kan ook ook in één regel gedefinieerd worden:

$geslacht'\ p\ =\ hd\ ([showsekse\ mv \mid Sekse\ mv \leftarrow p]\ +\!\!+\ ["onbekend"])$

De generator '*Sekse mv* ← *p*' werkt als een filter: alleen de elementen van *p* die passen in het patroon '*Sekse nv*' worden gegenereerd.
Nu de gevraagde definitie:

$geslacht\ ::\ [persoon]{\to}[char]{\to}[char]$

$geslacht\ []\ nm\ =\ \texttt{"}afwezig\texttt{"}$
$geslacht\ (p : ps)\ nm$
$=\ geslacht'\ p,\ if\ member\ p\ (Naam\ nm)$
$=\ geslacht\ ps\ nm,\ otherwise$

De drie clausules kunnen in één regel geformuleerd worden:

$geslacht\ ps\ nm\ =\ hd\ ([geslacht'\ p \mid p \leftarrow ps;\ member\ p\ (Naam\ nm)]\ +\!\!+\ [\texttt{"}afwezig\texttt{"}])$

169. $hd',\ max'\ ::\ [\alpha] \rightarrow maybe\ \alpha$
$max'\ []\ =\ No$
$max'\ [x]\ =\ Yes\ x$
$max'\ (x : y : zs)\ =\ Yes\ (x\ \underline{max2}\ r)\ where\ Yes\ r\ =\ max'\ (y : zs)$
$div'\ ::\ num \rightarrow num \rightarrow maybe\ num$
$div'\ m\ 0\ =\ No$
$div'\ m\ n\ =\ Yes\ (m\ div\ n)$

Merk op hoe in de laatste regel van max', met behulp van een patroon, een (het) onderdeel van het resultaat van $max'\ (y : zs)$ wordt benoemd.

$cmp\ ::\ (\beta \rightarrow maybe\ \gamma) \rightarrow (\alpha \rightarrow maybe\ \beta) \rightarrow (\alpha \rightarrow maybe\ \gamma)$
$(f'\ \underline{cmp}\ g')\ x$
$=\ h\ (g'\ x)$
$\quad where$
$\quad h\ No\ =\ No$
$\quad h\ (Yes\ y)\ =\ f'\ y$
$f'\ =\ (3\ \underline{div'})\ \underline{cmp}\ max'$

171. $depth\ ::\ tree \rightarrow num$
$depth\ (Tip\ n)\ =\ 0$
$depth\ (Node\ n\ t\ t')\ =\ 1 + max\ [depth\ t,\ depth\ t']$
$depth'\ (Node'\ n\ [])\ =\ 0$
$depth'\ (Node'\ n\ ts)\ =\ 1 + max\ (map\ depth'\ ts)$

172. Een goede naam voor f is *grootte* (of *size*):
$f\ ::\ tree \rightarrow num$
$f\ (Tip\ n)\ =\ 0$
$f\ (Node\ n\ t\ t')\ =\ f\ t + 1 + f\ t'$
$f'\ (Node'\ n\ ts)\ =\ 1 + sum\ (map\ f'\ ts)$

173. Een goede naam voor f is *tipsom*:
$f\ ::\ tree \rightarrow num$
$f\ (Tip\ n)\ =\ n$
$f\ (Node\ n\ t\ t')\ =\ f\ t + f\ t'$
$f'\ (Node'\ n\ [])\ =\ n$
$f'\ (Node'\ n\ ts)\ =\ sum\ (map\ f'\ ts)$

174. Een goede naam voor f is *labelsom* of *boomsom*:
$f\ ::\ tree \rightarrow num$

$f\ (Tip\ n)\ =\ n$
$f\ (Node\ n\ t\ t')\ =\ f\ t + n + f\ t'$
$f'\ (Node'\ n\ [])\ =\ n$
$f'\ (Node'\ n\ ts)\ =\ n + sum\ (map\ f'\ ts)$

175. Een goede naam voor f is *isIn* of *komtvoor*:
$f\ ::\ num \rightarrow tree \rightarrow bool$
$f\ m\ (Tip\ n)\ =\ m = n$
$f\ m\ (Node\ n\ t\ t')\ =\ m = n\ \vee\ f\ m\ t\ \ \vee\ \ f\ m\ t'$
$f'\ m\ (Node'\ n\ ts)\ =\ m = n\ \ \vee\ \ or\ (map\ (f'\ m)\ ts)$
Bedenk dat het rechterlid van $\vee$ en $\wedge$ niet wordt uitgerekend als het linkerlid de uitkomst al bepaalt.
176. Een goede naam voor f is $tips10maal$; de beste naam voor f is gewoon f:
$f\ ::\ tree \rightarrow tree$
$f\ (Tip\ n)\ =\ Tip\ (10{\times}n)$
$f\ (Node\ n\ t\ t')\ =\ Node\ n\ (f\ t)\ (f\ t')$
$f'\ (Node'\ n\ [])\ =\ Node'\ (10{\times}n)\ []$
$f'\ (Node'\ n\ ts)\ =\ Node'\ n\ (map\ f'\ ts)$

177. Een goede naam voor f is *gespiegelde*:
$f\ ::\ tree \rightarrow tree$
$f\ (Tip\ n)\ =\ Tip\ n$
$f\ (Node\ n\ t\ t')\ =\ Node\ n\ (f\ t')\ (f\ t)$
$f'\ (Node'\ n\ [])\ =\ Node'\ n\ []$
$f'\ (Node'\ n\ ts)\ =\ Node'\ n\ (map\ f'\ (reverse\ ts))$

178. Een goede naam voor f is *paden*:
$f\ ::\ tree \rightarrow [[num]]$
$f\ (Tip\ n)\ =\ [[n]]$
$f\ (Node\ n\ t\ t')\ =\ map\ (n\ :)\ (f\ t\ +\!\!+\ f\ t')$
$f'\ (Node'\ n\ [])\ =\ [[n]]$
$f'\ (Node'\ n\ ts)\ =\ map\ (n\ :)\ (concat\ (map\ f'\ ts))$

179. Ik weet geen betere naam voor f dan 'f'.
Een veralgemening $g'\ k$ van f' is gemakkelijk(er) recursief te definiëren:
$g'\ k\ t = (f'\ t\ \ \wedge\ \ t$ heeft minder dan k subbomen).
$f'\ ::\ tree' \rightarrow bool$
$g'\ ::\ num \rightarrow tree' \rightarrow bool$
$g'\ k\ (Node'\ n\ [])\ =\ k > 0$
$g'\ k\ (Node'\ n\ ts)\ =\ k > \#ts\ \wedge\ and\ (map\ (g'\ (\#ts))\ ts)$
$f'\ t'\ =\ g'\ hugenum\ t'$
In plaats van *hugenum* kun je ook nemen: $1 + \#t'$.
180. Een goede naam voor f is *preorder*, en dus *inorder* voor g etcetera:
$f, g, h\ ::\ tree \rightarrow [num]$
$j\ ::\ tree \rightarrow [[num]]$

$f\ (Tip\ n)\ =\ [n]$
$f\ (Node\ n\ t\ t')\ =\ [n]\ +\!\!+\ f\ t\ +\!\!+\ f\ t'$
$g\ (Tip\ n)\ =\ [n]$
$g\ (Node\ n\ t\ t')\ =\ g\ t\ +\!\!+\ [n]\ +\!\!+\ g\ t'$
$h\ (Tip\ n)\ =\ [n]$
$h\ (Node\ n\ t\ t')\ =\ h\ t\ +\!\!+\ h\ t'\ +\!\!+\ [n]$

Een correcte maar niet zo heel efficiënte definitie voor j gebruikt een hulpfunctie jj die de opsomming van alle knopen op precies één diepte oplevert:

$jj\ ::\ tree \rightarrow num \rightarrow [[num]]$
$j\ t\ =\ map\ (jj\ t)\ [0\ ..\ depth\ t]$
$jj\ (Tip\ n)\ 0\ =\ [n]$
$jj\ (Tip\ n)\ (k{+}1)\ =\ []$
$jj\ (Node\ n\ t\ t')\ 0\ =\ [n]$
$jj\ (Node\ n\ t\ t')\ (k{+}1)\ =\ jj\ t\ k\ +\!\!+\ jj\ t'\ k$

In de berekening van j kunnen de extra aanroep van *depth* en de vele afdalingen in de boom als volgt vermeden worden. In lijst $j\ t$ staat de opsomming van diepte i op plaats i, en bij $j\ t'$ is dat net zo. Dus door de elementen van $j\ t$ en $j\ t'$ met plaatsnummer i samen te voegen, krijgen we de opsommingen van $Node\ n\ t\ t'$ van diepte $i{+}1$. Dat samenvoegen doen we met *join*:

$join\ ::\ [[\alpha]] \rightarrow [[\alpha]] \rightarrow [[\alpha]]$
$join\ (xs : xss)\ (ys : yss)\ =\ (xs +\!\!+ ys) : join\ xss\ yss$
$join\ xss\ yss\ =\ xss\ +\!\!+\ yss\ \ \|\ een\ van\ beide\ is\ leeg$

En dus:

