Functioneel Programmeren
– Opdrachtenbundel –

Peter Achten
(P.Achten@cs.ru.nl)

Model Based Software Development,
Institute for Computing and Information Sciences,
Radboud University Nijmegen,
Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands.

Versie 27 maart 2015

Voorwoord

Deze opdrachtenbundel bevat opdrachten die de gehele stof beslaan van de cursus *Functioneel Programmeren (voor KI-studenten)*. Met deze opdrachten kun je je bekwamen in de stof die tijdens het hoorcollege aan bod is gekomen. Dit is tevens een goede voorbereiding voor het tentamen.

We maken gebruik van de Clean **2.4** distributie. Deze is gratis te downloaden vanaf de site: `http://wiki.clean.cs.ru.nl/`

De opdrachten zijn thematisch geordend. Sommige opdrachten zijn nodig voor vervolgopdrachten. In dat geval staat naast de opdracht de vervolgopdracht genoemd.

De opdrachten hebben een naam. Deze komt overeen met de naam van de 'main module' van de bijbehorende Clean implementatie. Bij iedere opdracht wordt de naam van de desbetreffende main module genoemd. Alle main modulen zijn beschikbaar gemaakt in een 'Practicum' folder die je van de cursus-site kunt downloaden als .zip bestand. Pak dit bestand uit en zet het in je Clean directory onder *Examples*.

De bundel wordt regelmatig bijgewerkt om goed aan te sluiten bij de stof en om opdrachten te verbeteren of toe te voegen. Je bent welkom om opmerkingen, vragen, tips, fouten in opdrachten (hopelijk geen) en dergelijke te sturen zodat dit verwerkt kan worden in volgende versies van de bundel.

Ik wens je veel plezier met het maken van de opdrachten.

Peter Achten
P.Achten@cs.ru.nl

Inhoud

1 Start **1**
- 1.1 Gebruik van de Clean IDE 1
- 1.2 Zoeken in de Clean IDE 2

2 Functies **5**
- 2.1 Notaties 5
- 2.2 Vindt de redex 6
- 2.3 Matching strings 6
- 2.4 Functie types 7
- 2.5 Priemgetallen bepalen 8
- 2.6 Priemfactoren 8
- 2.7 Cijfers optellen 9
- 2.8 Oh dennenboom 9
- 2.9 Overloading en `(,)` 9
- 2.10 Kogelbanen 10
- 2.11 99 Bottles of beer 11

3 Lijsten **13**
- 3.1 Lijst-notaties 13
- 3.2 Types en waarden 14
- 3.3 De eersten zullen de laatsten zijn 14
- 3.4 Lijsten plakken met ++ 15
- 3.5 Lijsten plakken met flatten 15
- 3.6 Lijst-generatoren 16
- 3.7 ZF-notaties 16
- 3.8 Lijst-generatoren, deel II 17
- 3.9 !! en ?? 17
- 3.10 Lijst comprehensions en `removeAt` 17
- 3.11 Lijst comprehensions en `updateAt` 18
- 3.12 Fibonacci 18
  - 3.12.1 De Fibonacci functie 18
  - 3.12.2 Somlijsten 18
  - 3.12.3 De Fibonacci reeks 18
- 3.13 Beginstukken 19
- 3.14 Fragmenten 19
- 3.15 Deellijsten 20
- 3.16 Permutaties 20

3.17 Partities 20
3.18 Lijsten en sorteren 21
3.19 Element frequentie 21
3.20 De laatsten zullen de eersten zijn 22
3.21 Cijferreeks 23
3.22 Overloading en `[]` 23
3.23 Tekst uitlijnen 23
3.24 Woordzoeker 24
3.25 Wisselgeld 25
3.26 Domino stenen 25
3.27 Zakkenvuller 26
3.28 Zakkenvuller, deel II 27
3.29 Boer zoekt vrouw 28
3.30 Braille 28
3.31 Modern English spelling 30

4 Eenvoudige Algebraïsche Types en Records **33**
4.1 Notaties 33
4.2 Overloading en records 33
4.3 Kaarten 34
4.4 Romeinse getallen 34
4.5 Boids 35
4.5.1 Optioneel: Boids 3D 36

5 Hogere-Orde Functies **39**
5.1 Notaties 39
5.2 Met of zonder curry 39
5.3 Functie compositie 40
5.4 Argumenten flippen 40
5.5 Twice 40
5.6 Ellips omtrek 41
5.7 Elementen groeperen 41
5.8 Woordenlijst 42
5.9 Woordfrequentie 42
5.9.1 Optioneel: Woordfrequenties tonen 42
5.10 Origami 43
5.11 Nou en of 43
5.12 Een betere `StdBool` 43
5.13 Romeinse Getallen, deel II 45
5.14 `scan` en `iterate` 45
5.15 Taylor reeksen 45
5.16 `seq` en `seqList` 46
5.17 Database queries 46
5.18 Pseudo-random getallen 47
5.19 `return` en `bind` 48

6 Data abstractie **49**
6.1 Breuken . . . 49
6.2 Getallen . . . 50
6.3 Uniforme verzamelingen . . . 50
6.4 Tijd . . . 51
6.5 Stack . . . 51
6.6 Gesorteerde Lijst . . . 52
6.7 Associatie Lijst . . . 52
6.8 Tekstcompositie . . . 53
6.9 Stack, deel II . . . 54
6.10 Eenvoudige graphics . . . 55

7 Console en File I/O **57**
7.1 Echo . . . 57
7.2 Oh dennenboom, deel II . . . 57
7.3 Zinspelingen . . . 58
7.4 Mastermind . . . 58
7.5 Pesten . . . 59
7.6 Een module voor file I/O . . . 60
7.7 Monadische Console en File I/O . . . 61
7.8 Woordenlijst, deel II . . . 62
7.9 Woordfrequentie, deel II . . . 62
7.10 Gesorteerde files en bomen . . . 62
7.11 Boggle . . . 63
7.12 Galgje . . . 64
7.13 One-Time Pad (OTP) encryptie . . . 64
7.14 Huffman codering . . . 66

8 Boomstructuren **69**
8.1 Binaire bomen . . . 69
8.2 Bomen tonen . . . 70
8.2.1 Afdrukken met inspringen . . . 70
8.2.2 2D afdrukken . . . 71
8.3 Binaire zoekbomen . . . 71
8.4 Bomen aflopen . . . 72
8.5 Map en fold over bomen . . . 73
8.6 Gegeneraliseerde bomen . . . 74
8.7 Gegeneraliseerde bomen tonen . . . 75
8.7.1 Afdrukken met inspringen . . . 75
8.7.2 2D afdrukken . . . 76
8.8 Stambomen . . . 76
8.9 Stambomen afdrukken . . . 77
8.10 AVL-bomen . . . 77
8.11 Propositie-logica . . . 78
8.11.1 Grondtermen . . . 78
8.11.2 Variabelen . . . 79
8.12 Drie-waardige propositie-logica . . . 80
8.13 Expressies refactoren . . . 80
8.13.1 Expressies afdrukken . . . 81

8.13.2 Vrije variabelen . . . 81
8.13.3 Ongebruikte variabelen . . . 82
8.13.4 Evaluator . . . 82
8.14 Een λ-reducer . . . 82
8.14.1 Printen . . . 83
8.14.2 Normaalvorm . . . 84
8.14.3 Variabelen . . . 84
8.14.4 Verse variabele . . . 84
8.14.5 Substitutie . . . 84
8.14.6 Reductie . . . 85
8.14.7 Strategie . . . 85
8.14.8 Herschrijven tot normaalvorm . . . 85
8.15 Vier op een rij . . . 86
8.16 Nim . . . 86
8.17 Othello . . . 86
8.18 Blokus . . . 87
8.19 Abalone . . . 88
8.20 `map` en type constructor classes . . . 89
8.21 RefactorXX, monadisch . . . 90
8.21.1 Monadische evaluator . . . 90
8.21.2 Monadische values . . . 90

9 Correctheidsbewijzen **91**
9.1 `map` en `o` . . . 92
9.2 `init` en `take` . . . 92
9.3 Peano aritmiek . . . 93
9.4 `map`, `flatten` en `++` . . . 93
9.4.1 `map` en `++` . . . 94
9.4.2 `map` en `flatten` . . . 94
9.5 Lijsten en bomen . . . 94
9.6 `subs` en `map` . . . 95

10 Dynamics **97**
10.1 Notaties . . . 97
10.2 Getallen raden . . . 97
10.3 Heterogene verzamelingen . . . 98
10.4 Een IKS interpreter . . . 99

11 SoccerFun **103**
11.1 Training: rondjes lopen . . . 103
11.2 Training: slalommen . . . 103
11.3 Training: overspelen . . . 103
11.4 Training: vrij overspelen . . . 104
11.5 Training: keeper . . . 104
11.6 Eindopdracht . . . 104

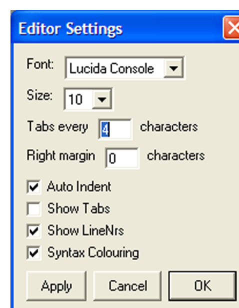
12 TOP **107**
12.1 Galgje . . . 108

Hoofdstuk 1

Start

De opdrachten in dit hoofdstuk helpen je op weg in het gebruiken van de Clean IDE. Ze maken je bekend met de diverse *windows* (bijvoorbeeld project, editor, types, errors) die gebruikt worden in de programmeeromgeving. Verder zijn er een aantal erg handige navigatiefuncties zoals het vinden van modulen, functies en types.
Tips:

- De gebruikershandleiding van de Clean IDE vind je met behulp van het commando “Help:Help:UserManual.pdf”.
- In de standaard-instellingen van de Clean IDE worden regelnummers niet weergegeven. Het is handig dat wel te doen in verband met foutmeldingen. Dat gaat met behulp van het commando “Defaults:Window Settings:Editor Settings...”. Vink het vakje “Show LineNrs” aan.



1.1 Gebruik van de Clean IDE

Main module:	*Start.icl*
Environment:	*StdEnv*

Open de module `Start.icl`. Maak er een Clean project van. Dit gaat met het commando “File:New Project...”. Bevestig de door de Clean IDE voorgestelde project file (“Start.prj”) in dezelfde directory. De inhoud van `Start.icl` is:

```
module Start

import StdEnv

Start = expr0

expr0 = "Hello World!"
```

1. Kies “Project:Bring Up To Date (Ctrl+U)”. Wat gebeurt er?
2. Kies “Project:Run”. Wat gebeurt er?

3. Kies “Module:Module Options…”. De dialoog “Module Options” verschijnt. Selecteer het vakje naast “Inferred Types”. Druk de “OK” knop in. Kies “File:Save Start.icl”. Kies “Project:Update and Run (Ctrl+R)”. Wat gebeurt er?

Voeg nu telkens een van onderstaande `expr`i `=` …definities toe, en vervang `Start = expr`i. Voer daarna “Project: Update and Run (Ctrl+R)” uit. Vertel telkens wat er gebeurt.

```
expr1  = "Hello␣" +++ "World!"
expr2  = 5
expr3  = 5.5
expr4  = 5 + 5.5
expr5  = [1..10]
expr6  = (expr1, expr2, expr3, expr5)
expr7  = [expr1, expr2, expr3, expr5]
expr8  = [1, 3 .. 10]
expr9  = ['a' .. 'z']
expr10 = ['a', 'c' .. 'z']
expr11 = ['Hello␣World!']
```

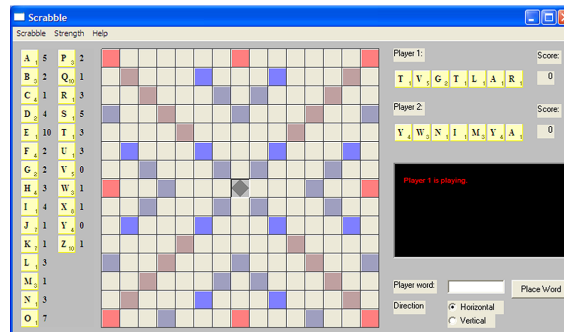
- Als een van de definities een typeringsfout geeft, kun je hem daarna in commentaar veranderen door er `//` vooraan te zetten (á la C).
- Je kunt snel naar de lokatie van de foutmelding in de broncode springen met behulp van het commando “Search:Goto Next Error… (Ctrl+E)”, of door de foutmelding te dubbel-klikken.

1.2 Zoeken in de Clean IDE

In de Clean IDE kun je op veel verschillende manieren zoeken naar elementen uit je programma's: (definitie en implementatie) modulen, types, functies, klassen, enzovoorts. Start de Clean IDE en open het project `scrabble.prj` dat je kunt vinden in

`{Application}\Examples\ObjectIO Examples\scrabble`.

Breng de applicatie *up to date* (Ctrl+U). Na *launchen* (Ctrl+R) zou je het *window* uit figuur 1.1 moeten zien.



Figuur 1.1: Het start venster van scrabble.

Modulen: het Projectvenster

Het *projectvenster* toont, nadat een applicatie eenmaal *up to date* is gebracht, uit welke modulen een project is opgebouwd. De hoofdmodule vind je *altijd* bovenaan. Modulen staan in *directories*. Je kunt de modulen uit een directory tonen en sluiten door te

dubbelklikken op de directorynaam. Door te dubbelklikken op een module-naam open je de bijbehorende *definitie* module, en met *shift*-dubbelklik de *implementatie* module. Je kunt hiermee snel alle definitie en implementatie modulen vinden.

Als je liever de implementatie module opent zonder de *shift* toets ingedrukt te houden, dan kun je dit instellen m.b.v. de project venster opties (commando "Defaults:-Window Settings:Project Window...").

Je kunt snel van implementatie (`icl`) en definitie (`dcl`) module wisselen m.b.v. Ctrl+/.