$j\ (Tip\ n)\ =\ [[n]]$
$j\ (Node\ n\ t\ t')\ =\ [[n]]\ +\!\!+\ join\ (j\ t)\ (j\ t')$

Nu de accent-versie van f en h; die van g is niet zinvol:

$f'\ (Node'\ n\ ts)\ =\ [n]\ +\!\!+\ concat\ (map\ f'\ ts)$
$h'\ (Node'\ n\ ts)\ =\ concat\ (map\ h'\ ts)\ +\!\!+\ [n]$

Voor de accent-versie van j werken we de methode met *join* uit:

$j'\ (Node'\ n\ [])\ =\ [[n]]$
$j'\ (Node'\ n\ ts)\ =\ [[n]]\ +\!\!+\ joins\ (map\ j'\ ts)$

Voor *joins* moet gelden:

$joins\ [xss, yss, zss, \ldots]\ =\ xss\ \underline{join}\ yss\ \underline{join}\ zss\ \ldots$

Met inductie is *joins* eenvoudig te definiëren; met *foldl* en *foldr* ook:

$joins\ ::\ [[[\alpha]]] \rightarrow [[\alpha]]$
$joins\ =\ foldl\ join\ []$
$joins\ =\ foldr\ join\ []$

181. Een goede naam voor f is *showtree* of *showtreelinear*:

$f, g, h\ ::\ tree \rightarrow [char]$
$f\ (Tip\ n)\ =\ show\ n$
$f\ (Node\ n\ t\ t')\ =\ \texttt{"("}\ +\!\!+\ show\ n\ +\!\!+\ \texttt{" "}\ +\!\!+\ f\ t\ +\!\!+\ \texttt{" "}\ +\!\!+\ f\ t'\ +\!\!+\ \texttt{")"}$
$g\ (Tip\ n)\ =\ show\ n$
$g\ (Node\ n\ t\ t')\ =\ \texttt{"("}\ +\!\!+\ g\ t\ +\!\!+\ \texttt{" "}\ +\!\!+\ show\ n\ +\!\!+\ \texttt{" "}\ +\!\!+\ g\ t'\ +\!\!+\ \texttt{")"}$

$h\ (Tip\ n)\ =\ show\ n$
$h\ (Node\ n\ t\ t')\ =\ \texttt{"("}\ +\!\!+\ h\ t\ +\!\!+\ \texttt{" "}\ +\!\!+\ h\ t'\ +\!\!+\ \texttt{" "}\ +\!\!+\ show\ n\ +\!\!+\ \texttt{")"}$

Voor de accent-versies:

$f'\ (Node'\ n\ [])\ =\ show\ n$
$f'\ (Node'\ n\ ts)\ =\ \texttt{"("}\ +\!\!+\ show\ n\ +\!\!+\ concat\ (map\ ((\texttt{" "}+\!\!+).f')\ ts)\ +\!\!+\ \texttt{")"}$
$h'\ (Node'\ n\ [])\ =\ show\ n$
$h'\ (Node'\ n\ ts)\ =\ \texttt{"("}\ +\!\!+\ concat\ (map\ ((+\!\!+\texttt{" "}).h')\ ts)\ +\!\!+\ show\ n\ +\!\!+\ \texttt{")"}$

182. $shownum'\ n\ =\ rjustify\ 3\ (shownum\ n)$

$showtree\ ::\ tree \rightarrow [char]$
$showtr\ ::\ num \rightarrow tree \rightarrow [char]$
$showtree\ =\ showtr\ 0$
$showtr\ k\ (Tip\ n)\ =\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ \texttt{"}\backslash n\texttt{"}$
$showtr\ k\ (Node\ n\ t\ t')$
$=\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ \texttt{"}\backslash n\texttt{"}\ +\!\!+\ showtr\ (k{+}4)\ t\ +\!\!+\ showtr\ (k{+}4)\ t'$

$showtree'\ =\ showtr'\ 0$
$showtr'\ k\ (Node'\ n\ ts)$
$=\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ \texttt{"}\backslash n\texttt{"}\ +\!\!+\ concat\ (map\ (showtr'\ (k{+}4))\ ts)$

Beide voorkomens van "$\backslash n$" kunnen ook vóór de $spaces\ k\ +\!\!+\ shownum'\ n$ geplaatst worden. Ook zonder *rjustify* komt er een goed plaatje uit; alleen staan dan de eenheden van de labels-van-gelijke-diepte niet recht onder elkaar.

183. Uitgaande van een regelovergang ná iedere *shownum*:

$shownum'\ n\ =\ rjustify\ 3\ (shownum\ n)$

$showtreeC\ =\ showtrC\ 0$
$showtrC\ k\ (Tip\ n)\ =\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ \texttt{"}\backslash n\texttt{"}$
$showtrC\ k\ (Node\ n\ t\ t')$
$=\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+$
$drop\ (1{+}k{+}3)\ (\texttt{"}\backslash n\texttt{"}\ +\!\!+\ showtrC\ (k{+}4)\ t\ +\!\!+\ showtrC\ (k{+}4)\ t')$

Het gedeelte $drop\ (1{+}k{+}3)\ (\texttt{"}\backslash n\texttt{"}\ +\!\!+\ \ldots$ kan vereenvoudigd worden tot $drop\ (k{+}3)\ (\ldots.$
Uitgaande van een regelovergang vóór iedere *shownum*:

$showtrC\ k\ (Tip\ n)\ =\ \texttt{"}\backslash n\texttt{"}\ +\!\!+\ spaces\ k\ +\!\!+\ shownum'\ n$
$showtrC\ k\ (Node\ n\ t\ t')$
$=\ \texttt{"}\backslash n\texttt{"}\ +\!\!+\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+$
$drop\ (1{+}k{+}3)\ (showtrC\ (k{+}4)\ t\ +\!\!+\ showtrC\ (k{+}4)\ t')$

Nu kan de $drop\ (1{+}k{+}3)\ \ldots$ niet vereenvoudigd worden tot $drop\ (k{+}3)\ \ldots.$
Voor de accent-versie, uitgaande van een regelovergang na iedere *shownum*:

$showtreeC'\ =\ showtrC'\ 0$
$showtrC'\ k\ (Node'\ n\ [])\ =\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ \texttt{"}\backslash n\texttt{"}$
$showtrC'\ k\ (Node'\ n\ ts)$
$=\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+$
$drop\ (1{+}k{+}3)\ (\texttt{"}\backslash n\texttt{"}\ +\!\!+\ concat\ (map\ (showtrC'\ (k{+}4))\ ts))$

Let op: als de clausule voor $showtrC'$ $(Node'\ n\ [])$ weggelaten wordt, wordt er nóóit een regelovergang getoond! Een alternatieve definitie voor $showtrC'$ is (zónder clausule voor $Node'\ n\ []$):

$$\begin{aligned}
&showtrC'\ k\ (Node'\ n\ ts)\\
&\ = spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\ drop\ (k{+}3)\ (lay\ (map\ (showtrC'\ (k{+}4))\ ts))
\end{aligned}$$

En nu uitgaande van een regelovergang vóór iedere $shownum$:

$$\begin{aligned}
&showtreeC' = showtrC'\ 0\\
&showtrC'\ k\ (Node'\ n\ ts)\\
&\ = \texttt{"\textbackslash n"}\ +\!\!+\ spaces\ k\ +\!\!+\ shownum'\ n\ +\!\!+\\
&\qquad drop\ (1{+}k{+}3)\ (concat\ (map\ (showtrC'\ (k{+}4))\ ts))
\end{aligned}$$

Er is nu geen aparte clausule voor $Node'\ n\ []$ nodig.

184. Een goede naam voor f is bijvoorbeeld *mktree* of *mktreepreorder*:

$$\begin{aligned}
&f\ [] = Tip\ 0\\
&f\ (x : xs) = Node\ x\ (f\ (take\ n\ xs))\ (f\ (drop\ n\ xs))\ where\ n = \#xs\ div\ 2\\
&g\ zs\\
&\ = Node\ x\ (g\ xs)\ (g\ ys)\\
&\qquad where\\
&\qquad n = (\#zs{-}1)\ div\ 2\\
&\qquad (xs,\ x,\ ys) = (take\ n\ zs,\ zs!n,\ drop\ (n{+}1)\ zs)\\
&h\ [] = Tip\ 0\\
&h\ zs\\
&\ = Node\ x\ (h\ (take\ n\ xs))\ (h\ (drop\ n\ xs))\\
&\qquad where\\
&\qquad (xs,\ x) = (init\ zs,\ last\ zs)\\
&\qquad n = \#xs\ div\ 2
\end{aligned}$$

De mate van evenwichtigheid wordt volkomen bepaald door de keuze van n; wat dat betreft is iedere waarde geoorloofd. Bij g moet n zodanig zijn dat $take\ n\ zs$ niet gelijk is aan zs, want in dat geval stopt de recursie niet. Bij f en h stopt de recursie altijd, voor iedere waarde van n.