Opdracht: Activeer het projectvenster van scrabble en open de volgende modulen:

1. Open de hoofdmodule van scrabble.
2. Open de module `language.icl` van scrabble.
3. Open de module `StdOverloaded.dcl` uit `StdEnv`.

Definities in project

Elke Clean module bestaat uit type definities en functie definities. Modulen kunnen gebruik maken van andere definities door de definitie module te *importeren* waarin die definitie *geëxporteerd* wordt. In een module kun je alleen die definities gebruiken die je daar zelf maakt of importeert. Definities kun je op verschillende manieren zoeken.

Binnen een module kun je snel een definitie vinden m.b.v. Ctrl+`: dit opent een *pop up menu* waarin alle (top-level) definities alfabetisch geordend getoond worden. De Clean IDE zet de cursor aan het begin van de regel van de geselecteerde definitie.

Het commando Ctrl+= laat je een naam (*identifier*) opzoeken in alle definitie modulen (Find Definition), alle implementatie modulen (Find Implementation) of alle voorkomens (Find Identifiers). De zoektocht kun je beperken tot alle geïmporteerde modulen (Search in Imported Files), binnen het project (Search in Project) of alle gebruikte directories (Search in Paths).

Als je met de cursor al een naam in zijn geheel geselecteerd hebt (bijv. met dubbelklik), dan kun je m.b.v. Ctrl+L de definitie module vinden waarin de naam geëxporteerd wordt, en m.b.v. Ctrl+Alt+L de implementatie module.

Opdracht: Voer de volgende zoekopdrachten uit:

1. Zoek de definitie module waar de functie `isEven` geëxporteerd wordt. Welke module is dat?
2. Activeer de hoofdmodule van scrabble. Zoek alle voorkomens van de identifier `isEven` binnen de geïmporteerde modulen. Hoeveel zijn dat er?
3. Activeer de module `language.icl` van scrabble. Zoek opnieuw alle voorkomens van `isEven` binnen de geïmporteerde modulen. Hoeveel zijn dat er?

Projectpaden wijzigen

Het scrabble spel staat in de distributie ingesteld op de Engelse taal. Er is ook een Nederlandstalige module die in een andere directory staat. Veel voorkomende combinaties van projectpaden zijn voorgedefinieerd in *environments* (een overzicht van alle environments vind je in het menu Environment). Je kunt snel van projectpaden wisselen door een andere environment te kiezen. Je kunt de projectpaden wijzigen in de *Project Options* dialoog (Project:Project Options...).

Opdracht: Open de *Project Options* dialoog en selecteer de *radio button* Project Paths. Verwijder nu {Project}\Engels en voeg {Project}\Nederlands toe. Sluit de dialoog. Hercompileer de hele applicatie m.b.v. Ctrl+**Shift**+U. Als alles goed gegaan is, heb je nu een Nederlandstalige scrabble.

Hoofdstuk 2

Functies

De opdrachten in dit hoofdstuk zijn oefeningen in het lezen van functionele expressies en er mee kunnen rekenen (herschrijven). Een goede vaardigheid in het rekenen met expressies is van essentiëel belang in het redeneren over functies en het maken van correctheidsbewijzen (dit komt uitgebreid aan bod in hoofdstuk 9). Verder bevat dit hoofdstuk oefeningen met een aantal klassieke standaardfuncties (functie compositie, currying, flip) waarmee je bestaande functies kunt combineren tot nieuwe functies. De overige oefeningen zijn bedoeld om te zien hoe je recursieve functies uitdrukt in een functionele stijl.

2.1 Notaties

Main module:	*NotatieFuncties.icl*
Environment:	*StdEnv*

Beschrijf voor elk van de volgende Clean functies wat ze betekenen. Controleer je antwoord door middel van een aantal `Start` regels die de onderstaande functies aanroepen met geschikte argumenten, indien van toepassing.

```
f1          :: Int
f1          = 1 + 5

f2          :: Int
f2          = (+) 1 5

f3          :: Int Int -> Int
f3 m n
| m < n     = m
| otherwise = n

f4          :: String Int -> String
f4 s n
| n <= 0    = ""
| otherwise = s +++ f4 s (n-1)
```

```
f5          :: Int Int -> Int
f5 x 0      = x
f5 x y      = f5 y (x rem y)

f6          :: (Int,Int) -> Int
f6 x        = fst x + snd x

f7          :: (a,b) -> (b,a)
f7 (a,b)    = (b,a)

f8          :: (a,a) -> (a,a)
f8 x        = f7 (f7 x)
```

2.2 Vindt de redex

Main module:	*VindtDeRedex.icl*
Environment:	*StdEnv*

Bereken het resultaat van onderstaande expressies door middel van herschrijven. Plaats eerst alle haakjes `(` en `)` in de expressie die de prioriteit van de operaties correct weergeven. Voer daarna de berekeningen uit zoals voorgedaan op het college: begin vanuit `Start = e`i, begin elke stap op een nieuwe regel, en onderstreep welke redex je uitrekent.

```
e1 = 42
e2 = 1 + 125 * 8 / 10 - 59
e3 = not True || True && False
e4 =  1 + 2  ==  6 - 3
e5 = "1␣+␣2" == "6␣-␣3"
e6 = "1111␣+␣2222" == "1111" +++ "␣+␣" +++ "2222"
```

2.3 Matching strings

Main module:	*MatchStrings.icl*
Environment:	*StdEnv*

De standaard-omgeving van Clean bevat functies voor het werken met teksten van type `String`. In deze opdracht mag je, wat deze *string* operaties betreft, alleen gebruik maken van de volgende functies:

- `size`: levert het aantal tekens van de gegeven `String` op.
  Voorbeeld: `size ""` levert 0 op;
  Voorbeeld: `size "0123456789"` levert 10 op.
- `.[]`: levert de `Char` waarde op van de gegeven `Int` index-positie in een string. De index posities van een niet-lege `String` s lopen van 0 tot en met `size` $s - 1$. De *lege* string `""` heeft geen valide index posities.
  Voorbeeld: `"0123456789".[4]` levert `'4'` op.
  Voorbeeld: `"0123456789".[-1]` levert een *Run Time Error: index out of range* op.
  Voorbeeld: `"0123456789".[10]` levert dezelfde foutmelding op.
- `%`: neemt een *slice* van een `String` waarde.
  Voorbeeld: `"0123456789" % (0,2)` levert `"012"` op.
  Voorbeeld: `"Madam,␣I'm␣Adam" % (7,12)` levert `"I'm␣Ad"` op.

1. Schrijf de twee functies `head :: String -> Char` en `tail :: String -> String` die van een niet-lege `String` respectievelijk het eerste element en de resterende `String` opleveren. Als het argument een lege string is, dan dient een foutmelding getoond te worden.

   Voorbeeld: `head "Madam,␣I'm␣Adam"` = `'M'`.
   Voorbeeld: `tail "Madam,␣I'm␣Adam"` = `"adam,␣I'm␣Adam"`.

2. Schrijf een *recursieve*[1] functie `is_gelijk` die de gelijkheid van twee `String` argumenten bepaalt. Er moet dus voor ieder mogelijk paar s_1, s_2 van type `String` gelden: `is_gelijk` s_1 s_2 = s_1 `==` s_2.

3. Schrijf een recursieve functie `is_deelstring` die twee `String` argumenten krijgt en een `Bool` waarde oplevert. Het resultaat is alleen `True` als het eerste argument een *deelstring* is van het tweede argument.
   Voorbeeld: `is_deelstring "there" "Is␣there␣anybody␣out␣there?"`[2] levert `True` op, immers: `"Is there anybody out there?"`.
   Voorbeeld: `is_deelstring "there" "Just␣for␣the␣record"`[3] levert `False` op omdat tussen `the` en `re` een spatie staat.

4. Schrijf een recursieve functie `is_deel` die twee `String` argumenten krijgt en een `Bool` waarde oplevert. Het resultaat is alleen `True` als de tekens van het eerste argument in dezelfde volgorde voorkomen in het tweede argument, waarbij willekeurige tekens uit het tweede argument overgeslagen mogen worden.
   Voorbeeld: `is_deel "there" "Just␣for␣the␣record"` levert nu wel `True` op omdat de spatie overgeslagen wordt.
   Voorbeeld: `is_deel "she␣and␣her" "Is␣there␣anybody␣in␣there?"` levert `True` op, immers: `"Is there anybody in there?"`.
   Voorbeeld: `is_deel "There␣there"`[4] `"Is␣there␣anybody␣in␣there?"` levert `False` op omdat `T` geen deel uitmaakt van de tweede tekst.

5. Schrijf een recursieve functie `is_match` die twee `String` argumenten krijgt (*patroon* en *bron*) en een `Bool` waarde oplevert. De functie bepaalt of *patroon* gepast kan worden (*matching*) op *bron*. In *patroon* kunnen *wildcard* tekens voorkomen:

   - `.`: deze past met ieder eerstvolgende teken in *bron*;
   - `*`: deze past met 0 of meer achtereenvolgende tekens in *bron*.

   Alle overige tekens in *patroon* moeten exact overeenkomen met de corresponderende tekens in *bron*.
   Voorbeeld: `is_match "*.here*.here*." "Is␣there␣anybody␣in␣there?"` levert `True` op.
   Voorbeeld: `is_match ".here.here." "Is␣there␣anybody␣in␣there?"` levert `False` op.

2.4 Functie types

Main module:	*FunctieTypes.icl*
Environment:	*StdEnv*

Hieronder staan een aantal functie-types. Geef van ieder type een functie implementatie. Deze functie mag niet triviaal zijn (bijv. `abort` of `undef`).

```
f1 :: Int
f2 :: Int -> Bool
f3 :: Real -> Real
```

[1] Als in deze opdrachtenbundel om een *recursieve* functie gevraagd wordt, dan kan het zijn dat je een hulpfunctie moet maken met meer argumenten die het probleem recursief oplost.
[2] Pink Floyd – The Wall (1979)
[3] Marillion – Clutching at straws (1987)
[4] Radiohead – Hail to the thief (2003)

```
f4 :: Real Real -> Real
f5 :: Real -> (Real -> Real)
f6 :: (Real Real -> Real)
f7 :: (Real -> Real) Real -> Real
```

2.5 Priemgetallen bepalen(*gebruikt in 2.6*)

Main module:	*Priemgetal.icl*
Environment:	*StdEnv*

Schrijf de recursieve functie `isPriemgetal :: Int -> Bool` die bepaalt of zijn argument een priemgetal is. Een priemgetal is een positief geheel getal dat geheel deelbaar is door precies twee positieve, gehele, *verschillende* getallen, n.l. zichzelf en 1 (vandaar dat 1 zelf geen priemgetal is). Test je functie met `Start = [x \\ x <- [1 .. 1000] | isPriemgetal x]`. Dit levert alle priemgetallen in het bereik 1 tot en met 1000 op. De uitkomst hoort te zijn:

> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

Opmerking: Gebruik voor integer deling met rest de functie `rem`. De functienaam `mod` bestaat weliswaar, maar is niet gedefinieerd voor gehele getallen.

2.6 Priemfactoren

Main module:	*Priemfactoren.icl*
Environment:	*StdEnv*

Ieder geheel getal $x > 1$ kan op een canonieke wijze ontbonden worden in priemfactoren, d.w.z.: je kunt een reeks priemgetallen $p_0 \dots p_k$ $(k \geq 0)$ vinden waarbij $p_i < p_j$ voor iedere $i < j$ en evenveel positieve getallen $n_0 \dots n_k$ zodanig dat:

$$x = p_0^{n_0} \cdot \ldots \cdot p_k^{n_k} = \prod_{i=0}^{k} p_i^{n_i}.$$

Voorbeelden:	36	=	$2^2 \cdot 3^2$
	52	=	$2^2 \cdot 13$
	133	=	$7 \cdot 19$
	123456789	=	$3^2 \cdot 3607 \cdot 3803$

Schrijf de functie `priemfactoren :: Int -> String` die een geheel getal ontbindt in priemfactoren. De gevonden priemfactoren worden in de resultaat-string achter elkaar geplakt.

Voorbeelden:

`priemfactoren 52`	=	`"2*2*13"`
`priemfactoren 36`	=	`"2*2*3*3"`
`priemfactoren 133`	=	`"7*19"`
`priemfactoren 123456789`	=	`"3*3*3607*3803"`

Maak gebruik van de functie `isPriemgetal` uit opdracht 2.5.

De volgende operaties op `String`s en `Int`s zijn handig: s_1 `+++` s_2 plakt `String` s_2 achter `String` s_1; `toString` n maakt van `Int` n een `String` waarde.

2.7 Cijfers optellen

Main module:	*Cijfersom.icl*
Environment:	*StdEnv*

Schrijf de recursieve functie `cijfersom :: Int -> Int` die de cijfers van een positief getal bij elkaar optelt.

Voorbeeld: `cijfersom 9876543` = 9+8+7+6+5+4+3 = 42.

Voorbeeld: `cijfersom 1000` = 1+0+0+0 = 1.

2.8 Oh dennenboom*(gebruikt in 7.2)*

Main module:	*OhDennenboom.icl*
Environment:	*StdEnv*

1. Schrijf de functie `driehoek` die tenminste één integer argument n krijgt. Deze functie 'tekent' een driehoek zoals hiernaast is afgebeeld met $n = 5$. Het 'tekenen' vindt plaats door een `String` op te leveren waarvan elke 'rij' eindigt met een *newline* teken.

```
    *
   ***
  *****
 *******
*********
```

2. Schrijf de functie `dennenboom` die tenminste één integer argument n krijgt. Deze functie 'tekent' een dennenboom zoals hiernaast afgebeeld met $n = 4$. Voor iedere deeldriehoek dient deze functie gebruik te maken van de hierboven gemaakte `driehoek` functie.

```
   *
   *
  ***
   *
  ***
 *****
   *
  ***
 *****
*******
```

De volgende operaties op `String`s en `Char`s zijn handig: s_1 `+++` s_2 plakt `String` s_2 achter `String` s_1; `toString` c maakt van `Char` c een `String` waarde; `'\n'` is het *newline* teken. Om een uitvoer zonder `String` haakjes te krijgen, selecteer je in de Clean IDE dialoog "Project:Project Options..." de optie "Basic Values Only".

2.9 Overloading en `(,)`

Main module:	*TupleOverloading.icl*
Environment:	*StdEnv*

In module `StdTuple` is gelijkheid gedefinieerd voor *tuples* `(,)` en *triplets* `(,,)` mits er

gelijkheid is voor de componenten. Definieer op analoge wijze instanties van tuples `(,)` en *triplets* `(,,)` voor de overloaded functies `+`, `-`, `*`, `/`, `~`, `zero` en `one`. De implementaties dienen de volgende eigenschappen te hebben:

```
∀a : zero + a = a = a + zero
∀a : a - zero = a = ~ (zero - a)
∀a :  one * a = a = a * one
∀a : zero * a = zero = a * zero
∀a :  a / one = a
∀a :  ~ (~ a) = a
```

Dit kun je eenvoudig testen door een `Start` regel te maken die voor een aantal waarden van type `(,)` en `(,,)` de volgende `test` functie aanroept:

```
test a = ( zero + a == a    && a    == a + zero
         , a - zero == a    && a    == ~ (zero - a)
         ,  one * a == a    && a    == a * one
         , zero * a == zero && zero == a * zero
         ,  a / one == a
         ,  ~ (~ a) == a
         )
```

2.10 Kogelbanen

Main module:	*Kogelbaan.icl*
Environment:	*StdEnv*

Een object, bijvoorbeeld een bal, dat vanaf een plat vlak weggeschoten wordt onder een hoek $\theta_0 (0 < \theta_0 \leq \frac{\pi}{2})$[5] en beginsnelheid $v_0 (0 < v_0)$, beschrijft een boogvormige baan onder invloed van de zwaartekracht[6]. Als we effecten als luchtweerstand en wind negeren, dan is de baan van de bal een curve die weergegeven kan worden in een rechtopstaand plat vlak, waarbij de x-as de grondafstand weergeeft, en de y-as de hoogte van de bal.

Omdat we de luchtweerstand verwaarlozen, is de snelheid van de bal in de x-richting ($v_x(t)$) constant. De snelheid van de bal in de y-richting ($v_y(t)$) hangt af van de tijd t en de valversnelling $g = 9.81 \frac{m}{s^2}$:

$$\begin{aligned} v_x(t) &= v_0 \cdot cos(\theta_0) \\ v_y(t) &= v_0 \cdot sin(\theta_0) - g \cdot t \end{aligned}$$

De afstand $x(t)$ en hoogte $y(t)$ van de bal op tijdstip t kun je als volgt berekenen:

$$\begin{aligned} x(t) &= v_0 \cdot cos(\theta_0) \cdot t \\ y(t) &= v_0 \cdot sin(\theta_0) \cdot t - \frac{1}{2} \cdot g \cdot t^2 \end{aligned}$$

De hoogte h, uitgedrukt in de afstand x is:

$$h(x) = tan(\theta_0) \cdot x - \frac{g}{2 \cdot (v_0 \cdot cos(\theta_0))} \cdot x^2$$

[5]Hoeken in radialen.

[6]Bron: Halliday, Resnick. *Physics, Parts I and II, combined edition*, Wiley International Edition, 1966, ISBN 0 471 34524 5

De bal bereikt het hoogste punt op tijdstip t_{maxy}:

$$t_{maxy} = \frac{v_0 \cdot sin(\theta_0)}{g}$$

De maximum hoogte die op dat tijdstip bereikt wordt vind je dus door invullen van t_{maxy} in $y(t)$. De bal doet er net zo lang over om het hoogste punt te bereiken als weer op de grond aan te komen. De bal is dus $t_2 = 2 \cdot t_{maxy}$ seconden in de lucht. De afstand die de bal aflegt vind je door invullen van t_2 in $x(t)$. De snelheid die de bal heeft bij neerkomen vind je door invullen van t_2 in $v_x(t)$ en $v_y(t)$. De gecombineerde waarde is:

$$v_1 = \sqrt{v_x(t_2)^2 + v_y(t_2)^2}$$

Functies Schrijf de bovenstaande functies v_x, v_y, x, y en h nu in Clean met de respectievelijke namen `v_x`, `v_y`, `x_at`, `y_at` en `h`. Voeg indien nodig extra argumenten toe aan deze functies.

Beste hoek Schrijf een recursieve functie `beste_hoek` die, gegeven een beginsnelheid v_0, de hoek $\theta \in \{\frac{1}{100}\pi, \frac{2}{100}\pi \ldots \frac{50}{100}\pi\}$ kiest zodanig dat de bal die met snelheid v_0 en hoek θ gespeeld wordt, de *grootste* afstand aflegt.

Experimenteer met verschillende waarden voor v_0. Komt hier hetzelfde resultaat uit?

2.11 99 Bottles of beer

Main module:	*BottlesOfBeer.icl*
Environment:	*StdEnv*

Implementeer de functie `bottles_of_beer` die het volgende rijm produceert:

99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down and pass it around, 97 bottles of beer on the wall.

⋮

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

Probeer een zo kort mogelijk programma te schrijven.
De volgende operaties op `String`s en `Char`s zijn handig: s_1 `+++` s_2 plakt `String` s_2 achter `String` s_1; `toString` c maakt van `Char` c een `String` waarde; `toString` n maakt van `Int` n een `String` waarde; `'\n'` is het *newline* teken (type `Char`), `""` is de *lege string* (type `String`). Om een uitvoer zonder `String` haakjes te krijgen, selecteer je in de Clean IDE dialoog "Project:Project Options..." de optie "Basic Values Only".

Hoofdstuk 3

Lijsten

Lijsten zijn een van de meest gebruikte recursieve data structuren in functionele programmeertalen. De opdrachten in dit hoofdstuk laten je oefenen met het creëren van lijsten, het manipuleren van lijsten, en het werken met zogenaamde *lijst comprehensions*. Lijsten kunnen optimaal gebruik maken van het *luie* karakter van Clean; dit stelt je in staat om te werken met potentiëel oneindig lange lijsten, en het tijdelijk in de wacht zetten van berekeningen die je wellicht in de toekomst nodig hebt.

3.1 Lijst-notaties

Main module:	*NotatieLijsten.icl*
Environment:	—

In deze opgave oefen je het lezen van lijsten. Hieronder staan een aantal lijsten genoteerd. Doe voor elke lijst het volgende:

- leg in eigen woorden uit welke lijst er staat,
- geef aan hoeveel elementen de lijst heeft,
- geef de kortste notatie die gelijkwaardig is aan de gegeven lijst (dit kan dezelfde expressie zijn),
- geef het resultaat van de standaard functies `hd`, `tl`, `init` en `last` op de lijst. Deze functies vind je in de module *StdList.icl*.

Dit zijn de lijsten waar het om gaat:

```
l1    = []
l2    = [1000:[]]
l3    = [[]]
l4    = [[]:[]]
l5    = ['a':['b':['c':['d':['e':[]]]]]]
l6    = [[1],[],[2,3]:[[4,5,6]]]
l7    = [[[1,2],[3,4]],[],[[],[],[5,6]]]
```

3.2 Types en waarden

Main module:	*LijstTypes.icl*
Environment:	*StdEnv*

In deze opgave krijg je een aantal lijst-types. Geef van ieder type een expressie die hetzelfde type heeft. Deze expressie mag niet triviaal zijn (bijv. `abort`, `undef`, `[]`).

```
e1 :: [Int]
e1 = ...

e2 :: [Bool]
e2 = ...

e3 :: [[Int]]
e3 = ...

e4 :: [[[Real]]]
e4 = ...

e5 :: [Int Int -> Int]
e5 = ...
```

3.3 De eersten zullen de laatsten zijn

Main module:	*EersteOfLaatste.icl*
Environment:	*StdEnv*

Eerste twee en laatste twee Schrijf de functies `eerste2` en `laatste2` die beide een *lijst* als argument krijgen en een *lijst* opleveren. Het type van beide functies is dus: `[a] -> [a]`. De functie `eerste2` levert de eerste twee elementen van het argument op, en `laatste2` levert de laatste twee elementen van het argument op. Genereer een foutmelding m.b.v. de functie `abort` als de argument lijsten te kort zijn.

Voorbeelden:

```
eerste2  [42]          =  "Lijst is te kort."
eerste2  [1,2,3,4,5]   =  [1,2]
laatste2 [42]          =  "Lijst is te kort."
laatste2 [1,2,3,4,5]   =  [4,5]
```

Rekenen In deze opgave bereken je op papier het resultaat van een aantal functies met behulp van herschrijven. De functies die we gebruiken zijn `eerste2` en `laatste2` of komen uit *StdEnv*. Schrijf deze berekeningen op dezelfde wijze als op het college: begin vanuit `Start`, begin elke stap op een nieuwe regel, en onderstreep wat je berekent. Voorbeelden zijn te vinden op de college-slides.

1. `Start = hd (hd (hd [[[1,2,3],[4]],[[5],[6]]]))`
2. `Start = hd (tl [1,2,3,4,5])`
3. `Start = eerste2 [[1],[],[2,3],[4,5,6]]`
4. `Start = laatste2 [[1],[],[2,3],[4,5,6]]`

Eerste *n* en laatste *n* Schrijf de recursieve functies `eersten` en `laatsten` die beide een *getal* en een *lijst* als argument krijgen en een *lijst* opleveren. Het type van beide

functies is dus: `Int [a] -> [a]`. De functie `eersten n lijst` levert de eerste `n` elementen van `lijst` op, en `laatsten n lijst` levert de laatste `n` elementen van `lijst` op. Genereer een foutmelding m.b.v. de functie `abort` als de argument lijsten te kort zijn.

Voorbeelden:

```
eersten  4 [42]         =  "Lijst is te kort."
eersten  4 [1,2,3,4,5]  =  [1,2,3,4]
laatsten 4 [42]         =  "Lijst is te kort."
laatsten 4 [1,2,3,4,5]  =  [2,3,4,5]
```

Eigenschappen Maak de volgende beweringen af:

$$\begin{array}{lll}
\forall 0 \leq n, \quad xs :: [a] : \texttt{eersten}\ n\ (\texttt{eersten}\ n\ xs) & = & \ldots \\
\forall 0 \leq n, \quad xs :: [a] : \texttt{eersten}\ n\ (\texttt{laatsten}\ n\ xs) & = & \ldots \\
\forall 0 \leq n, \quad xs :: [a] : \texttt{laatsten}\ n\ (\texttt{eersten}\ n\ xs) & = & \ldots \\
\forall 0 \leq n, \quad xs :: [a] : \texttt{laatsten}\ n\ (\texttt{laatsten}\ n\ xs) & = & \ldots \\
\forall 0 \leq m \leq n, xs :: [a] : \texttt{eersten}\ m\ (\texttt{eersten}\ n\ xs) & = & \ldots \\
\forall 0 \leq m \leq n, xs :: [a] : \texttt{length}\ (\texttt{eersten}\ m\ xs) & ? & \texttt{length}\ (\texttt{eersten}\ n\ xs)
\end{array}$$

3.4 Lijsten plakken met ++

Main module:	*ReducerenVanLijsten.icl*
Environment:	*StdEnv*

Lijst concatenatie (`++`) is als volgt gedefinieerd (zie module `StdList.icl`):

```
(++) infixr 5 :: [a] [a] -> [a]
(++) [hd : tl] list = [hd : tl ++ list]
(++) [ ]       list = list
```

Bereken van de onderstaande expressies de uitkomst door herschrijven van de `++` operator. De waarden van $\texttt{x}_i$ zijn irrelevant.

1. `[] ++ []`
2. `[] ++ [x0,x1] ++ []`
3. `[[]] ++ [x0,x1]`
4. `[x0,x1] ++ [[]]`
5. `[] ++ ([x0] ++ ([x1, x2] ++ [x3, x4, x5]))`
6. `(([] ++ [x0]) ++ [x1, x2]) ++ [x3, x4, x5]`

3.5 Lijsten plakken met flatten

Main module:	*Flatten.icl*
Environment:	*StdEnv*

De `flatten` operatie op lijsten is als volgt gedefinieerd (zie module `StdList.icl`):

```
flatten :: [[a]] -> [a]
flatten [xs : xss] = xs ++ flatten xss
flatten [ ]        = [ ]
```

Bereken van de onderstaande expressies de uitkomst door herschrijven van de `flatten` en `++` functies. De waarden van $\texttt{x}_i$ zijn irrelevant.