185. Een goede naam voor f is ligt niet voor de hand.

$$\begin{aligned}
&f\ (Tip\ n) = True\\
&f\ (Node\ n\ t\ t') = (n = wortel\ t + wortel\ t')\ \wedge\ f\ t\ \wedge\ f\ t'\\
&wortel\ (Tip\ n) = n\\
&wortel\ (Node\ n\ t\ t') = n\\
&f'\ (Node'\ n\ []) = True\\
&f'\ (Node'\ n\ ts) = n = sum\ (map\ wortel'\ ts)\ \wedge\ and\ (map\ f'\ ts)\\
&wortel'\ (Node'\ n\ ts) = n
\end{aligned}$$

186. Een goede naam voor f is *isEvenwichtig*:

$$\begin{aligned}
&f\ (Tip\ n) = True\\
&f\ (Node\ n\ t\ t') = abs\ (depth\ t - depth\ t') \le 1\ \wedge\ f\ t\ \wedge\ f\ t'\\
&f'\ (Node'\ n\ ts)\\
&\ = ts = []\ \vee\ \ max\ ds - min\ ds \le 1\ \wedge\ and\ (map\ f'\ ts)\\
&\qquad where\\
&\qquad ds = map\ depth'\ ts
\end{aligned}$$

187. Een goede naam voor f is misschien *isScheef*:

$f\ (Tip\ n)\ =\ True$
$f\ (Node\ n\ t\ t')\ =\ depth\ t\ \leq\ depth\ t'\ \wedge\ f\ t\ \wedge\ f\ t'$

$f'\ (Node'\ n\ ts)\ =\ ts = []\ \vee\ isStijgend\ (map\ depth'\ ts)\ \wedge\ and\ (map\ f'\ ts)$
$isStijgend\ xs\ =\ and\ (zipwith\ (\leq)\ (xs,\ tl\ xs))$

188. Een goede naam voor f is *isVol*.
Voor de hand ligt de volgende definitie:

$f\ (Tip\ n)\ =\ True$
$f\ (Node\ n\ t\ t')\ =\ depth\ t\ =\ depth\ t'\ \wedge\ f\ t\ \wedge\ f\ t'$

Maar we kunnen de berekening van *depth* met die van f combineren. Daartoe definiëren we *fdepth* zó dat: $fdepth\ t\ =\ (f\ t,\ depth\ t)$. Dan is f zelf gemakkelijk in *fdepth* uit te drukken:

$f\ =\ fst\ .\ fdepth$
$fdepth\ (Tip\ n)\ =\ (True,\ 0)$
$fdepth\ (Node\ n\ t\ t')$
$\ =\ (d = d'\ \wedge\ b\ \wedge\ b',\ \ 1 + max\ [d, d'])$
$\quad where$
$\quad (b, d)\ =\ fdepth\ t$
$\quad (b', d')\ =\ fdepth\ t'$

Een iets andere schrijfwijze voor de tweede clausule van *fdepth*:

$fdepth\ (Node\ n\ t\ t')\ =\ combineer\ (fdepth\ t)\ (fdepth\ t')$
$combineer\ (b, d)\ (b', d')\ =\ (d = d'\ \wedge\ b\ \wedge\ b',\ 1 + max\ [d, d'])$

Voor de accent-versie:

$fdepth'\ (Node'\ n\ [])\ =\ (True,\ 0)$
$fdepth'\ (Node'\ n\ ts)\ =\ combineer'\ (map\ fdepth'\ ts)$
$combineer'\ bds$
$\ =\ (max\ ds\ =\ min\ ds\ \wedge\ and\ bs,\ \ 1 + max\ ds)$
$\quad where$
$\quad (bs,\ ds)\ =\ unzip\ bds\qquad \|\ =\ (map\ fst\ bds,\ map\ snd\ bds)$

189. Een goede naam voor f ligt niet voor de hand. Wat dacht je van *isNietrafelig*?

$f\ (Tip\ n)\ =\ True$
$f\ (Node\ n\ t\ t')\ =\ soort\ t\ =\ soort\ t'\ \wedge\ f\ t\ \wedge\ f\ t'$
$soort\ (Tip\ n)\ =\ 0$
$soort\ (Node\ n\ t\ t')\ =\ 1$

Een andere manier.

$f\ (Tip\ n)\ =\ True$
$f\ (Node\ n\ t\ t')\ =\ equiv\ t\ t'\ \wedge\ f\ t\ \wedge\ f\ t'$
$equiv\ (Tip\ m)\ (Tip\ n)\ =\ True$
$equiv\ (Node\ m\ s\ s')\ (Node\ n\ t\ t')\ =\ True$
$equiv\ s\ t\ =\ False$

Voor de accent-versie:

$f'\ (Node'\ n\ ts)\ =\ \#\ mkset\ (map\ soort'\ ts)\ \leq\ 1\ \wedge\ and\ (map\ f'\ ts)$

$$soort'\ (Node'\ n\ []) \ = \ 0$$
$$soort'\ (Node'\ n\ ts) \ = \ 1$$

De conditie $\#\ mkset\ (map\ soort'\ ts) \ \leq\ 1$ is precies dan waar als alle elementen in ts van dezelfde soort zijn. Herinner je dat *mkset* de standaardfunctie is die dubbele voorkomens uit een lijst verwijdert (en mogelijk de volgorde verandert). Er zijn vele andere manieren om die conditie uit te drukken.

190. Een goede naam voor f is *gelijkeStructuur* of *sameForm* of *eqStructure*:

$$f\ (Tip\ m)\ (Tip\ n) \ = \ True$$
$$f\ (Node\ m\ s\ s')\ (Node\ n\ t\ t') \ = \ f\ s\ t \ \wedge \ f\ s'\ t'$$
$$f\ t\ t' \ = \ False$$

De definitie van f' ziet er eenvoudiger uit:

$$f'\ (Node'\ m\ ss)\ (Node'\ n\ ts) \ = \ \#ss = \#ts \ \wedge \ and\ (zipwith\ f'\ (ss, ts))$$

191. Een goede naam voor f is *isZoekboom*.
We veralgemenen f tot een functie g met: $g\ p\ q\ t$ = 't is een zoekboom met alle waarden tussen p en q'. Dan is g gemakkelijk te definiëren:

$$g\ p\ q\ (Tip\ n) \ = \ p < n < q$$
$$g\ p\ q\ (Node\ n\ t\ t') \ = \ p < n < q \ \wedge \ g\ p\ n\ t \ \wedge \ g\ n\ q\ t'$$

En f is nu een speciaal geval van g:

$$f\ t \ = \ g\ (-hugenum)\ hugenum\ t$$

Een andere manier is om f te veralgemenen tot h met: $h\ t = (f\ t,\ wortel\ t)$. Die h is weer gemakkelijk te definiëren, en daarmee ook f:

$$f \ = \ fst\ .\ h$$
$$h\ (Tip\ n) \ = \ (True,\ n)$$
$$h\ (Node\ n\ t\ t')$$
$$= \ (w < n < w' \ \wedge\ b\ \wedge\ b',\quad n) \quad where\ (b, w) \ = \ h\ t; \quad (b', w') \ = \ h\ t'$$

192. Een goede naam voor f is *isIn* of *komtvoor* (of *isInZoekboom* etc):

$$f\ m\ (Tip\ n) \ = \ m = n$$
$$f\ m\ (Node\ n\ t\ t') \ = \ (m < n\ \wedge\ f\ m\ t) \ \vee \ m = n \ \vee \ (n < m\ \wedge\ f\ m\ t')$$

Bedenk dat Miranda de $\wedge$ en $\vee$ condities van links naar rechts uitrekent, en alleen zover als nodig is om het eindresultaat te bepalen. Dus als in de laatste clausule geldt $n < m$, dan wordt de conditie '$f\ m\ t$' niet uitgerekend (de rest wel).

193. Een goede naam voor f ligt niet voor de hand.
We veralgemenen f tot $g\ k$; die zet in iedere Node en Tip: k + de som van de getallen in het pad vanaf die Node of Tip naar boven.

$$g\ k\ (Tip\ n) \ = \ Tip\ (n{+}k)$$
$$g\ k\ (Node\ n\ t\ t') \ = \ Node\ k'\ (g\ k'\ t)\ (g\ k'\ t') \quad where\ k' \ = \ n{+}k$$
$$f \ = \ g\ 0$$

De accent-versie:

$$f' \ = \ g'\ 0$$
$$g'\ k\ (Node'\ n\ ts) \ = \ Node'\ k'\ (map\ (g'\ k')\ ts) \quad where\ k' \ = \ n{+}k$$

194. Een goede naam voor f ligt niet voor de hand.
Eerst met een hulpfunctie g die de som van alle getallen in een boom oplevert:

$f\ (Tip\ n)\ =\ Tip\ n$
$f\ (Node\ n\ t\ t')\ =\ Node\ (n + g\ t + g\ t')\ (f\ t)\ (f\ t')$
$g\ (Tip\ n)\ =\ n$
$g\ (Node\ n\ t\ t')\ =\ n + g\ t + g\ t'$

De berekeningen van g kunnen tegelijk met die van f gedaan worden. We definiëren daartoe een hulpfunctie h met $h\ t = (f\ t,\ g\ t)$:

$h\ (Tip\ n)\ =\ (Tip\ n,\ n)$
$h\ (Node\ n\ t\ t')$
$=\ (Node\ (n{+}k{+}k')\ s\ s',\ n{+}k{+}k')\quad where\ (s, k)\ =\ h\ t;\quad (s', k')\ =\ h\ t'$
$f\ =\ fst\ .\ h$