1. `flatten [[x0,x1,x2], [x3,x4], [x5], []]`
2. `flatten [[[x0,x1,x2], [x3,x4]], [], [[x5], []]]`
3. `flatten (flatten [[[x0,x1,x2], [x3,x4]], [], [[x5], []]])`

3.6 Lijst-generatoren

Main module:	*LijstGenerator.icl*
Environment:	*StdEnv*

Schrijf de volgende functies die lijsten genereren:

- `allemaal` x berekent de oneindig lange lijst $[x, x, x \ldots]$.
  Voorbeeld: `allemaal 42` = `[42, 42, 42, 42, ...]`.
- `vanaf` x berekent de oneindig lange lijst $[x,\ x$+`one`, x+`one`+`one`, $\ldots]$.
  Voorbeeld: `vanaf -10` = `[-10, -9, -8, -7 ...]`.
  Voorbeeld: `vanaf 'a'` = `['a','b','c','d' ...]`.
- `vanaf_met_stap` $x\ z$ berekent de oneindig lange lijst $[x, x + z, x + z + z, \ldots]$.
  Voorbeeld: `vanaf_met_stap 0 -2` = `[0, -2, -4, -6 ...]`.
- `vanaf_tot` $x\ y$ (met $x \leq y$) berekent de lijst $[x, x$+`one`$, x$+`one`+`one`$, \ldots y']$ zodanig dat $y' \leq y$ en y'+`one` $> y$.
  `vanaf_tot` $x\ y$ (met $x > y$) levert de lege lijst op.
  Voorbeeld: `vanaf_tot 'a' 'z'` = `['abcdefghijklmnopqrstuvwxyz']`.
  Voorbeeld: `vanaf_tot 0.5 7.0` = `[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5]`.
  Voorbeeld: `vanaf_tot 3 3` = `[3]`.
  Voorbeeld: `vanaf_tot 3 0` = `[]`.
- `vanaf_tot_met_stap` $x\ y\ z$ berekent de lijst $[x, x + z, x + z + z, \ldots, y']$ zodanig dat als $x \leq y \wedge z >$`zero`: $y' \leq y \wedge y' + z > y$ en als $x \geq y \wedge z <$`zero`: $y' \geq y \wedge y' + z < y$.
  Voor alle andere gevallen is het resultaat de lege lijst.
  Voorbeeld: `vanaf_tot_met_stap 0 -10 -2` = `[0, -2, -4, -6, -8, -10]`.
  Voorbeeld: `vanaf_tot_met_stap 0 -10 -3` = `[0, -3, -6, -9]`.
  Voorbeeld: `vanaf_tot_met_stap 0 -10 -42` = `[0]`.
  Voorbeeld: `vanaf_tot_met_stap 0 -10 2` = `[]`.

Maak de types van deze functies zo algemeen mogelijk.

3.7 ZF-notaties

Main module:	*NotatieZF.icl*
Environment:	*StdEnv*

Leg van elk van de onderstaande functies uit wat ze uitrekenen en geef tevens het meest algemene type van elk van de functies:

```
g1 as bs = [(a,b) \\ a  <- as,  b <- bs]
g2 as bs = [(a,b) \\ a  <- as & b <- bs]
g3 as bs = [(a,b) \\ a  <- as,  b <- bs | a <> b]
g4 as bs = [  a   \\ a  <- as,  b <- bs | a == b]
g5   xss = [  x   \\ xs <- xss, x <- xs]
g6 a  xs = [  i   \\ i  <- [0 ..] & x <- xs | a == x]
```

3.8 Lijst-generatoren, deel II

Main module:	*LijstGenerator2.icl*
Environment:	*StdEnv*

Schrijf dezelfde functies als in opdracht 3.6, maar maak nu uitsluitend gebruik van *lijst-comprehensions*. Hierdoor kan het voorkomen dat de types van deze functies aangepast moeten worden. Verifieer dat de nieuwe implementaties dezelfde uitkomsten opleveren als degene uit opdracht 3.6.

3.9 !! en ??

Main module:	*ZFZoek.icl*
Environment:	*StdEnv*

De operator `(!!)` **infixl** `9 :: ![a] !Int -> a` uit `StdList` selecteert het i-de element uit een lijst xs na aanroep van xs`!!`i (geteld vanaf 0). Als xs te weinig elementen bevat, dan wordt er een *run-time error* gegenereerd.

Implementeer m.b.v. een lijst comprehension de operator

```
(??) infixl 9 :: ![a] !a -> Int | Eq a
```

die de index opzoekt van een waarde x uit een lijst xs, indien deze bestaat. Als deze waarde niet bestaat, dan moet `??` de waarde -1 opleveren. Als xs een lijst is met een element x, dan moet dus gelden: xs`!!`$(xs$`??`$x) = x$. Als x geen element is van xs, dan levert xs`!!`$(xs$`??`$x)$ een *run-time error* op.

Voorbeelden:

```
[1,2,3,4,5,6]   ?? 3   = 2
[1,2,3,4,5,6]   ?? 10  = -1
['Hello␣world'] ?? 'o' = 4
```

3.10 Lijst comprehensions en `removeAt`

Main module:	*ZFRemoveAt.icl*
Environment:	*StdEnv*

Beschouw de functie `removeAt` uit de standaard module `StdList`. Schrijf met behulp van een lijst comprehension de functie `removeAt2` die dezelfde argumenten krijgt en hetzelfde resultaat oplevert, m.a.w.:

$$\forall n :: \texttt{Int}, \forall xs :: [\texttt{a}] : \texttt{removeAt}\ n\ xs = \texttt{removeAt2}\ n\ xs.$$

3.11 Lijst comprehensions en `updateAt`

Main module:	*ZFUpdateAt.icl*
Environment:	*StdEnv*

Beschouw de functie `updateAt` uit de standaard module `StdList`. Schrijf met behulp van een lijst comprehension de functie `updateAt2` die dezelfde argumenten krijgt en hetzelfde resultaat oplevert, m.a.w.:

$$\forall n :: \texttt{Int}, \forall x :: \texttt{a}, \forall xs :: [\texttt{a}] : \texttt{updateAt}\ n\ x\ xs = \texttt{updateAt2}\ n\ x\ xs.$$

3.12 Fibonacci

Main module:	*Fibonacci.icl*
Environment:	*StdEnv*

3.12.1 De Fibonacci functie

De *Fibonacci* getallenreeks $F = F_0, F_1, F_2 ...$ is gedefinieerd door:

$$F = \begin{cases} F_0 & = & 1 \\ F_1 & = & 1 \\ F_n & = & F_{n-1} + F_{n-2} \end{cases}$$

Schrijf de direct-recursieve functie `fibonacci` op bovenstaande wijze met betekenis: `fibonacci n` $= F_n$. Voor welke waarden is deze functie gedefinieerd?

Test je functie met opeenvolgende waardes:

```
N     = 46
Start = [fibonacci i \\ i <- [1 .. N]]
```

Beschrijf en verklaar het executiegedrag van het programma.

3.12.2 Somlijsten

Schrijf de functie `somLijsten` met betekenis:

$$\texttt{somLijsten}\ [a_0, ..., a_n]\ [b_0, ..., b_n] = [a_0 + b_0, ..., a_n + b_n].$$

Kies zelf wat de functie betekent voor lijst-argumenten van ongelijke lengte en motiveer je keuze. Test je functie met representatieve invoerwaarden. Welke zijn dat en geef de uitkomst weer.

3.12.3 De Fibonacci reeks(*gebruikt in 3.18*)

Een andere manier om tegen de fibonacci reeks F aan te kijken is:

```
 [1,1,2,3, 5, 8,13,21,34 ...]                    F
 [1,2,3,5, 8,13,21,34,55 ...]               (tl F)
--------------------------------------------------
 [2,3,5,8,13,21,34,55,89 ...]   somLijsten F (tl F)
```

De eerste N fibonacci getallen kun je dan als volgt berekenen:

```
Start    = eersten N fibs
where
    fibs = [1,1:somLijsten fibs (tl fibs)]
```

Beschrijf en verklaar het executiegedrag van het programma. Vergelijk het executiegedrag met het gedrag van het programma uit 3.12.1.

3.13 Beginstukken(*gebruikt in 7.3*)

Main module:	*Firsts.icl*
Environment:	*StdEnv*

De functie `firsts` berekent van een lijst `xs` alle beginstukken:

$$\texttt{firsts}\ [\,] = [[\,]]$$
$$\texttt{firsts}\ [x_0 \ldots x_n] = [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2] \ldots [x_0 \ldots x_n]] (\text{voor } n \geq 0)$$

1. Geef het meest algemene type van `firsts`.
2. Als `xs` een lijst met n elementen is, hoeveel elementen heeft dan de lijst (`firsts xs`)?
3. Implementeer deze functie. Het resultaat van deze functie moet *dezelfde* elementen bevatten als hierboven beschreven. De *volgorde* van de elementen in het resultaat mag afwijken.

3.14 Fragmenten(*gebruikt in 7.3*)

Main module:	*Frags.icl*
Environment:	*StdEnv*

De functie `frags` berekent van een lijst `xs` alle *fragmenten*. Een fragment van `xs` is een deellijst van `xs` zodanig dat het enkel achtereenvolgende elementen van `xs` bevat. Het aantal en de positie van het eerste element is willekeurig (maar uiteraard begrensd door `xs`). Merk op dat zowel `[]` als `xs` zelf een fragment zijn van `xs`. Bijvoorbeeld:

$$\begin{aligned}
\texttt{frags}\ [\,] &= [[\,]] \\
\texttt{frags}\ [x_0 \ldots x_n] &= [[\,] \\
&\quad , [x_0] \ldots [x_n] \\
&\quad , [x_0, x_1], [x_1, x_2], \ldots [x_{n-1}, x_n] \\
&\quad \vdots \\
&\quad , [x_0 \ldots x_{n-1}], [x_1 \ldots x_n] \\
&\quad , [x_0 \ldots x_n] \\
&\quad] \qquad (\text{voor } n \geq 0)
\end{aligned}$$

1. Geef het meest algemene type van `frags`.
2. Als `xs` een lijst met n elementen is, hoeveel elementen heeft dan de lijst (`frags xs`)?
3. Implementeer deze functie. Het resultaat van deze functie moet *dezelfde* elementen bevatten als hierboven beschreven. De *volgorde* van de elementen in het resultaat mag afwijken.

3.15 Deellijsten(*gebruikt in 7.3*)

Main module:	*Subs.icl*
Environment:	*StdEnv*

De functie `subs` berekent van een lijst `xs` alle *deellijsten*. Een deellijst van `xs` is een lijst met dezelfde elementen in dezelfde volgorde, maar waarvan een willekeurig aantal elementen niet opgenomen zijn. Merk op dat zowel `[]` en `xs` een deellijst van `xs` zijn. Bijvoorbeeld:

$$\begin{array}{lcl}
\texttt{subs}\ [\,] & = & [[\,]] \\
\texttt{subs}\ [x_0] & = & [[\,],[x_0]] \\
\texttt{subs}\ [x_0,x_1] & = & [[\,],[x_0],[x_1],[x_0,x_1]] \\
\texttt{subs}\ [x_0,x_1,x_2] & = & [[\,],[x_0],[x_1],[x_2],[x_0,x_1],[x_0,x_2],[x_1,x_2],[x_0,x_1,x_2]]
\end{array}$$

1. Geef het meest algemene type van `subs`.
2. Als `xs` een lijst met n elementen is, hoeveel elementen heeft dan de lijst (`subs xs`)?
3. Implementeer deze functie. Het resultaat van deze functie moet *dezelfde* elementen bevatten als hierboven beschreven. De *volgorde* van de elementen in het resultaat mag afwijken.

3.16 Permutaties(*gebruikt in 3.18,7.3*)

Main module:	*Perms.icl*
Environment:	*StdEnv*

De functie `perms` berekent van een lijst `xs` alle *permutaties*. Een permutatie van `xs` is een lijst die dezelfde elementen bevat, maar mogelijk in een andere volgorde. Merk op dat `xs` een permutatie van `xs` is. Bijvoorbeeld:

$$\begin{array}{lcl}
\texttt{perms}\ [\,] & = & [[\,]] \\
\texttt{perms}\ [x_0] & = & [[x_0]] \\
\texttt{perms}\ [x_0,x_1] & = & [[x_0,x_1],[x_1,x_0]] \\
\texttt{perms}\ [x_0,x_1,x_2] & = & [[x_0,x_1,x_2],[x_0,x_2,x_1] \\
 & & ,[x_1,x_0,x_2],[x_1,x_2,x_0] \\
 & & ,[x_2,x_0,x_1],[x_2,x_1,x_0] \\
 & &]
\end{array}$$

1. Geef het meest algemene type van `perms`.
2. Als `xs` een lijst met n elementen is, hoeveel elementen heeft dan de lijst (`perms xs`)?
3. Implementeer deze functie. Het resultaat van deze functie moet *dezelfde* elementen bevatten als hierboven beschreven. De *volgorde* van de elementen in het resultaat mag afwijken.

3.17 Partities

Main module:	*Partities.icl*
Environment:	*StdEnv*

De functie `partities` berekent van een lijst `xs` alle *partities*. Een partitie van `xs` is een

lijst $[A_1 \ldots A_n]$ $(n \geq 0)$, zodanig dat A_1 ++ … ++ A_n = `xs` en iedere A_i is niet leeg. De lege lijst heeft alleen zichzelf als mogelijke partitie. Bijvoorbeeld:

`partities` $[\,]$	$=$	$[[[\,]]]$
`partities` $[x_0]$	$=$	$[[[x_0]]]$
`partities` $[x_0, x_1]$	$=$	$[[[x_0], [x_1]], [[x_0, x_1]]]$
`partities` $[x_0, x_1, x_2]$	$=$	$[[[x_0], [x_1], [x_2]], [[x_0], [x_1, x_2]], [[x_0, x_1], [x_2]], [[x_0, x_1, x_2]]]$

1. Geef het meest algemene type van `partities`.
2. Als `xs` een lijst met n elementen is, hoeveel elementen heeft dan (`partities xs`)?
3. Implementeer deze functie. Het resultaat van deze functie moet *dezelfde* elementen bevatten als hierboven beschreven. De *volgorde* van de elementen in het resultaat mag afwijken.

3.18 Lijsten en sorteren

Main module:	*ZFSort.icl*
Environment:	*StdEnv*

Een lijst $L = [l_0 \ldots l_n]$ is gesorteerd als voor iedere $0 \leq i < j \leq n$: $l_i \leq l_j$.

Test op ordening Maak gebruik van het idee uit opdracht 3.12.3 om een functie `is_gesorteerd` te maken die een lijst krijgt en test of deze gesorteerd is. De functie `is_gesorteerd` mag de lijst alleen met een lijst comprehension inspecteren.

Lijsten sorteren De lijst L' is de gesorteerde lijst van L als L' een *permutatie* is van L *en* gesorteerd is. Schrijf de functie `sorteer` die een lijst krijgt en deze sorteert volgens deze definitie. Maak gebruik van de functie `perms` uit opdracht 3.16, de functie `is_gesorteerd` die je hierboven gemaakt hebt, en alleen een lijst comprehension.

Complexiteit Wat is de orde complexiteit van deze manier van sorteren in termen van de lengte van de lijst? Is het een goede manier om te sorteren?

3.19 Element frequentie(*gebruikt in 5.9*)

Main module:	*Frequentielijst.icl*
Environment:	*Object IO*

Schrijf de functie `frequentielijst :: [a] -> [(a,Int)] | == a` die van een lijst de *frequentietabel* berekent. Een frequentietabel is een lijst van *tuples*. Het eerste element van de tuple komt uit de originele lijst, en het tweede element is het aantal voorkomens van dat element in de lijst.

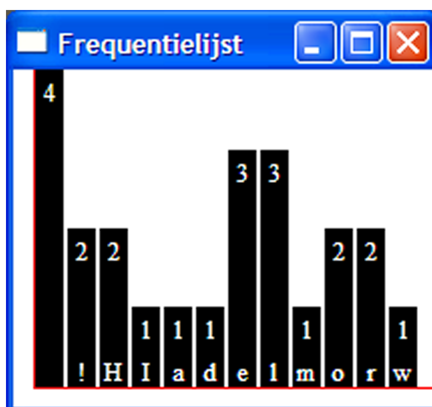
Voorbeeld:

```
frequentielijst ['Hello␣world!␣Here␣I␣am!']
   =
[('!',2),('m',1),('a',1),('␣',4),('I',1),('e',3)
,('r',2),('H',2),('d',1),('l',3),('o',2),('w',1)]
```

Je kunt het resultaat visualiseren met behulp van de module `FrequentielijstGUI` die een functie `toonFrequentielijst` heeft.
Voorbeeld:

```
import FrequentielijstGUI

Start world = toonFrequentielijst (sort (frequentielijst tekst)) world
where
   tekst    = ['Hello␣world!␣Here␣I␣am!']
```



Figuur 3.1: The frequentielijst van 'Hello world! Here I am!'

3.20 De laatsten zullen de eersten zijn

Main module:	*EersteIsLaatste.icl*
Environment:	*StdEnv*

De functie `last` die van een lijst het laatste element oplevert kun je op verschillende manieren schrijven. Een eerste manier is middels een recursieve functie:

```
last1 :: [a] -> a
last1 [x]    = x
last1 [_:xs] = last1 xs
```

Een andere manier is door te zeggen dat het laatste element van een lijst hetzelfde is als het eerste element van de omgekeerde lijst (de functie `reverse` vind je in module `StdList`):

```
last2 :: ([a] -> a)
last2 = hd o reverse
```

Rekenen Herschrijf de volgende expressies door iedere stap op een nieuwe regel te beginnen, en de redexen te onderstrepen die je herschrijft.

1. `Start = last1 [1,2,3,4]`
2. `Start = last2 [1,2,3,4]`

Functie-gelijkheid versus executiegedrag De functies `last1` en `last2` zijn *gelijk*, dat wil zeggen: met dezelfde argumenten leveren ze hetzelfde resultaat op. Toch zijn ze niet hetzelfde. Leg uit wat het verschil is, en gebruik de hierboven verkregen resultaten.

3.21 Cijferreeks

Main module:	*Cijferreeks.icl*
Environment:	*StdEnv*

Bestudeer de functie `intChars :: (Int -> [Char])` uit het diktaat, paragraaf 3.2.4 (*"Displaying a number as a list of characters"*). Schrijf in dezelfde stijl de inverse functie, `charsInt :: ([Char] -> Int)`, die een `Char` lijst krijgt waarvan alle elementen digits zijn, en die het getal uitrekent dat door de lijst gerepresenteerd wordt.
Voorbeeld: `charsInt ['3210']` = 3210.

3.22 Overloading en []

Main module:	*LijstOverloading.icl*
Environment:	*StdEnv*

In module `StdList` is gelijkheid gedefinieerd voor *lijsten* mits er gelijkheid is voor de componenten. Definieer op analoge wijze instanties voor `zero`, `one`, `+`, `-`, `*`, / en `~`. De implementaties dienen de volgende eigenschappen te hebben:

```
∀a : zero + a = a = a + zero
∀a : a - zero = a = ~ (zero - a)
∀a :  one * a = a = a * one
∀a :  a / one = a
∀a :  ~ (~ a) = a
```

Test dit op soortgelijke wijze als in opdracht 2.9.

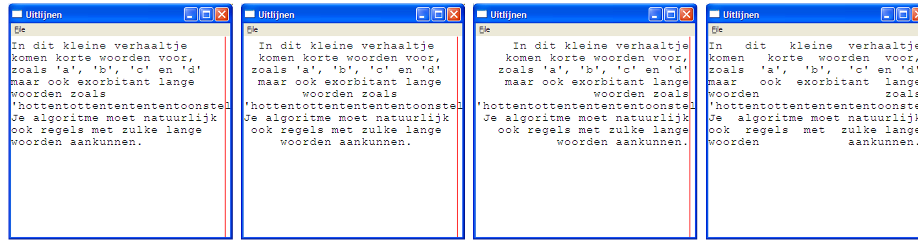
3.23 Tekst uitlijnen

Main module:	*Uitlijnen.icl*
Environment:	*Object IO*

In deze opdracht breid je de hoofdmodule `Uitlijnen` uit. Deze module opent een *window* waarin een tekst getoond wordt (zie figuur 3.2). Het is de bedoeling dat de tekst *uitgelijnd* wordt. Dit kan in vier verschillende modi, die met behulp van het type `UitlijnMode` dat synoniem is met `Char` uitgedrukt worden:

```
:: UitlijnMode :== Char
Links           = 'l'
Centreren       = 'c'
Rechts          = 'r'
Uitlijnen       = 'u'
```

Het programma biedt een *menu* aan waarin je de actieve uitlijnmode kunt kiezen zodat je eenvoudig je programma kunt testen. Bovendien stelt het programma je in staat om de lettergrootte te veranderen.



Figuur 3.2: De vier uitlijnmodi in actie: `Links`, `Centreren`, `Rechts` en `Uitlijnen`.

Het programma is bijna af: de implementatie van de uitlijnfunctie ontbreekt nog. Dat is de functie die je moet schrijven. Het type is:

```
::  Tekst           :== String
::  Regel           :== String
::  TekstBreedte    :== Int
::  LetterBreedte   :== Int

uitlijnen :: UitlijnMode LetterBreedte TekstBreedte Tekst -> [Regel]
```

Het eerste argument is de gewenste uitlijnmode. Het tweede argument geeft de breedte van iedere letter aan (het gaat hier om een zogenaamd *niet-proportioneel* font, ofwel alle tekens hebben dezelfde breedte). Het derde argument is de breedte van het *window* waarin de tekst getekend moet worden. Het vierde argument is de tekst die uitgelijnd dient te worden.

Merk op dat dit programma gebruik maakt van de GUI bibliotheek van Clean, Object I/O. Zet daarom de environment op *Object IO* in de Clean IDE nadat je een project gemaakt hebt.

3.24 Woordzoeker

Main module:	*Woordzoeker.icl*
Environment:	*StdEnv*

Woordzoekers zijn puzzels waarin letters in een $m \times n$ matrix staan. Gegeven is een woordenlijst die afgestreept moeten worden in de woordzoeker. Leestekens worden genegeerd. Elk woord komt precies éénmaal voor. Een woord kan horizontaal voorkomen (van links naar rechts en omgekeerd), verticaal (van boven naar beneden en omgekeerd) en schuin (van linksonder naar rechtsboven en omgekeerd en links-boven naar rechts-onder en omgekeerd). Vaak kan er na wegstrepen van de resterende letters een zin gemaakt worden. Het voorbeeld hieronder levert na wegstrepen de naam van een bekende programmeertaal op:

```
T  E  R  P          FEL
C  W  L  F          TWEE
L  A  E  E          PRET
A  L  N  E          ALP
```

Implementeer een algoritme dat gegeven een $m \times n$ matrix van letters en een woordenlijst alle woorden wegstreept zoals hierboven beschreven. Het algoritme dient de resterende letters van links naar rechts, boven naar beneden op te leveren.

3.25 Wisselgeld

Main module:	*Wisselgeld.icl*
Environment:	*StdEnv*

Schrijf in deze opdracht een programma voor een wisselgeld-automaat. De wisselgeld-automaat heeft van elke soort munt en briefgeld een oneindige hoeveelheid tot zijn beschikking. Hij mag echter niet meer dan een tevoren gegeven hoeveelheid k munten / briefgeld tegelijkertijd uitgeven. Schrijf een functie *wissel* die, gegeven het te wisselen bedrag, de beschikbare valuta, en het maximum aantal te gebruiken munten / briefgeld alle mogelijke oplossingen genereert.

Maak gebruik van de volgende type synoniemen en type van *wissel*:

```
:: Bedrag    :== Int      // een positief getal
:: Valuta    :== Int      // een positief getal
:: Valutas   :== [Valuta] // een niet-lege lijst
:: Munt      :== Int      // een positief getal
:: K         :== Int      // een positief getal
:: WisselGeld :== [Munt]

wissel :: Bedrag Valutas K -> [WisselGeld]
```

Voorbeelden: (de uitkomsten mogen in een andere volgorde opgeleverd worden)

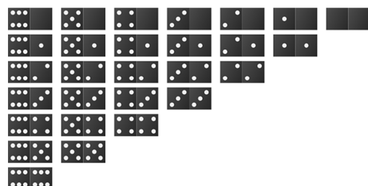
```
wissel 50 [100,50,20,10,5,1] 1 ⇒ [[50]]
wissel 50 [100,50,20,10,5,1] 2 ⇒ [[50]]
wissel 50 [100,50,20,10,5,1] 3 ⇒ [[50],[10,20,20]]
wissel 50 [100,50,20,10,5,1] 4 ⇒ [[50],[10,20,20],[5,5,20,20],[10,10,10,20]]
```

3.26 Domino stenen

Main module:	*Domino.icl*
Environment:	*StdEnv*

Het klassieke dominospel (de zogenaamde dubbel *zes* variant) bestaat uit 28 langwerpige stenen. Iedere steen bestaat uit twee helften die ieder van een aantal *ogen* is voorzien. De 28 stenen bevatten alle unieke combinaties van ogen van *nul* tot en met *zes* (zie ook figuur 3.3).

Er bestaan varianten met andere combinaties, zoals dubbel *negen*, dubbel *twaalf* en zelfs dubbel *achttien*. We noemen een dergelijke verzameling dominostenen een dubbel N spel (met N het maximum aantal ogen op één helft van een steen). Een dubbel N spel heeft $MAX = \frac{(N+2)(N+1)}{2}$ stenen. Met de stenen van een dubbel N spel kun je een 'slang' maken, d.w.z. een reeks achtereenvolgende dominostenen zodanig dat de korte zijden aan elkaar liggen, en het aantal ogen op beide aan elkaar liggende helften hetzelfde aantal ogen heeft (dus 0 tegen 0, 1 tegen 1, enz.). Een slang bevat geen 'lussen', zoals je tijdens een echt dominospel wel mag leggen.



Figuur 3.3: De dubbel zes domino-set.

Schrijf de functie `alle_slangen` die alle mogelijke slangen met lengte *MAX* van een dubbel *N* spel afdrukt.
Vb: voor een dubbel 1 spel zijn alle mogelijkheden van lengte 3:

```
[0:0] [0:1] [1:1]
[1:1] [1:0] [0:0]
```

Vb: voor een dubbel 2 spel zijn alle mogelijkheden van lengte 6:

```
[(0,2),(2,2),(2,1),(1,1),(1,0),(0,0)]
[(0,1),(1,1),(1,2),(2,2),(2,0),(0,0)]
[(1,1),(1,2),(2,2),(2,0),(0,0),(0,1)]
[(0,0),(0,2),(2,2),(2,1),(1,1),(1,0)]
[(2,2),(2,1),(1,1),(1,0),(0,0),(0,2)]
[(0,0),(0,1),(1,1),(1,2),(2,2),(2,0)]
[(1,2),(2,2),(2,0),(0,0),(0,1),(1,1)]
[(1,0),(0,0),(0,2),(2,2),(2,1),(1,1)]
[(2,2),(2,0),(0,0),(0,1),(1,1),(1,2)]
[(1,1),(1,0),(0,0),(0,2),(2,2),(2,1)]
[(2,1),(1,1),(1,0),(0,0),(0,2),(2,2)]
[(2,0),(0,0),(0,1),(1,1),(1,2),(2,2)]
```

3.27 Zakkenvuller(*gebruikt in 3.28*)

Main module:	*Zakkenvuller.icl*
Environment:	*StdEnv*

In veel zogenaamde *role playing games* bestuurt de speler een computerpersonage dat middels avonturen (*quests*) objecten kan verwerven. Een typisch assortiment bestaat uit edelstenen (klein, licht, en veel waard), toverdrankjes (klein, wat zwaarder en niet erg duur), wapentuig (zwaarden, bijlen, bogen, knotsen, enz.; groot, zwaar, en variërend van goedkoop tot heel duur), bepantsering (helmen, maliënkolders, handschoenen, laarzen, schouderstukken, enz.; variërend in alle factoren qua prijs, gewicht en afmeting), boeken (klein, wat zwaarder, en soms duur) en wat je zoal tegenkomt in dit soort werelden (betoverde uitrusting zoals ringen, wapens, bepantsering). Sommige objecten zijn *stackable*, ofwel stapelbaar. Dit wil zeggen dat ze een zekere omvang en gewicht hebben, maar dat er meerdere tegelijktijd dezelfde ruimte kunnen innemen. Dit leidt ertoe dat wel het aantal en totale gewicht toeneemt, maar niet de in beslag genomen ruimte. Typische voorwerpen die *stackable* zijn, zijn goudstukken, drankjes, ringen, e.d.