De accent-versie:

$f'\ (Node'\ n\ ts)\ =\ Node'\ (n + sum\ (map\ g'\ ts))\ (map\ f'\ ts)$
$g'\ (Node'\ n\ ts)\ =\ n + sum\ (map\ g'\ ts)$

Hulpfunctie h' voldoet aan $h'\ t'\ =\ (f'\ t',\ g'\ t')$:

$h'\ (Node'\ n\ ts)$
$=\ (Node'\ n'\ ss,\ n')$
$\quad where$
$\quad (ss,\ ks)\ =\ unzip\ (map\ h'\ ts)$
$\quad n'\ =\ n + sum\ ks$
$f'\ =\ fst\ .\ h'$

Het deel '$unzip\ (map\ h'\ ts)$' kan ook geschreven worden als:

$(map\ fst\ x,\ \ map\ snd\ x)\quad where\ x\ =\ map\ h'\ ts$

195. We veralgemenen f tot $g\ k$; deze verhoogt van links naar rechts de Tip-waarden met $k{+}0$, $k{+}1$, $k{+}2$, ... Bovendien gebruiken we een hulpfunctie *tiptal* die het aantal Tips van z'n argument oplevert:

$g\ k\ (Tip\ n)\ =\ Tip\ (n{+}k)$
$g\ k\ (Node\ n\ t\ t')\ =\ Node\ n\ (g\ k\ t)\ (g\ (k + tiptal\ t)\ t')$
$tiptal\ (Tip\ n)\ =\ 1$
$tiptal\ (Node\ n\ t\ t')\ =\ tiptal\ t + tiptal\ t'$
$f\ =\ g\ 0$

De berekening van *tiptal* kan met die van g gecombineerd worden. Daartoe definiëren we functie h zó dat $h\ k\ t = (g\ k\ t,\ tiptal\ t)$:

$h\ k\ \ (Tip\ n)\ =\ (Tip\ (n{+}k),\ 1)$
$h\ k\ (Node\ n\ t\ t')$
$=\ (Node\ n\ s\ s',\ a{+}a')$
$\quad where$
$\quad (s,\ a)\ =\ h\ k\ t$
$\quad (s',\ a')\ =\ h\ (k{+}a)\ t'$
$f\ =\ fst\ .\ h\ 0$

196. We behandelen alleen het pre-order geval. We veralgemenen f tot $g\ k$; deze verhoogt in pre-order volgorde iedere Node en Tip waarde met $k{+}0$, $k{+}1$, $k{+}2$, ... Bovendien gebruiken we

een hulpfunctie *omvang* die het aantal Nodes en Tips van z'n argument oplevert:

$g\ k\ (Tip\ n)\ =\ Tip\ (n{+}k)$
$g\ k\ (Node\ n\ t\ t')\ =\ Node\ (n{+}k)\ (g\ (k{+}1)\ t)\ (g\ (k{+}1{+}\ omvang\ t)\ t')$
$omvang\ (Tip\ n)\ =\ 1$
$omvang\ (Node\ n\ t\ t')\ =\ 1 + omvang\ t + omvang\ t'$
$f\ =\ g\ 0$

De berekening van *omvang* kan met die van g gecombineerd worden. Daartoe definiëren we functie h zó dat $h\ k\ t = (g\ k\ t,\ omvang\ t)$:

$h\ k\ \ (Tip\ n)\ =\ (Tip\ (n{+}k),\ 1)$
$h\ k\ (Node\ n\ t\ t')$
$\ =\ (Node\ (n{+}k)\ s\ s',\ 1{+}o{+}o')$
$\quad where$
$\quad (s,\ o)\ =\ h\ (k{+}1)\ t$
$\quad (s',\ o')\ =\ h\ (k{+}1{+}o)\ t'$

Voor de accent-versie van g hebben we een hulpfunctie *initsums* nodig met:

$initsums\ k\ [x, y, z, \ldots]\ =\ [k,\ k{+}x,\ k{+}x{+}y,\ k{+}x{+}y{+}z,\ \ldots]$

De definitie met recursie is eenvoudig:

$initsums\ k\ []\ =\ [k]$
$initsums\ k\ (x : xs)\ =\ k :\ initsums\ (k{+}x)\ xs$

De definitie van g' is nu analoog aan die van g:

$g'\ k\ (Node'\ n\ ts)$
$\ =\ Node'\ (n{+}k)\ (zipwith\ g'\ (ks,\ ts))$
$\quad where$
$\quad os\ =\ map\ omvang'\ ts \quad \|\ de\ rij\ van\ omvangen\ der\ subbomen$
$\quad ks\ =\ initsums\ (k{+}1)\ os$

De definitie van h' is analoog aan die van h. We gebruiken daarbij de volgende nomenclatuur: letters s en t staan voor bomen, letter o voor omvang, en verder wordt natuurlijk ook de meervouds-s gebruikt. De variabele *sos* duidt een lijst van paren (s, o) aan.

$h'\ k\ (Node'\ n\ ts)$
$\ =\ (Node'\ (n{+}k)\ ss,\ 1 + sum\ os)$
$\quad where$
$\quad sos\ =\ zipwith\ h'\ (initsums\ (k{+}1)\ os,\ \ ts)$
$\quad (ss,\ os)\ =\ unzip\ sos \quad \|\ =\ (map\ fst\ sos,\ map\ snd\ sos)$

Het opmerkelijke hieraan is dat *sos* en *os* wederzijds recursief worden gedefinieerd. Met de bovengegeven definitie van *initsums* leidt dit in de berekening tot een vicieuze cirkel:

- *om het eerste element van sos uit te rekenen*, moet de aanroep van *zipwith* h' het eerste element van zijn argument kennen; dat vereist dat *initsums* uitgerekend heeft of *os* in het patroon [] dan wel $x : xs$ past; en dus *moet het eerste element van os gedeeltelijk uitgerekend zijn.*
- *om het eerste element van os gedeeltelijk uit te rekenen*, moet de aanroep van *map* het eerste element van *sos* kennen; en dus *moet het eerste element van sos gedeeltelijk uitgerekend zijn.*

Maar met een andere definitie voor *initsums* is verdwijnt de vicieuze cirkel:

$initsums\ k\ xs$

$= k : f\ k\ xs$
$\quad where$
$\quad f\ k\ (x : xs)\ =\ k{+}x : f\ (k{+}x)\ xs$
$\quad f\ k\ []\ =\ []$

Nu kan het eerste element van *initsums* $k\ xs$ al uitgerekend en gebruikt worden, zonder dat het eerste element van xs uitgerekend hoeft te worden (want xs wordt niet direct in een patroon gepast).

197. De preorder opsomming (met plus/min voor Nodes/Tips) wordt opgeleverd door:

$preorderpm\ (Tip\ n)\ =\ [-n]$
$preorderpm\ (Node\ n\ t\ t')\ =\ [n] \mathbin{+\!\!+} preorderpm\ t \mathbin{+\!\!+} preorderpm\ t'$

We veralgemenen f zó dat $f\ xs\ =\ t$ als xs *begint* met $preorderpm\ t$ (en eventueel nog langer is). De definitie van die f ligt voor de hand:

$f\ (x : xs)$
$= Tip\ (-x),\ if\ x < 0$
$= Node\ x\ t\ t',\ if\ 0 < x$
$\quad where$
$\quad t\ =\ f\ xs$
$\quad t'\ =\ f\ (xs \mathbin{-\!-} preorderpm\ t)$

De berekening van $xs \mathbin{-\!-} preorderpm\ t$ kan al met die van $f\ xs$ gecombineerd worden. Daartoe definiëren we functie g zó dat $g\ xs = (f\ xs,\ xs \mathbin{-\!-} preorderpm\ xs)$:

$g\ (x : xs)$
$= (Tip\ (-x),\ xs),\ if\ x < 0$
$= (Node\ x\ t\ t',\ zs),\ if\ 0 < x$
$\quad where$
$\quad (t,\ ys)\ =\ g\ xs$
$\quad (t',\ zs)\ =\ g\ ys$
$f\ =\ fst\ .\ g$

198. $f\ (Tip\ n)\ []\ =\ []$
$f\ (Node\ n\ t\ t')\ (0 : xs)\ =\ n : f\ t\ xs$
$f\ (Node\ n\ t\ t')\ (1 : xs)\ =\ n : f\ t'\ xs$
$g\ (Tip\ n)\ []\ =\ n$
$g\ (Node\ n\ t\ t')\ (0 : xs)\ =\ g\ t\ xs$
$g\ (Node\ n\ t\ t')\ (1 : xs)\ =\ g\ t'\ xs$

De accent-versies:

$f'\ (Node'\ n\ [])\ []\ =\ []$
$f'\ (Node'\ n\ ts)\ (x : xs)\ =\ n : f'\ (ts!x)\ xs$
$g'\ (Node'\ n\ [])\ []\ =\ n$
$g'\ (Node'\ n\ ts)\ (x : xs)\ =\ g'\ (ts!x)\ xs$

199. $fInv\ (Tip\ n)\ []\ =\ [[]]$
$fInv\ (Tip\ n)\ xs\ =\ []$
$fInv\ (Node\ n\ t\ t')\ (x : xs)$
$= [],\ if\ n \neq x$
$= map\ (0 :)\ (fInv\ t\ xs) \mathbin{+\!\!+} map\ (1 :)\ (fInv\ t'\ xs),\ otherwise$

$gInv\ (Tip\ n)\ x$
$= [[]],\ if\ n = x$
$= [],\ otherwise$
$gInv\ (Node\ n\ t\ t')\ x\ =\ map\ (0 :)\ (gInv\ t\ x)\ +\!\!+\ map\ (1 :)\ (gInv\ t'\ x)$

De Tip-clausules van *fInv* en *gInv* kunnen ook in één regel:

$fInv\ (Tip\ n)\ xs\ =\ [[]\ |\ xs = []]$
$gInv\ (Tip\ n)\ x\ =\ [[]\ |\ n = x]$

De accent-versies:

$fInv'\ (Node'\ n\ [])\ xs\ =\ [[]\ |\ xs = []]$
$fInv'\ (Node'\ n\ ts)\ (x : xs)$
$= [],\ if\ n \neq x$
$= concat\ (zipwith\ f\ (ts,\ index\ ts)),\ otherwise$
$\quad where$
$\quad f\ t\ i\ =\ map\ (i :)\ (fInv'\ t\ xs)$
$gInv'\ (Node'\ n\ [])\ x\ =\ [[]\ |\ n = x]$
$gInv'\ (Node'\ n\ ts)\ x$
$= concat\ (zipwith\ f\ (ts,\ index\ ts))$
$\quad where$
$\quad f\ t\ i\ =\ map\ (i :)\ (gInv'\ t\ x)$

200. Er zijn vele varianten voor *mobiel* mogelijk:

$mobiel\ ::=\ Gew\ num\ \mid\ Arm\ (num,\ mobiel)\ (num,\ mobiel)$
$mobiel\ ::=\ Gew\ num\ \mid\ Arm\ num\ mobiel\ num\ mobiel$
$mobiel\ ::=\ Gew\ num\ \mid\ Arm\ num\ mobiel\ mobiel\ num$
$mobiel\ ::=\ Gew\ num\ \mid\ Arm\ (num,\ num)\ (mobiel,\ mobiel)$
$mobiel\ ::=\ Gew\ num\ \mid\ Arm\ (num,\ mobiel,\ num,\ mobiel)$

Hieronder zullen wij van de eerste definitie uit gaan.
Twee schrijfwijzen voor mobiel m:

$m\ =\ Arm\ (5,\ Arm\ (8,\ Gew\ 1)\ (2,\ Gew\ 4))\ (3,\ Gew\ 9)$
$m\ =$
$\quad Arm\ (5, m1)\ (3, m2)$
$\quad where$
$\quad m1\ =\ Arm\ (8,\ Gew\ 1)\ (2,\ Gew\ 4)$
$\quad m2\ =\ Gew\ 9$

201. $isBal\ (Gew\ n)\ =\ True$
$isBal\ (Arm\ (n, m)\ (n', m'))\ =\ n \times gew\ m\ =\ n' \times gew\ m'\ \wedge\ isBal\ m\ \wedge\ isBal\ m'$

De veronderstelde definitie van *gew* luidt:

$gew\ (Gew\ n)\ =\ n$
$gew\ (Arm\ (n, m)\ (n', m'))\ =\ gew\ m + gew\ m'$

202. We definiëren bal' zó dat $bal'\ m = (isBal\ m,\ gew\ m)$:

$bal'\ (Gew\ n)\ =\ (True,\ n)$
$bal'\ (Arm\ (n, m)\ (n', m'))$
$= (n{\times}g = n'{\times}g'\ \wedge\ b\ \wedge\ b',\quad g{+}g')$

$where$
$(b, g) \;=\; bal'\ m$
$(b', g') \;=\; bal'\ m'$

Dus nu kunnen we definiëren:

$isBal \;=\; fst\ .\ bal'$

203. $expressie\ ::=\ Getal\ num \mid Plus\ expressie\ expressie \mid Maal\ expressie\ expressie$

$showExpr\ ::\ expressie \rightarrow [char]$
$showExpr\ (Getal\ n) \;=\; shownum\ n$
$showExpr\ (Plus\ e\ e') \;=\; behaakt\ (showExpr\ e\ +\!\!+\ \texttt{" + "}\ +\!\!+\ showExpr\ e')$
$showExpr\ (Maal\ e\ e') \;=\; behaakt\ (showExpr\ e\ +\!\!+\ \texttt{" × "}\ +\!\!+\ showExpr\ e')$

$behaakt\ xs \;=\; \texttt{"("}\ +\!\!+\ xs\ +\!\!+\ \texttt{")"}$

204. $showExprZ\ (Getal\ n) \;=\; shownum\ n$
$showExprZ\ (Plus\ e\ e') \;=\; onhaakt\ (showExprZ\ e\ +\!\!+\ \texttt{" + "}\ +\!\!+\ showExprZ\ e')$
$showExprZ\ (Maal\ e\ e') \;=\; onhaakt\ (showExprH\ e\ +\!\!+\ \texttt{" × "}\ +\!\!+\ showExprH\ e')$

Functie *showExprH* is identiek aan *showExprZ* (zie de laatste clausule hieronder), behalve wanneer zijn argument een Plus-expressie is — dan komen er haakjes omheen:

$showExprH\ (Plus\ e\ e') \;=\; behaakt\ (showExprZ\ (Plus\ e\ e'))$
$showExprH\ e \;=\; showExprZ\ e$

205. $showExprP\ (Getal\ n) \;=\; shownum\ n$
$showExprP\ (Plus\ e\ e') \;=\; \texttt{"+ "}\ +\!\!+\ showExprP\ e\ +\!\!+\ \texttt{" "}\ +\!\!+\ showExprP\ e'$
$showExprP\ (Maal\ e\ e') \;=\; \texttt{"× "}\ +\!\!+\ showExprP\ e\ +\!\!+\ \texttt{" "}\ +\!\!+\ showExprP\ e'$

206. $exprVal\ (Getal\ n) \;=\; n$
$exprVal\ (Plus\ e\ e') \;=\; exprVal\ e + exprVal\ e'$
$exprVal\ (Maal\ e\ e') \;=\; exprVal\ e \times exprVal\ e'$

207. $stdVorm\ (Getal\ n) \;=\; Getal\ n$
$stdVorm\ (e\ \underline{Plus}\ e') \;=\; stdVorm\ e\ \underline{Plus}\ stdVorm\ e'$
$stdVorm\ (e\ \underline{Maal}\ e') \;=\; \ldots$

De laatste clausule vergt enige oplettendheid. De uitkomst moet qua waarde gelijk zijn aan $stdVorm\ e\ \underline{Maal}\ stdVorm\ e'$, maar deze expressie is misschien *niet* in standaard vorm: namelijk wanneer $stdVorm\ e$ of $stdVorm\ e'$ een Plus-expressie is. We kunnen niet zomaar $stdVorm\ (stdVorm\ e\ \underline{Maal}\ stdVorm\ e')$ als uitkomst nemen, want dan treedt er oneindige recursie op, bijvoorbeeld bij $Getal\ 1\ \underline{Maal}\ Getal\ 2$. We gebruiken daarom een hulp-operatie $\underline{maal}$ en definiëren:

$stdVorm\ (e\ \underline{Maal}\ e') \;=\; stdVorm\ e\ \underline{maal}\ stdVorm\ e'$

Operatie $\underline{maal}$ stelt een $\underline{Maal}$ voor en levert een standaard vorm op, gegeven dat beide argumenten in standaard vorm staan. (Hieronder gebruiken we x, y en z uitsluitend voor expressies in standaard vorm.) De definitie luidt:

$x\ \underline{maal}\ (y\ \underline{Plus}\ z) \;=\; (x\ \underline{maal}\ y)\ \underline{Plus}\ (x\ \underline{maal}\ z)$

$x\ \underline{maal}\ y\ =\ x\ \underline{maal'}\ y\ \|\ y\ is\ geen\ Plus,\ x\ is\ mogelijk\ Plus$

Operatie $\underline{maal'}$ stelt een $\underline{Maal}$ voor en levert een standaard vorm op, gegeven dat beide argumenten in standaard vorm staan en de rechter geen Plus is.