Het geheel aan voorwerpen dat een speler verwerft wordt ook wel *inventory* genoemd. Deze inventory heeft altijd een zekere *gelimiteerde* opslagcapaciteit. De limiet wordt

soms enkel bepaald door aantallen, soms door gewicht, soms door afmetingen, en meestal door een combinatie van deze factoren.

In het computerspel *Skyrim* hebben objecten een gewicht en een waarde. Alle objecten zijn *stackable*. De inventarisatie wordt enkel beperkt door het gezamenlijke gewicht: dat mag niet boven een gegeven waarde uitkomen.

De hoofdrolspeler wil zijn inventarisatie zo optimaal mogelijk vullen met de in het spel (al dan niet legitiem) verworven voorwerpen. Deze speler is niet gehecht aan welk voorwerp dan ook, en is van plan deze te verkopen zodat hij zijn eigen geldvoorraad kan aanvullen. Hij wil dus zijn inventarisatie zodanig vullen met voorwerpen dat deze de maximaal mogelijke waarde heeft; dat wil zeggen dat er geen andere vulling is waarvan de gezamenlijke waarde van voorwerpen groter is (mag wel hetzelfde zijn). Schrijf voor deze hoofdrolspeler een functie die de volgende argumenten krijgt:

1. een capaciteit van een maximaal gewicht.
2. een lijst van objecten die elk een waarde en een gewicht hebben.

De functie levert een vulling op die de maximale waarde heeft. Ontwerp geschikte datastructuren voor deze opdracht.

3.28 Zakkenvuller, deel II

Main module:	*Zakkenvuller2.icl*
Environment:	*StdEnv*

In opdracht 3.27 heb je een functie geschreven die een rugzak met items vult om een maximale waarde-opbrengst te krijgen. Het criterium om te stoppen met vullen was het maximale gewicht dat de rugzak kon dragen. In deze opdracht introduceren we een *ruimte* criterium, en dat doen we aan de hand van het computerspel *Neverwinter Nights*. Hierin hebben objecten naast een gewicht en waarde bovendien een *afmeting* (breedte en hoogte). De inventarisatie wordt in een soort rugzak gedragen, die eveneens een zekere breedte en hoogte heeft. In dit spel wordt de inventarisatie begrensd door de afmetingen: objecten kunnen elkaar niet overlappen. Verder zijn in *Neverwinter Nights* sommige type objecten *stackable*, en sommige niet. Typische voorbeelden van *stackable* objecten zijn drankjes, pijlen, ringen, enz. Typische niet-*stackable* objecten zijn zwaarden, harnassen, helmen, e.d.

Een bijzonder element in *Neverwinter Nights* is de zogenaamde *bag of holding*: dit is een rugzak die je in je inventory kunt plaatsen, en waarin je opnieuw voorwerpen kunt plaatsen. Elke *bag of holding* heeft een eigen afmeting, waarde en capaciteit (deze is normaal gesproken veel groter dan de afmeting van de *bag of holding* – bijv. een *bag of holding* met afmeting 2×2 eenheden, maar met een interne capaciteit van 30×25 eenheden). Het spreekt vanzelf dat dit zeer geliefde voorwerpen zijn voor avonturiers omdat deze de capaciteit van een speler significant uitbreiden.

Schrijf opnieuw een functie voor de avonturier om een optimale vulling van zijn rugzak met objecten te krijgen zodat verkoop van deze spullen maximale winst oplevert. Schrijf dus een functie die de volgende argumenten krijgt:

1. een rugzak met een zekere breedte en hoogte.
2. een lijst van objecten die elk een breedte en hoogte hebben, een waarde, en een gewicht. Een object behoort tot een soort, en de soort kan al dan niet *stackable* zijn.

De functie dient als resultaat een vulling van de rugzak op te leveren waarvoor geldt dat de gezamenlijke waarde van alle objecten in de rugzak maximaal is; d.w.z. er is geen andere vulling van de rugzak waarvan de gezamenlijke waarde *groter* is dan deze (hij mag dus wel gelijk zijn).

3.29 Boer zoekt vrouw

Main module:	*BoerZoektVrouw.icl*
Environment:	*StdEnv*

In het "stable marriage" probleem gaat het erom om een verzameling van N mannen op monogame wijze te koppelen aan N vrouwen zodanig dat dit 'stabiele' relaties oplevert. Iedere persoon heeft een volgorde van voorkeur aangegeven voor alle kandidaten van het andere geslacht middels een nummering van 1 t/m N, waarbij de beoordeling 1 de meest favoriete partner aangeeft, en N de minst favoriete partner. Een koppeling tussen alle mannen en alle vrouwen is stabiel als er geen man en vrouw te vinden zijn die liever bij elkaar zouden zijn dan met hun huidige, gekoppelde partner.

Als N een even aantal is, dan is er een algoritme dat een dergelijke stabiele koppeling berekent. Dit algoritme is in 1962 door David Gale and Lloyd Shapley correct bewezen[1]. Het is een iteratief algoritme dat telkens de volgende berekening uitvoert: vind een nog niet gekoppelde man en zoek de meest favoriete vrouw die hij nog geen huwelijksaanzoek gedaan heeft. De man loopt een blauwtje als de vrouw al gekoppeld is aan een man die hoger op haar favorietenlijstje staat. In ieder ander geval accepteert ze het aanbod. Mocht ze in dat geval al gekoppeld zijn met een man (die dus lager op haar lijstje staat), dan is deze man hierna niet meer gekoppeld. Dit herhaalt zich totdat alle mannen gekoppeld zijn.

Implementeer dit algoritme. Doe dit door de volgende functie te implementeren:

```
::  Nr :== Int    // 1..N

boer_zoekt_vrouw :: ([[Nr]],[[Nr]]) -> [(Nr,Nr)]
```

zodanig dat `(boer_zoekt_vrouw (voorkeuren_mannen,voorkeuren_vrouwen))` een 'stable marriage' oplossing berekent tussen de populatie mannen en vrouwen middels het Gale / Shapley algoritme mits de invoer aan de volgende voorwaarden voldoet:

1. de lengte `N` van `voorkeuren_mannen` is identiek aan de lengte van `voorkeuren_vrouwen`, en is bovendien een even waarde;
2. de voorkeuren van iedere man en iedere vrouw is een permutatie van `[1 .. N]`

De oplossing is een lijst van koppels (man,vrouw) die stabiel is.

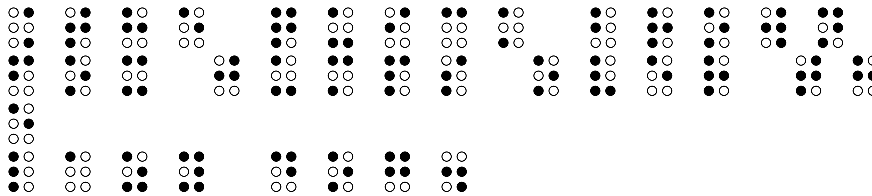
3.30 Braille

Main module:	*Braille.icl*
Environment:	*StdEnv*

In figuur 3.4 worden de tekens uit het Nederlandse *braille* schrift opgesomd (bron: MUZIEUM Nijmegen `www.muzieum.nl`). Het braille schrift is uitgevonden door Louis

[1] Bron: `http://en.wikipedia.org/wiki/Stable_marriage_problem`.

Braille (1809-1852) om blinden en slechtzienden in staat te stellen teksten te schrijven en te lezen. Het braille-schrift is een *reliëf-schrift*, d.w.z. dat in papier puntjes gedrukt worden die afgetast kunnen worden. Braille-tekens bestaan uit 6 punten, geordend in drie onder elkaar staande rijen van twee punttekens. De zwarte punten (•) vormen de 'puntjes' in het papier en de lege punten (◦) vormen geen reliëf in het papier. De cijfertekens 1 tot en met 9 en 0 zijn gelijk aan de lettertekens a tot en met j. Om ambiguïteit te voorkomen, moeten getallen voorafgegaan worden door het cijferreeks teken. De zin *"The quick brown fox jumps over the lazy dog."* ziet er in braille als volgt uit:



⠁	⠃	⠉	⠙	⠑	⠋	⠛	⠓	⠊	⠚	⠅	⠇	⠍	⠝
a	b	c	d	e	f	g	h	i	j	k	l	m	n

⠕	⠏	⠟	⠗	⠎	⠞	⠥	⠧	⠺	⠭	⠽	⠵
o	p	q	r	s	t	u	v	w	x	y	z

⠯	⠿	⠮	⠻	⠜	⠫	⠪	⠳	⠮	⠬
ç	é	è	ë	ä	ï	ö	ü	ß	&

⠁	⠃	⠉	⠙	⠑	⠋	⠛	⠓	⠊	⠚
1	2	3	4	5	6	7	8	9	0

⠂	⠆	⠒	⠲	⠢	⠖	⠶	⠦	⠴	⠔
,	;	:	.	?	!	( )	“	”	*

⠖	⠤	⠦	⠲	⠶	⠳
+	-	×	:	=	/

⠨	hoofdletterteken	⠘	permanent hoofdletterteken	⠠	herstelteken
⠸	cursiefteken	⠰	taalwisselingsteken	⠈	geen functie
⠜	versregelteken	⠐	nadrukteken	⠔	herhalingsteken
⠼	cijferreeks	⠤	koppelteken		

Figuur 3.4: Het Nederlands braille schrift.

Representatie: Schrijf de functie `toonBrailleTeken :: Char -> [(String,String,String)]` die een letter uit het alfabet uit figuur 3.4 omzet naar een braille-teken. Hierbij wordt een lege ruimte met '.' genoteerd, en een putje met '*'.

Voorbeeld: `toonBrailleTeken 'T' = [(".*","..",".*"),(".*","**","*.")]`. Deze bestaat uit twee elementen, vanwege het hoofdletterteken, gevolgd door de *t*-representatie.

Voorbeeld: `toonBrailleTeken 'a'` = `[("*.",". .",". .")]`

Voorbeeld: `toonBrailleTeken '0'` = `[(".*",".*","**"),(".*","**",". .")]`. Deze bestaat uit twee elementen, vanwege het cijferreeksteken, gevolgd door de 0-representatie.

Voorbeeld: `toonBrailleTeken '␣'` = `[("␣","␣","␣")]` (drie rijen van twee spaties). Spaties geef je dus weer als een 'lege' letter.

Voorbeeld: `toonBrailleTeken '@'` = `[]`. Deze is leeg, omdat `@` geen deel uitmaakt van het alfabet.

Schrijven: Schrijf de functie `toonBrailleTekst :: [Char] -> [(String,String,String)]` die een reeks letters uit het alfabet uit figuur 3.4 omzet in de drie onder elkaar staande regels van Braille-puntjes.

Voorbeeld:

```
toonBrailleTekst ['The␣quick␣brown'] = [(".*.**.*.␣␣***: :****.␣␣*.*.*: :***"
                                       ,". :****.*␣␣**: *: .. ..␣␣*.**.***.*"
                                       ,".**: .. ..␣␣*.**: .. *.␣␣, *.*: :**."
                                      )]
```

Lezen: Schrijf de functie `leesBrailleTekst :: [(String,String,String)] -> [Char]` die een braille-tekst zoals hierboven weergegeven omzet naar de leestekens.

Voorbeeld:

```
leesBrailleTekst [(".*.**.*.␣␣***: :****.␣␣*.*.*: :***"
                 ,". :****.*␣␣**: *: .. ..␣␣*.**.***.*"
                 ,".**: .. ..␣␣*.**: .. *.␣␣, *.*: :**."
                )] = ['The␣quick␣brown']
```

3.31 Modern English spelling

Main module:	*ModernEnglishSpelling.icl*
Environment:	*StdEnv*

Aan de Amerikaanse schrijver Mark Twain (1835-1910) wordt vaak het onderstaande satirische voorstel toegeschreven om de spelling van de Engelse taal te verbeteren:

A plan for the improvement of English spelling

For example, in Year 1 that useless letter c *would be dropped to be replased either by* k *or* s*, and likewise* x *would no longer be part of the alphabet. The only kase in which* c *would be retained would be the* ch *formation, which will be dealt with later.*

Year 2 might reform w *spelling, so that* which *and* one *would take the same konsonant, wile Year 3 might well abolish* y *replasing it with* i *and Iear 4 might fiks the* g/j *anomali wonse and for all.*

Jenerally, then, the improvement would kontinue iear bai iear with Iear 5 doing awai with useless double konsonants, and Iears 6-12 or so modifaiing vowlz and the rimeining voist and unvoist konsonants.

> *Bai Iear 15 or sou, it wud fainali bi posibl tu meik ius ov thi ridandant letez* c*,* y *and* x *– bai now jast a memori in the maindz ov ould doderez – tu riplais* ch*,* sh*, and* th *rispektivli.*
>
> *Fainali, xen, aafte sam 20 iers ov orxogrefkl riform, wi wud hev a lojikl, kohirnt speling in ius xrewawt xe Ingliy-spiking werld.* (Mark Twain)

Implementeer een programma dat een Engelse tekst (bijvoorbeeld *"ImproveEnglish-Spelling.txt"* in de *Practicum* folder) volgens de hierboven beschreven regels aanpast. Bovenstaande tekst laat nogal wat ruimte over voor interpretatie van de nieuwe spellingregels. Implementeer in plaats daarvan de volgende benadering van het voorstel ter verbetering van de Engelse spelling. Deze regels moeten *achtereenvolgens* worden toegepast (houd zelf rekening met hoofdletters):

1. 'ch' blijft ongewijzigd; 'c' gevolgd door 'e' of 'i' wordt 's'; 'c' wordt 'k' in alle andere gevallen.
2. 'x' wordt 'ks'
3. 'wh' wordt 'w'
4. 'y' wordt 'i'
5. 'g' gevolgd door 'e', 'i', 'y' wordt 'j' behalve bij *gear*, *get*, *give*, *gir*, *gift*.
6. alle dubbele medeklinkers worden enkele medeklinkers
7. 'ph' wordt 'f'
8. 'r' wordt verwijderd als deze gevolgd wordt door nul of meer *white-space* tekens gevolgd door een medeklinker.
9. 'ch' wordt vervangen door 'c'; 'sh' wordt vervangen door 'y'; 'th' wordt vervangen door 'x'.