$(x\ \underline{Plus}\ y)\ \underline{maal'}\ z\ =\ (x\ \underline{maal'}\ z)\ \underline{Plus}\ (y\ \underline{maal'}\ z)$
$x\ \underline{maal'}\ y\ =\ x\ \underline{Maal}\ y$

208. *expressie* ::=
Var [*char*] | *Getal num* |
Plus expressie expressie | *Maal expressie expressie*

Bij *showExpr*, *showExprZ*, *showExprP* en *stdVorm* komt er één clausule bij:

showExpr (*Var a*) = *a*
showExprP (*Var a*) = *a*
stdVorm (*Var a*) = *Var a*

Bij *maal* en $maal'$ en *showExprH* hoeft er geen clausule bij omdat bij elk van de gegeven definities de laatste clausule een patroon heeft waarin alles past, dus ook *Var a*.
De definitie van *exprVal* moet geheel aangepast worden; bij elke aanroep komt er een omgevings-argument *omg* bij:

exprVal omg (*Var a*) = *omg a*
exprVal omg (*Getal n*) = *n*
$exprVal\ omg\ (Plus\ e\ e')\ =\ exprVal\ omg\ e + exprVal\ omg\ e'$
$exprVal\ omg\ (Maal\ e\ e')\ =\ exprVal\ omg\ e \times exprVal\ omg\ e'$

209. *expressie* ::= *Getal num* | *Opex operator expressie expressie*
operator ::= *Plus* | *Min* | *Deel* | *Maal*

De naam 'Opex' komt van 'operator-expressie'; in plaats van de namen 'Getal' en 'Opex' hadden we ook kunnen nemen: 'Elementair' en 'Samengesteld'.
De gegevens van de operatoren leggen we eerst vast:

opGegevens
$= [(Plus, 1, "+", (+)),\ (Min, 1, "-", (-)),\ (Deel, 2, "/", (/)),\ (Maal, 2, "\times", (\times))]$
$prio\ op\ =\ hd\ [p \mid (o, p, s, v) \leftarrow opGegevens;\ o = op]$
$opVal\ op\ =\ hd\ [v \mid (o, p, s, v) \leftarrow opGegevens;\ o = op]$
$showOp\ op\ =\ hd\ [s \mid (o, p, s, v) \leftarrow opGegevens;\ o = op]$

De functie-definities van *showExpr*, *showExprZ*, *showExprP* en *exprVal* zijn analoog aan die in opgave 203, 204, 205 en 206. Bij het zuinig tonen van $e \oplus e'$ moeten er haakjes om e en e' als $\oplus$ voorrang heeft op de operator van e respectievelijk e'; bij e' komen er bovendien ook bij *gelijke* prioriteiten haakjes als $\oplus$ niet associatief is, zoals Min en Deel:

showExprZ (*Getal n*) = *shownum n*
$showExprZ\ (Opex\ op\ e\ e')$
$=\ \|\ niet{-}associatieve\ operatoren :$
$showExprH\ (prio\ op)\ e\ +\!\!+\ "\ "\ +\!\!+\ showOp\ op\ +\!\!+\ "\ "\ +\!\!+\ showExprH'\ (prio\ op)\ e',$
$if\ op = Min\ \vee\ op = Deel$
$=\ \|\ associatieve\ operatoren$
$showExprH\ (prio\ op)\ e\ +\!\!+\ "\ "\ +\!\!+\ showOp\ op\ +\!\!+\ "\ "\ +\!\!+\ showExprH\ (prio\ op)\ e',$
otherwise

```
|| showExprH p e :− haakjes om e als z'n prioriteit kleiner is dan p :
showExprH p (Opex op e e')
 = "(" ++ showExprZ (Opex op e e') ++ ")", if prio op < p
showExprH p e = showExprZ e

|| showExprH' p e :− haakjes om e als z'n prioriteit kleiner of gelijk p is :
showExprH' p (Opex op e e')
 = "(" ++ showExprZ (Opex op e e') ++ ")", if prio op <= p
showExprH' p e = showExprZ e
```

Bij de definities van *showExprH* en *showExprH'*, en ook bij sommige definities hieronder, gebruiken we het feit dat in Miranda de tweede clausule van een definitie wordt gekozen als het argument wél past bij de eerste clausule maar de bewakingsconditie niet vervuld is. Dat is niet in alle talen zo. Door de aanroep *showExprH'* (*prio op*) e' te vervangen door *showExprH* (1 + *prio op*) e' wordt de hulpfunctie *showExprH'* overbodig.

```
|| showExprP e : toont e in Poolse notatie (operatoren in prefix notatie) :
showExprP (Getal n) = shownum n
showExprP (Opex op e e') = showOp op ++ " " ++ showExprP e ++ " " ++ showExprP e'

exprVal (Getal n) = n
exprVal (Opex op e e') = opVal op (exprVal e) (exprVal e')
```

De functie *stdVorm* is het lastigst. Het allerbelangrijkste is een werkbare specificatie te geven van wat een standaard vorm is. Wij maken onder andere de keuze dat de standaard vorm van $(a+b)+(c+d)$ de expressie $((a+b)+c)+d$ is, en net zo is de standaard vorm van $(a+b)-(c+d)$ de expressie $((a+b)-c)-d$:

```
|| stdVorm e = een expr, met gelijke exprVal als e, en de eigenschap dat
|| de argumenten van Maal zijn geen Plus, Min, of Deel, en
|| de argumenten van Deel zijn geen Deel, en
|| het rechter argument van Plus en Min is geen Plus of Min.

stdVorm (Opex Maal e e') = maal (stdVorm e) (stdVorm e')
stdVorm (Opex Deel e e') = deel (stdVorm e) (stdVorm e')
stdVorm (Opex Min e e') = minus (stdVorm e) (stdVorm e')
stdVorm (Opex Plus e e') = plus (stdVorm e) (stdVorm e')
stdVorm (Getal n) = Getal n

|| Functies maal, deel, minus, plus leveren de standaard vorm op
|| mits beide argumenten al in std vorm staan.
```

We definiëren *maal*, *deel*, *minus* en *plus* wederzijds recursief, met inductie naar de opbouw van hun rechter argument. De hulpfuncties *maal'* en *deel'* worden gedefinieerd met inductie naar de opbouw van hun linker argument:

```
|| Hieronder : x, y, z zijn expressies in standaard vorm.
|| Ook functies maal', deel' leveren std vorm op
|| mits beide argumenten al in standaard vorm staan.

maal x (Opex Plus y z) = plus (maal x y) (maal x z)
maal x (Opex Min y z) = minus (maal x y) (maal x z)
```

$maal\ x\ (Opex\ Deel\ y\ z)\ =\ deel\ (maal\ x\ y)\ z$
$maal\ x\ y\ =\ maal'\ x\ y$

$\|\ maal'$: *neem aan dat rechter arg geen Plus, Min, of Deel expressie is.*
$maal' :: expressie \rightarrow expressie \rightarrow expressie$
$maal'\ (Opex\ Plus\ x\ y)\ z\ =\ plus\ (maal'\ x\ z)\ (maal'\ y\ z)$
$maal'\ (Opex\ Min\ x\ y)\ z\ =\ minus\ (maal'\ x\ z)\ (maal'\ y\ z)$
$maal'\ (Opex\ Deel\ x\ y)\ z\ =\ Opex\ Deel\ (maal'\ x\ z)\ y$
$maal'\ x\ y\ =\ Opex\ Maal\ x\ y$

$deel\ x\ (Opex\ Deel\ y\ z)\ =\ deel'\ (maal\ x\ z)\ y$
$deel\ x\ y\ =\ deel'\ x\ y$

$\|\ deel'$: *neem aan dat rechter argument geen Deel expressie is.*
$deel'\ (Opex\ Deel\ x\ y)\ z\ =\ deel'\ x\ (maal\ y\ z)$
$deel'\ x\ y\ =\ Opex\ Deel\ x\ y$

$minus\ x\ (Opex\ Min\ y\ z)\ =\ Opex\ Plus\ (minus\ x\ y)\ z$
$minus\ x\ (Opex\ Plus\ y\ z)\ =\ Opex\ Min\ (minus\ x\ y)\ z$
$minus\ x\ y\ =\ Opex\ Min\ x\ y$

$plus\ x\ (Opex\ Min\ y\ z)\ =\ Opex\ Min\ (plus\ x\ y)\ z$
$plus\ x\ (Opex\ Plus\ y\ z)\ =\ Opex\ Plus\ (plus\ x\ y)\ z$
$plus\ x\ y\ =\ Opex\ Plus\ x\ y$

210. $propositie\ ::=\ Con\ constant \mid Neg\ propositie \mid propositie\ \underline{Pijl}\ propositie$
$constant\ \equiv\ num$

Con i representeert constante $\mathbf{a}_i$ (voor $i = 0, 1, \ldots$), en *Neg* en *Pijl* representeren connectieven $\neg$ en $\Rightarrow$. De voorbeeld-proposities worden als volgt gerepresenteerd:

$$\begin{array}{lcl}
\mathbf{a}_{17} & \approx & Con\ 17 \\
(\mathbf{a}_7 \Rightarrow (\neg \mathbf{a}_3)) & \approx & Con\ 7\ \underline{Pijl}\ Neg\ (Con\ 3) \\
((\mathbf{a}_0 \Rightarrow (\neg \mathbf{a}_1)) \Rightarrow ((\neg \mathbf{a}_1) \Rightarrow \mathbf{a}_0)) & \approx & (Con\ 0\ \underline{Pijl}\ Neg\ (Con\ 1))\ \underline{Pijl}\ \ldots
\end{array}$$

In de rest van de antwoorden zijn '*num*' en '*constant*' voor elkaar uitwisselbaar. Dat was niet zo geweest als we gedefinieerd hadden: $constant\ ::=\ C\ num$.

211. $v\ ::\ constant \rightarrow num$
$val\ ::\ propositie \rightarrow num$

Om *val* te definiëren zijn er verscheidene mogelijkheden. Volgens de suggestie uit de opgave:

$val\ (p\ \underline{Pijl}\ q)\ =\ 0,\ if\ val\ p\ =\ 1\ \wedge\ val\ q\ =\ 0$
$val\ (Neg\ p)\ =\ 0,\ if\ val\ p\ =\ 1$
$val\ (Con\ i)\ =\ v\ i$
$val\ p\ =\ 1$

Ietwat mooier is de volgende definitie:

$val\ (Con\ i)\ =\ v\ i$
$val\ (Neg\ p)\ =\ 1 - val\ p$
$val\ (p\ \underline{Pijl}\ q)\ =\ max\ [1 - val\ p,\ val\ q]$

212. $val\ ::\ (constant \to num) \to propositie \to num$

Herzie de definitie van *val* door bij iedere aanroep van *val* de v als argument erbij te zetten:

$val\ v\ (Con\ i)\ =\ v\ i$
$val\ v\ (Neg\ p)\ =\ 1 - val\ v\ p$
$val\ v\ (p\ \underline{Pijl}\ q)\ =\ max\ [1 - val\ v\ p,\ val\ v\ q]$

213. $consts,\ consts'\ ::\ propositie \to [constant]$
$consts\ =\ mkset\ .\ consts'$
$consts'\ (Con\ i)\ =\ [i]$
$consts'\ (Neg\ p)\ =\ consts'\ p$
$consts'\ (p\ \underline{Pijl}\ q)\ =\ consts'\ p\ +\!\!+\ consts'\ q$

214. $isTaut\ p\ =\ and\ [val\ v\ p\ =\ 1 \mid v \leftarrow vs\ (consts\ p)]$

215. $vs\ ::\ [constant] \to [\ constant {\to} num\ ]$
$vs\ []\ =\ [\ldots]$
$vs\ (c : cs)\ =\ [v\ \underline{met}\ (c,0) \mid v \leftarrow vs\ cs]\ +\!\!+\ [v\ \underline{met}\ (c,1) \mid v \leftarrow vs\ cs]$

Het doet er niet toe wat er op de plaats van ... staat (mits van het goede type), bijvoorbeeld *undef*. Functie $v\ \underline{met}\ (c,k)$ is een valuatie die gelijk is aan v behalve dat op argument c het resulaat k opgeleverd wordt:

$met\ ::\ (constant \to num) \to (constant,\ num) \to (constant \to num)$
$(v\ \underline{met}\ (c,k))\ c'\ =\ k,\ if\ c' = c$
$(v\ \underline{met}\ (c,k))\ c'\ =\ v\ c'$

216. Semantisch wel correct, maar niet door Miranda geaccepteerd:

$bagunion\ xs\ ys\ =\ xs\ +\!\!+\ ys$

Het type van *bagunion* is $[\alpha] \to [\alpha] \to [\alpha]$. Vanwege de abstype-verklaring is $[\alpha]$ ongerelateerd aan $bag\ \alpha$. Dus de type-specificatie $bagunion\ ::\ bag\ \alpha \to bag\ \alpha \to bag\ \alpha$ en een expressie zoals *bagunion empty empty* wordt door Miranda niet geaccepteerd.

Oplossing 1: voeg de regel $bagunion\ ::\ bag\ \alpha \to bag\ \alpha \to bag\ \alpha$ toe aan de abstype-verklaring.

Oplossing 2a: Voeg een functie toe aan de abstype-verklaring waarmee je de beschikking krijgt over de individuele elementen in de bag. Bijvoorbeeld:

$isEmpty\ ::\ bag\ \alpha \to bool$
$any\ ::\ bag\ \alpha \to \alpha$

met definitie:

$isEmpty\ xs\ =\ xs = []$
$any\ (x : xs)\ =\ x$

We kunnen dan definiëren:

$bagunion\ ::\ bag\ \alpha \to bag\ \alpha \to bag\ \alpha$
$bagunion\ xs\ ys\ =\ ys,\ if\ isEmpty\ xs$
$bagunion\ xs\ ys\ =\ bagunion\ (del\ xs\ x)\ (add\ xs\ x)\ where\ x\ =\ any\ xs$

Oplossing 2b: Een andere mogelijkheid is aan de abstype-verklaring toe te voegen:

$bag2list\ ::\ bag\ \alpha \to [\alpha]$

met definitie:

$bag2list\ xs\ =\ xs$

We kunnen dan definiëren:

$bagunion\ ::\ bag\ \alpha \rightarrow bag\ \alpha \rightarrow bag\ \alpha$
$bagunion\ xs\ ys\ =\ foldl\ add\ xs\ (bag2list\ ys)$

217. Van onze definities van 'vereniging' (*bagunion*) behoeven alleen diegene geen verandering, die gespecificeerd zijn met type $bag\ \alpha \rightarrow bag\ \alpha \rightarrow bag\ \alpha$.
De aanpassing van de 'abstype-ingrediënten' luidt:

$tree\ \alpha\ ::=\ Nil \mid Node\ (tree\ \alpha)\ \alpha\ (tree\ \alpha)$
$\|\ forall\ (Node\ t\ y\ t')\ :\ forall\ x\ in\ t,\ z\ in\ t'\ :\ \ x\ \leq\ y\ <\ z$

$bag\ \alpha\ \equiv\ tree\ \alpha$

$empty\ =\ Nil$
$nbr\ Nil\ x\ =\ 0$
$nbr\ (Node\ t\ y\ t')\ x$
$=\ \#[1|y = x] + nbr\ t\ x,\ if\ x \leq y$
$=\ nbr\ t'\ x,\ if\ y < x$
$add\ Nil\ x\ =\ Node\ Nil\ x\ Nil$
$add\ (Node\ t\ y\ t')\ x$
$=\ Node\ (add\ t\ x)\ y\ t',\ if\ x \leq y$
$=\ Node\ t\ y\ (add\ t'\ x),\ if\ y < x$
$del\ Nil\ x\ =\ Nil$
$del\ (Node\ t\ y\ t')\ x\ =\ Node\ (del\ t\ x)\ y\ t',\ if\ x < y$
$del\ (Node\ t\ y\ t')\ x\ =\ Node\ t\ y\ (del\ t'\ x),\ if\ y < x$
$\|\ in\ de\ volgende\ drie\ del\ clausules\ geldt\ \ x = y$
$del\ (Node\ Nil\ y\ Nil)\ x\ =\ Nil$
$\|\ in\ de\ volgende\ twee\ del\ clausules\ zijn\ t\ en\ t'\ niet\ Nil$
$del\ (Node\ t\ y\ Nil)\ x\ =\ Node\ (del\ t\ z)\ z\ Nil\ \ \ where\ z\ =\ treemax\ t$
$del\ (Node\ Nil\ y\ t')\ x\ =\ Node\ Nil\ z'\ (del\ t'\ z')\ \ \ where\ z'\ =\ treemax\ t'$
$treemax\ (Node\ t\ y\ Nil)\ =\ y$
$treemax\ (Node\ t\ y\ t')\ =\ treemax\ t'$

Zoals opgemerkt in het antwoord van opgave 216, is het verstandig om ook een *any* en *isEmpty* in de abstype op te nemen, danwel een *bag2list*:

$isEmpty\ t\ =\ t\ =\ Nil$
$any\ (Node\ t\ y\ t')\ =\ y$
$bag2list\ Nil\ =\ []$
$bag2list\ (Node\ t\ y\ t')\ =\ bag2list\ t\ +\!\!+\ [y]\ +\!\!+\ bag2list\ t'$

218. $bag\ \alpha\ \equiv\ [\alpha]$
$empty\ =\ []$
$nbr\ xs\ x\ =\ \#dropwhile\ (< x)\ (takewhile\ (\leq x)\ xs)$
$add\ xs\ x\ =\ takewhile\ (< x)\ xs\ +\!\!+\ [x]\ +\!\!+\ dropwhile\ (< x)\ xs$
$del\ xs\ x$

$= \ takewhile\ (< x)\ xs \ +\!\!+\ drop\ 1\ (takewhile\ (= x)\ (dropwhile\ (< x)\ xs))\ +\!\!+$
$\quad dropwhile\ (\leq x)\ xs$

Een efficiënte definitie voor *bagunion* gebruikt de geordendheid van de representatie:

$bagunion \ = \ merge$

(De *merge* is een standaard functie.) Het type hiervan is $[\alpha]\rightarrow[\alpha]\rightarrow[\alpha]$, en dat is *niet* de gewenste $bag\ \alpha \rightarrow bag\ \alpha \rightarrow bag\ \alpha$. Door $bagunion\ ::\ bag\ \alpha \rightarrow bag\ \alpha \rightarrow bag\ \alpha$ op te nemen in de abstype-verklaring, kun je wél van de geordendheid van de representatie gebruik maken in de implementatie van 'vereniging'.