Hoofdstuk 4

Eenvoudige Algebraïsche Types en Records

Algebraïsche types en *records* zijn hét instrument bij uitstek van functionele programmeertalen om nieuwe datastructuren te modelleren. Nieuwe datastructuren spelen een essentiële rol bij *overloading* dat immers type-gestuurd is. Met algebraïsche types introduceer je nieuwe constructoren in je programma's. Dit verhoogt in het algemeen de leesbaarheid en correctheid van je programma. Record types stellen je in staat collecties van waarden te manipuleren zonder dat je precies hoeft te weten waar de waarden in de collectie precies gedefiniëerd zijn.

4.1 Notaties

Main module:	*NotatieADT.icl*
Environment:	*StdEnv*

Hieronder staan een aantal algebraïsche types gedefinieerd. Geef van iedere definitie *drie verschillende* expressies (niet de triviale expressies `abort` en `undef`), indien mogelijk.

```
:: Dag      = Maandag | Dinsdag | Woensdag | Donderdag | Vrijdag | Zaterdag | Zondag
:: Nat      = Nat Int
:: Getal    = Geheel Nat | Decimaal Real
:: Functies = F0 Int | F1 (Int -> Int) | F2 (Int Int -> Int)
:: Void     = Void
```

4.2 Overloading en records

Main module:	*VectorOverloading.icl*
Environment:	*StdEnv*

Definieer het volgende record type voor twee-dimensionale vectoren:

```
:: Vector2 a = {x0 :: a, x1 :: a}
```

Implementeer voor dit nieuwe type instanties voor de overloaded klassen `==`, `zero`, `one`, `+`, `-`, `*`, / en ~ op soortgelijke wijze als in opdracht 2.9.

4.3 Kaarten(gebruikt in 7.5)

Main module:	*Kaart.icl*
Environment:	*StdEnv*

Een standaard kaartspel bestaat uit 52 kaarten. Iedere kaart heeft een *kleur* en een *waarde*. De kleuren zijn *hart* (♡), *ruit* (♢), *schop* (♠) en *klaver* (♣). De mogelijke waarden zijn de getallen 2 tot en met 10 en *boer*, *dame*, *heer* en *aas*. Implementeer de onderstaande onderdelen en maak zoveel mogelijk gebruik van algebraïsche data types en record types.

1. **Representatie** Ontwerp geschikte datastructuren om het kaartspel mee te representeren. De types die een speelkaart, kleur en waarde representeren moeten respectievelijk `Kaart`, `Kleur` en `Waarde` heten.

2. **Gelijkheid van kaarten** Schrijf de `Kaart` instantie voor de overloaded functie `==`.

3. **Printen en parseren** Schrijf voor je nieuwe type de instanties voor de overloaded functies `toString` en `fromString`. Voor de `fromString` instantie dient te gelden dat deze de inverse is van de `toString` instantie.

   $\forall k$ `:: Kaart` : `fromString (toString` k`)` $= k$.

   Kun je ook eisen dat voor iedere `String` s geldt:

   let `k :: Kaart; k = fromString` s **in** `toString k` $= s$?

4. **Kaartspel** Schrijf de functie `kaartspel :: [Kaart]` die een complete lijst van alle kaarten uit een kaartspel genereert. Streef naar een zo kort mogelijke definitie.

5. **Sorteren naar waarde** Schrijf de functie `sorteer_naar_waarde` die een kaartspel sorteert in oplopende volgorde. Eerst wordt geordend op waarde, daarna op kleur in de volgorde zoals hierboven aangegeven. Streef naar een zo kort mogelijke definitie.

6. **Sorteren naar kleur** Schrijf de functie `sorteer_naar_kleur` die een kaartspel sorteert in oplopende volgorde. Eerst wordt geordend op kleur in de volgorde zoals hierboven aangegeven, en daarna op waarde. Streef naar een zo kort mogelijke definitie.

4.4 Romeinse getallen(gebruikt in 5.13)

Main module:	*RomeinsGetal.icl*
Environment:	*StdEnv*

De klassieke romeinen noteerden getallen met de volgende symbolen: M, D, C, L, X, V en I met respectievelijke waarden 1000, 500, 100, 50, 10, 5 en 1. Een positief getal wordt genoteerd door deze symbolen achter elkaar op te schrijven, waarbij de symbolen met de grootste waarde links staan, en de symbolen met de laagste waarde rechts. De waarde van dat getal is de som van de waarden van de individuele symbolen. Negatieve getallen en nul kunnen niet gerepresenteerd worden.

Voorbeeld: MDCLXVI $= 1000 + 500 + 100 + 50 + 10 + 5 + 1 = 1666$.
Voorbeeld: MMVIII $= 1000 + 1000 + 5 + 1 + 1 + 1 = 2008$.

Voor veel voorkomende patronen werden *afkortingen* gebruikt:

DCCCC	⇒	CM	LXXXX	⇒	XC	VIIII	⇒	IX
CCCC	⇒	CD	XXXX	⇒	XL	IIII	⇒	IV

Voorbeeld: CMXCIX $= (500 + 4 \cdot 100) + (50 + 4 \cdot 10) + (5 + 4 \cdot 1) = 999$.
Voorbeeld: CDXLIV $= 4 \cdot 100 + 4 \cdot 10 + 4 \cdot 1 = 444$.

Laat nu het algebraïsche type `RD` romeinse 'digits' representeren, en het type `Roman` een romeins genoteerd getal.

```
:: RD    = M | D | C | L | X | V | I
:: Roman = Roman [RD]
```

Roman → Int Schrijf een instantie voor de klasse `toInt` voor `Roman` die aan de hierboven gegeven regels voldoet.

Int → Roman Schrijf een instantie voor de klasse `fromInt` voor `Roman` die een *positieve* `Int` waarde omzet naar de *kortste* romeinse representatie.

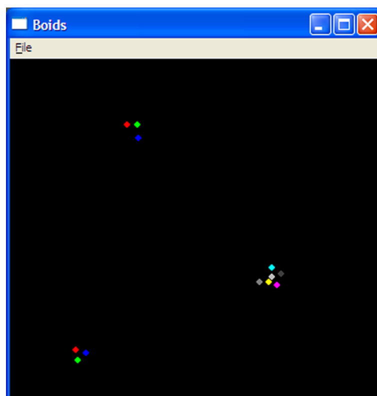
4.5 Boids

Main module:	*Boid.icl*
Environment:	*Object IO*

Boids zijn in 1986 door Craig Reynolds bedacht als voorbeeld van *artificial life* om het bewegingsgedrag van dieren in een groep te beschrijven, zoals een vlucht vogels of een school vissen. Zie voor meer achtergrondinformatie hierover
`http://www.red3d.com/cwr/boids/`.
Een simulatie bestaat uit een verzameling individuen, *boid* genaamd. Iedere boid heeft een *positie* en een *snelheid*. Je kunt de simulatie zowel ruimtelijk als in een vlak laten plaatsvinden. In deze opgave gaan we uit van een vlak. Een positie is dus een punt (rx, ry), waarbij $0 \leq rx \leq 1$ en $0 \leq ry \leq 1$ en een snelheid is een vector (vx, vy). We nemen aan dat we functies *pos* en *vel* hebben die van een boid de positie en snelheid opleveren. Iedere boid houdt zich aan drie regels die ieder een deel van zijn volgende snelheid, en dus ook volgende positie, bepalen. Een andere uitleg over deze regels kun je ook vinden op
`http://www.vergenet.net/~conrad/boids/pseudocode.html`.
Deze regels zijn:

1. *Boids bewegen richting het middelpunt van nabije boids.* Stel dat we voor boid B de nieuwe deelsnelheid v_1 willen berekenen. Laat dan $B_1 \ldots B_{n_1}$ de boids zijn die zich op korte afstand d_1 van B bevinden (experimenteer zelf met waarden voor d_1). De nieuwe deelsnelheid v_1 voor B is:

$$v_1 = \frac{\frac{\sum_{i=1}^{n_1} pos(B_i)}{n_1} - pos(B)}{100}$$



Figuur 4.1: Het Boids simulatie programma

2. *Boids bewaren (een kleine) afstand tot objecten, inclusief andere boids.* Stel dat we voor boid B de nieuwe deelsnelheid v_2 willen berekenen. Laat dan $B_1 \ldots B_{n_2}$ de boids zijn die zich op korte afstand d_2 van B bevinden (experimenteer zelf met waarden voor d_2). De nieuwe deelsnelheid v_2 voor B is:

$$v_2 = \frac{(0,0) - \sum_{i=1}^{n_2}(pos(B_i) - pos(B))}{8}$$

3. *Boids passen hun snelheid aan aan de snelheid van boids in hun omgeving.* Stel dat we voor boid B de nieuwe deelsnelheid v_3 willen berekenen. Laat dan $B_1 \ldots B_{n_3}$ de boids zijn die zich op korte afstand d_3 van B bevinden (experimenteer zelf met waarden voor d_3). De nieuwe deelsnelheid v_3 voor B is:

$$v_3 = \frac{\frac{\sum_{i=1}^{n_3} vel(B_i)}{n_3} - vel(B)}{8}$$

De totale nieuwe snelheid voor boid B is $v = vel(B) + \sum_{i=1}^{3} v_i$, en de nieuwe positie is $pos(B) + v$. Door deze berekening voor iedere boid B uit een verzameling boids toe te passen, kun je telkens de nieuwe snelheden en posities berekenen.

Schrijf een functie `simulatie` die een lijst van boids krijgt, en die een nieuwe lijst oplevert waarin de snelheden en posities van alle boids zijn aangepast zoals hierboven beschreven. In het voorgegeven deel van de uitwerking wordt een venster geopend waarin je met behulp van de muis nieuwe boids kunt plaatsen in het venster. Deze boids hebben initiëel snelheid $(0,0)$. Het raamwerk tekent boids op eenvoudige manier als gekleurde cirkels (zie figuur 4.1). Maak in je implementatie zoveel mogelijk gebruik van records en/of algebraïsche data types (om boids, snelheden en posities weer te geven), overloading (optellen en aftrekken van snelheden en posities) en lijst comprehensions.

4.5.1 Optioneel: Boids 3D

Voer dezelfde opdracht uit, maar laat de boids zich nu in een ruimte bewegen. Een positie is dan een punt (rx, ry, rz) $(0 \leq rz \leq 1)$ en een snelheid een vector (vx, vy, vz).

Pas het tekenen van boids aan zodanig dat hogere boids groter en later getekend worden dan lagere boids. Je kunt ook overwegen om de kleuren aan te passen aan de hand van de hoogte van de boid: lagere boids krijgen een kleurverschuiving richting zwart.

Hoofdstuk 5

Hogere-Orde Functies

In eerdere hoofdstukken hebben we al functies gebruikt die uit functies nieuwe functies maken (zoals `o` en `flip`). *Currying* staat je toe functies op minder argumenten toe te passen dan de ariteit toestaat (bijvoorbeeld `((+) 1)`). Dit zijn allemaal voorbeelden van *hogere-orde* functies. Hogere-orde functies zijn essentiële gereedschappen voor de functionele programmeermethode omdat je er nieuwe *programmeerpatronen* mee kunt definiëren.

5.1 Notaties

Main module:	*NotatieHOF.icl*
Environment:	*StdEnv*

Hieronder staan een aantal functies. Leid van iedere functie het meest algemene type af en leg uit wat de functie doet.

```
f1 a b       = a b
f2 a b c     = a c (b c)
f3 a b       = a (a b)
f4 a b c     = [x \\ x <- [b .. c] | a x]
f5 a b (c,d) = (a c,b d)
f6           = f5
f7 "-"       = -
f7 "+"       = +
f7 "*"       = *
f7 "/"       = /
```

5.2 Met of zonder curry

Main module:	*MetOfZonderCurry.icl*
Environment:	*StdEnv*

Bestudeer de `curry` en `uncurry` functies uit de module `StdTuple`.

1. Wat doen deze functies?

2. Leid het type af van de volgende expressies: `curry fst` en `curry snd`. Wat is de betekenis van deze expressies?

3. Leid het type af van de volgende expressies: `uncurry (+)`, `uncurry (-)`, `uncurry (*)` en `uncurry (/)`. Wat is de betekenis van deze expressies?

5.3 Functie compositie

Main module:	*FunctieCompositie.icl*
Environment:	*StdEnv*

De functie compositie operator `o`, te definiëren als: `o f g x = f (g x)` (is in module `StdFunc` een beetje anders gedefiniëerd) kun je gebruiken om van twee bestaande functies eenvoudig een nieuwe functie te maken. Leg van de onderstaande composities uit wat ze doen:

```
e1 = ((*) 5)   o ((+) 1)
e2 = ((+) 1)   o ((*) 5)
e3 = ((*) 2)   o ((*) 2)
e4 = (min 100) o (max 0)
e5 = ((<) 2)   o length
```

5.4 Argumenten flippen

Main module:	*Flipper.icl*
Environment:	*StdEnv*

De functie `flip`, te definiëren als: `flip f a b = f b a` (is in module `StdFunc` een beetje anders gedefiniëerd) kun je gebruiken om de eerste twee argumenten van een functie om te draaien. Vergelijk de onderstaande functies met hun “geflipte” variant en leg uit wat het verschil is:

1. `(+) 4 2` versus `flip (+) 4 2`.
2. `(-) 4 2` versus `flip (-) 4 2`.
3. `(*) 4 2` versus `flip (*) 4 2`.
4. `(/) 4 2` versus `flip (/) 4 2`.