219. Herdefinieer *rdNumS* als volgt:

$rdNumS \ = \ rdStrS\ \ \underline{sq}\ \ (return\ .\ numval)$
$rdStrS$
$= \ echoOff\ \ \underline{sqn}$
$\quad rdStr\ \ \underline{sq}\ \ f\ where\ f\ xs \ =$
$\quad wrStr\ ([\, '-'\ |\ x \leftarrow xs] +\!\!+ \texttt{"}\backslash n\texttt{"})\ \ \underline{sqn}$
$\quad echoOn\ \ \underline{sqn}$
$\quad return\ xs$

220. Eerst een handige hulpfunctie *rrdNum* (robuuste *rdNum*):

$rrdNum$
$= \ rdStr\ \ \underline{sq}\ \ f\ where\ f\ xs$
$\qquad = \ return\ (numval\ xs),\ \ if\ \ and\ [digit\ x\ |\ x \leftarrow xs]$
$\qquad = \ wrStr\ \texttt{"}Alleen\ cijfers:\ \texttt{"}\ \ \underline{sqn}\ \ rrdNum,\ \ otherwise$

Nu hoeft in de definitie van *hooglaag* alleen maar iedere *rdNum* vervangen te worden door *rrdNum*.

221. $iohandler\ \alpha \ \equiv\ [char] \rightarrow (([char],\ [sys\_message]),\ \alpha)$
$return\ x\ input \ = \ ((input,\ []),\ x)$
$wrStr\ xs\ input \ = \ ((input,\ [Stdout\ xs]),\ ())$
$do\ xs\ input \ = \ ((input,\ [System\ xs]),\ ())$
$rdChar\ (x : xs) \ = \ ((xs,\ []),\ x)$
$sq\ f\ g\ in$
$= \ ((in2,\ out1 +\!\!+ out2),\ r2)$
$\quad where$
$\quad ((in1,\ out1),\ r1) \ = \ f\ in$
$\quad ((in2,\ out2),\ r2) \ = \ g\ r1\ in1$
$run\ f \ = \ output' \qquad where\ ((input',\ output'),\ x) \ = \ f\ \$-$

222. Beschouw een dialoog zoals:

$rdStr\ \ \underline{sq}\ \ f\ where\ f\ xs \ =$
$wrStr\ (\texttt{"}Dat\ was:\ \texttt{"}\ +\!\!+\ xs)$

Wanneer deze ge*run*d wordt, verschijnt de tekst '`Dat was: `' al op het scherm vóórdat de gebruiker iets heeft ingetikt. De rekenregels die Miranda hanteert maken dat mogelijk en dus zal Miranda dat doen. Het euvel wordt verholpen door op kunstmatige manier de output van *rdStr* afhankelijk te maken van de toetsaanslagen die *rdStr* consumeert. Bijvoorbeeld:

$rdStr\ input$
$= \ ((dropline\ input,\ []),\ xs),\ \ if\ xs = xs$

$$\quad \mathit{where}\ \mathit{xs}\ =\ \mathit{takeline}\ \mathit{input}$$

Een andere manier is de standaard functies *seq* en *force* te gebruiken:

$$\begin{array}{l}
\mathit{rdStr}\ \mathit{input}\\
=\ \mathit{seq}\ (\mathit{force}\ \mathit{xs})\ ((\mathit{dropline}\ \mathit{input},\ []),\ \mathit{xs})\\
\quad \mathit{where}\ \mathit{xs}\ =\ \mathit{takeline}\ \mathit{input}
\end{array}$$

De uitdrukking $\mathit{seq}\ (\mathit{force}\ \mathit{xs})\ y$ levert y op, maar pas nadat xs is uitgerekend ('enigszins' door *seq* en 'helemaal' door *force*).

223. Uit de gebruikelijke schrijfwijze volgt dat elk sluithaakje direct na de dichtstbijzijnde f komt:

$$\begin{array}{l}
\mathit{linkerlid}\\
=\ (\mathit{xxx}1\ \ \underline{\mathit{sqn}}\\
\quad (\mathit{xxx}2\ \ \underline{\mathit{sq}}\ f))\ \mathit{where}\ f\ x\ =\\
\quad (\mathit{xxx}3\ \ \underline{\mathit{sqn}}\\
\quad (\mathit{xxx}4\ \ \underline{\mathit{sq}}\ f))\ \mathit{where}\ f\ y\ =\\
\quad (\mathit{xxx}5\ \ \underline{\mathit{sqn}}\\
\quad \mathit{xxx}6\ \ \underline{\mathit{sq}}\ f))\ \mathit{where}\ f\ z\\
\quad \ldots
\end{array}$$

Dit toont een nadeel van ons idioom: de schrijfwijze suggereert ten onrechte dat de sluithaakjes allemaal bij elkaar komen aan het eind.

224. $\mathit{sys\_message}\ ::=\ \mathit{System}\ [\mathit{char}]\ |\ \mathit{Stdout}\ [\mathit{char}]\ |\ \ldots$

225. De versie hieronder is niet robuust:

$$\begin{array}{l}
\mathit{hangman}\\
=\ \mathit{wrStr}\ \texttt{"}\mathit{Woord}:\ \texttt{"}\ \ \underline{\mathit{sqn}}\\
\quad \mathit{echoOff}\ \ \underline{\mathit{sqn}}\\
\quad \mathit{rdStr}\ \ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ \mathit{woord}\ =\\
\quad \mathit{echoOn}\ \ \underline{\mathit{sq}}\\
\quad \mathit{wrStr}\ \texttt{"}\mathit{Begin\ maar.}\backslash n\texttt{"}\ \ \underline{\mathit{sqn}}\\
\quad \mathit{hangman}'\ (\mathit{woord},\ [])\\
\mathit{hangman}'\ (\mathit{woord},\ \mathit{ls})\\
=\ \mathit{wrStr}\ \texttt{"}\mathit{Letter}:\ \texttt{"}\ \ \underline{\mathit{sqn}}\\
\quad \mathit{rdChar}\ \ \underline{\mathit{sq}}\ f\ \mathit{where}\ f\ l\\
\quad\ =\ \ \mathit{wrStr}\ (\texttt{"}\mathit{Score}:\ \texttt{"}\ +\!\!+\ \mathit{score}\ +\!\!+\ \texttt{"}.\backslash n\texttt{"})\ \ \underline{\mathit{sqn}}\ \ \mathit{hangman}'\ (\mathit{word},\ l : \mathit{ls}),\\
\qquad \mathit{if}\ \mathit{member}\ \mathit{score}\ '{-}'\\
\quad\ =\ \ \mathit{wrStr}\ (\texttt{"}\mathit{Yesss}:\ \texttt{"}\ +\!\!+\ \mathit{score}),\\
\qquad \mathit{otherwise}\\
\quad \mathit{where}\\
\quad \mathit{score}\ =\ \mathit{map}\ \mathit{dash}\ \mathit{word}\\
\quad \mathit{dash}\ c\ =\ \mathit{hd}\ ([c|\ \mathit{member}\ (l : \mathit{ls})\ c]\ +\!\!+\ ['{-}'])
\end{array}$$

226.
$$\begin{array}{l}
\mathit{priems}\ =\ \mathit{zeef}\ [2..]\\
\mathit{zeef}\ (x : \mathit{xs})\ =\ x\ :\ \mathit{zeef}\ [y\ |\ y \leftarrow \mathit{xs};\ y\ \mathit{mod}\ x\ \neq\ 0]
\end{array}$$

227.

$$\mathit{ham}\ =\ 1:\ \mathit{map}\ (2\times)\ \mathit{ham}\ \ \underline{\mathit{merge}'}\ \ \mathit{map}\ (3\times)\ \mathit{ham}\ \ \underline{\mathit{merge}'}\ \ \mathit{map}\ (5\times)\ \mathit{ham}$$

Functie merge' is in opgave 141 behandeld.

228. Allereerst twee hulpfuncties:

$regel\ i\ =\ \texttt{"}Regel\texttt{ "}\ +\!\!+\ shownum\ i\ +\!\!+\ \texttt{" }luidt: \texttt{"}$

$quote\ r\ =\ \texttt{"\textbackslash""}\ +\!\!+\ r\ +\!\!+\ \texttt{"\textbackslash""}$

Een mogelijke oplossing, zónder regelnummers, luidt:

$regels\ =\ mkregels\ 1$

$mkregels\ i\ =\ regel\ i\ :\ quote\ (regels'!(i-1)) :\ mkregels\ (i+1)$

En met regelnummers:

$nr\ i\ =\ rjustify\ 3\ (shownum\ i)\ +\!\!+\ \texttt{") "}$

$regels\ =\ mkregels\ 1\ 1$

$mkregels\ i\ j$

$\ =\ (nr\ i\ +\!\!+\ regel\ j)\ :\ (nr\ (i+1)\ +\!\!+\ quote\ (regels!(j-1))) :\ mkregels\ (i+2)\ (j+1)$

Hier is nog een andere oplossing (zonder gebruik te maken van indicering !):

Zónder regelnummers:

$quoteRegel\ r\ i\ =\ (quote\ r,\ regel\ i)$

$mklist\ ((x, y) : zs)\ =\ x :\ y :\ mklist\ zs$

$regels\ =\ regel\ 1 :\ mklist\ (zipwith\ quoteRegel\ (regels,\ [2..]))$

Mét regelnummers; alleen de definitie van *regels* luidt iets anders:

$regels\ =\ lines\ (layn\ (regel\ 1 :\ mklist\ (zipwith\ quoteRegel\ (regels,\ [2..]))\ ))$

Of met functie-compositie:

$regels\ =\ (lines\ .\ layn\ .\ (regel\ 1 :)\ .\ mklist\ .\ zipwith\ quoteRegel)\ (regels,\ [2..])$