5.5 Twice

Main module:	*Twice.icl*
Environment:	*StdEnv*

Bestudeer de functie `twice` uit module `StdFunc`. Wat doet deze functie? Beredeneer wat de uitkomst is van de volgende `Start`-regel.

```
Start = (                        inc 0
      ,                    twice inc 0
      ,              twice twice inc 0
      ,        twice twice twice inc 0
      , twice twice twice twice inc 0
      )
```

Als je je antwoord wilt controleren m.b.v. de Clean IDE, moet je zowel de *Maximum Heap Size* als *Stack Size* op *1M* zetten.

5.6 Ellips omtrek

Main module:	*EllipsOmtrek.icl*
Environment:	*StdEnv*

De *omtrek* van een ellips met stralen r_1 en r_2 ($r_1 \geq r_2 > 0$) kan benaderd worden m.b.v. de volgende getallenreeks:

$$
\begin{aligned}
omtrek &= 2r_1\pi(1 - \sum_{i=1}^{\infty} s_i) \\
s_1 &= \frac{1}{4}e^2 \\
s_i &= s_{i-1} \cdot \frac{(2i-1)(2i-3)}{4i^2} \cdot e^2 \qquad \textbf{if } i > 1 \\
e &= \frac{\sqrt{r_1^2 - r_2^2}}{r_1}.
\end{aligned}
$$

Schrijf een programma dat de omtrek van een ellips met stralen r_1 en r_2 ($r_1 \geq r_2 > 0$) berekent met een gewenste precisie. Doe dit op een analoge wijze zoals is uitgewerkt in het diktaat in sectie 2.4.2. (blz.37-39) met de functie `until`.

5.7 Elementen groeperen(*gebruikt in 5.8,5.17*)

Main module:	*Group.icl*
Environment:	*StdEnv*

Schrijf een polymorfe functie `group :: (a -> Bool) [a] -> [[a]]` die als argumenten een predicaat p en een lijst xs krijgt. De functie berekent een *bijna-partitie* van xs. Een bijna-partitie is een partitie (zie 3.17) waarvan het eerste en het laatste element leeg mogen zijn. Laat $[A_0, A_1, \ldots A_{2 \cdot n}, A_{2 \cdot n+1}]$ het resultaat zijn van (`group` p xs). Dan geldt:

- Voor alle elementen uit iedere A_i met i een *even* getal is het predicaat p geldig. A_0 is alleen leeg als het eerste element van xs niet voldoet aan p.
- Voor alle elementen uit iedere A_i met i een *oneven* getal is het predicaat p ongeldig. $A_{2 \cdot n+1}$ is alleen leeg als het laatste element van xs voldoet aan p.
- `flatten (group` p xs`)` $= xs$, waarbij `flatten` uit de standaard omgeving van Clean komt. M.a.w.: `group` gooit geen elementen weg en introduceert geen nieuwe elementen.

Voorbeelden:

```
group isEven  [1 .. 10] = [[],[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[]]
group isOdd   [1 .. 10] = [   [1],[2],[3],[4],[5],[6],[7],[8],[9],[10]   ]
group isDigit ['7␣Dwergen␣en␣11␣prinsessen␣zoenden␣3␣kikkerprinsen.']
      = [['7'],['␣Dwergen␣en␣'],['11'],['␣prinsessen␣zoenden␣'],['3'],['␣kikkerprinsen.']]
```

5.8 Woordenlijst (gebruikt in 5.9,7.3,7.8,7.9)

Main module:	*Words.icl*
Environment:	*StdEnv*

Schrijf een functie `words` die een `Char` lijst krijgt en daar alle woorden uit selecteert. Een woord wordt simplistisch gedefiniëerd als een opeenvolgende reeks alphanumerieke tekens. Maak voor deze functie gebruik van de `group` uit opdracht 5.7, en roep deze dus aan met een geschikt predicaat (zoek in de standaard module StdChar).

5.9 Woordfrequentie (gebruikt in 7.9)

Main module:	*Woordfrequentie.icl*
Environment:	*Object IO*

In opdracht 3.19 is de functie `frequentielijst` gemaakt die van een lijst van elementen de frequentielijst berekent. In opdracht 5.8 is de functie `words` gemaakt die alle woorden selecteert uit een tekst, gerepresenteerd als een `Char` lijst. In Clean worden teksten normaal gesproken gerepresenteerd als `String` waarden (om precies te zijn: arrays (`{}`) van unboxed (`#`) characters (`Char`)), ofwel: `{#Char}`). Je kunt van een lijst van `Char`s een `String` maken middels de overloaded functie `toString`, en omgekeerd, van een `String` een `Char` lijst middels de overloaded functie `fromString`.

Combineer deze functies zodanig dat je een nieuwe functie maakt, `woordfrequentie`, die een tekst als argument krijgt, gerepresenteerd als `String`, en die de frequentielijst van alle woorden uit de tekst berekent. De woorden in de frequentielijst moeten zelf ook als `String` waarden gerepresenteerd zijn.

5.9.1 Optioneel: Woordfrequenties tonen

In opdracht 3.19 heb je gezien dat je frequentielijsten kunt visualiseren m.b.v. de functie `toonFrequentielijst` uit module `FrequentielijstGUI`. Deze visualisatie is alleen geschikt voor relatief korte lijsten en elementen met korte tekstrepresentatie. In module `FrequentielijstGUI` staat nog een tweede functie gedefiniëerd: `toonFrequentielijst2` die alle elementen *onder elkaar* afbeeldt. Figuur 5.1 toont de gegenereerde uitvoer.

Figuur 5.1: De frequentielijst van `['Hello␣world!␣Here␣I␣am!']`.

Als je de twee functies vergelijkt, dan zie je dat ze bijna hetzelfde zijn. Schrijf nu een derde, meer algemene functie, die door geschikte parameterisatie gebruikt kan worden

om de twee bestaande functies veel korter mee te realiseren.

5.10 Origami

Main module:	*Origami.icl*
Environment:	*StdEnv*

Druk de volgende functies uit `StdEnv` opnieuw uit m.b.v. `foldl` of `foldr` en λ-abstracties: `sum`, `prod`, `flatten`, `length`, `filter`, `reverse`, `takeWhile` en `maxList`.

5.11 Nou en of

Main module:	*NouEnOf.icl*
Environment:	*StdEnv*

Bestudeer de functies `and`, `or`, `all` en `any` uit module `StdList`. Verklaar in eigen woorden wat de betekenis is van deze functies. Schrijf de functie `and`` die gebruik maakt van `all` en dezelfde betekenis heeft als `and`. Schrijf de functie `or`` die gebruik maakt van `any` en dezelfde betekenis heeft als `or`.

`foldl` en `foldr`

De functies `and` en `or` kun je uitdrukken in `all` en `any`. Druk nu de functies `all` en `any` uit in zowel `foldl` en `foldr`. Noem deze `all_l`, `all_r` en `any_l` en `any_r`.

`foldl` of `foldr`?

Zowel de `&&` als de `or` operator zijn *conditionele* tests, d.w.z.: ze inspecteren eerst de waarde van het eerste argument, en als op grond daarvan al het resultaat bepaald kan worden, dan wordt het tweede argument niet geëvalueerd. Dit kun je aan het type van beide functies zien, die is in beide gevallen:

```
:: !Bool Bool -> Bool
```

De strictheidsannotatie (`!`) voor het eerste argument geeft aan dat de functie het argument zal evalueren, terwijl de afwezigheid van deze annotatie bij het tweede argument aangeeft dat het argument geëvalueerd zal worden als dat nodig is (luie evaluatie).

Voorspel wat er gebeurt als `all_l`, `all_r` en `any_l` en `any_r` toegepast worden op een oneindige lijst van boolean waarden. Doe dit aan de hand van de volgende voorbeelden:

```
Start = all_l id [False:repeat True ]
Start = any_l id [True :repeat False]
Start = all_r id [False:repeat True ]
Start = any_r id [True :repeat False]
```

5.12 Een betere `StdBool`

Main module:	*StdBool2.icl*
Environment:	*StdEnv*

Lift

Leid van de volgende functies de meest algemene types af en leg uit wat deze functies doen:

```
lift0 f          a = f a
lift1 f g1       a = f (g1 a)
lift2 f g1 g2    a = f (g1 a) (g2 a)
lift3 f g1 g2 g3 a = f (g1 a) (g2 a) (g3 a)
```

Klasse Booleans

In de module `StdBool` uit `StdEnv` zijn de volgende boolean operaties gedefinieerd:

```
not           :: !Bool      ->  Bool  // Not arg1
(||) infixr 2 :: !Bool Bool ->  Bool  // Conditional or of arg1 and arg2
(&&) infixr 3 :: !Bool Bool ->  Bool  // Conditional and of arg1 and arg2
```

Deze operaties zijn, in tegenstelling tot de rekenkundige operaties `+`, `-`, enz. niet *overloaded*. Het komt vaak voor dat je in programma's niet alleen `Bool` waarden wilt combineren tot een nieuwe `Bool`, maar ook predikaten (`a -> Bool`) tot een nieuw predikaat (`a -> Bool`). Om bijvoorbeeld alle elementen tussen 3 en 8 te selecteren moet je opschrijven:

```
Start = filter (\x -> 3 < x && 8 > x) [1 .. 10]
```

Als `&&` nu overloaded was, zou je deze ook op predikaten kunnen definiëren, met als resultaat:

```
Start = filter ((<) 3 && (>) 8) [1 .. 10]
```

Om dit te realiseren is de volgende definitie module gemaakt, die `StdBool` uitbreidt met de gewenste overloaded operatoren:

```
definition module StdBool2

import StdBool

class    ~~             a ::    !a -> a   // logical negation
class   (|||) infixr 2 a :: !a !a -> a   // logical or
class   (&&&) infixr 3 a :: !a !a -> a   // logical and

instance ~~  Bool
instance ||| Bool
instance &&& Bool

instance ~~  (a -> Bool)
instance ||| (a -> Bool)
instance &&& (a -> Bool)
```

Schrijf de bijbehorende implementatie module. Maak, indien mogelijk, gebruik van de `lift`*i* functies.

5.13 Romeinse Getallen, deel II

Main module:	*StdRoman.icl*
Environment:	*StdEnv*

In opdracht 4.4 heb je een implementatie gemaakt van romeinse getallen. Ontwikkel nu een nieuwe Clean *module* `StdRoman` waarin romeinse getallen en berekeningen beschikbaar worden gemaakt. Maak instanties van dezelfde overloaded klassen als die beschikbaar zijn voor `StdInt` met de volgende uitzonderingen:

- Hernoem `toInt` en `fromInt` naar `toRoman` en `fromRoman` respectievelijk en definieer de bijbehorende klasse definities.
- De `Int` operaties `bit(x)or`, `bitand`, `bitnot` en `<<` en `>>` zijn geen klassen en kunnen niet voor `Roman`s gebruikt worden.

Gebruik zoveel mogelijk hogere-orde functies voor je implementatie. Je implementaties van de meeste instanties zouden eigenlijk *one-liners* moeten zijn.

5.14 `scan` en `iterate`

Main module:	*ScanEnIterate.icl*
Environment:	*StdEnv*

Bestudeer de functies `scan` en `iterate` uit `StdList`. Bereken handmatig de uitkomst van de volgende `Start`-regels:

```
Start = scan (+) 0 [1 .. 10]
Start = scan (*) 1 [2 .. 10]
Start = take 5  (iterate (flip (^) 2) 2)
Start = take 10 (iterate (flip (/) 10) 123456)
```

5.15 Taylor reeksen

Main module:	*TaylorReeks.icl*
Environment:	*StdEnv*

Gebruik in deze opdracht alleen functies uit `StdEnv`, λ-abstracties en lijst comprehensions.

Plussen en minnen Schrijf een functie `plusminus` die een lijst $[x_0 \ldots x_n]$ ($n \geq 0$) van waarden krijgt en die de waarde $x_0 - x_1 + x_2 - x_3 + \ldots$ oplevert.

Taylor-reeks voor sinus De *Taylor*-reeks voor de *sinus* functie is als volgt gedefinieerd:

$$\mathit{sinus}\ x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!}$$

Implementeer de functie `sinus` die een benadering van deze reeks berekent.

Taylor-reeks voor cosinus De *Taylor*-reeks voor de *cosinus* functie is als volgt gedefinieerd:

$$\mathit{cosinus}\ x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n}}{(2n)!}$$

Implementeer de functie `cosinus` die een benadering van deze reeks berekent.

5.16 seq en seqList

Main module:	*SeqEnSeqList.icl*
Environment:	*StdEnv*

Bestudeer de functies `seq` en `seqList` uit `StdFunc`. Implementeer opnieuw de volgende functies uit de module `StdStack` die je in opdracht 6.5 gemaakt hebt:

- `pushes` m.b.v. `seq` en `push`.
- `popn` m.b.v. `seq` en `pop`.
- `topn` m.b.v. `seqList` en `top` en `pop`.
- `elements` m.b.v. `seqList` en `top` en `pop`.

5.17 Database queries

Main module:	*FQL.icl*
Environment:	*StdEnv*

In deze opdracht ga je een aantal database queries formuleren met behulp van lijst comprehensions en hogere-orde functies. Gebruik voor het inlezen van de database gegevens het programma `FQL.icl`. Deze importeert de module `StdT` die je in opdracht 6.4 gemaakt hebt. De `Start` regel van deze module leest het bestand *"Nummers.dbs"* in. Hierin staan de gegevens van bijna 5000 tracks van muziek CDs. Om alle gegevens in te kunnen lezen, heeft je applicatie de beschikking nodig over een grotere *heap* en *stack* dan standaard ingesteld is. Maak eerst een project aan. Pas vervolgens de opties aan ("Project:Project Options..."): de *"Maximum Heap Size"* wordt *"4M"* en *"Stack Size"* wordt *"1M"*. Na inlezen zijn alle tracks beschikbaar als element van een lijst van type `[Nummer]`:

```
::  Nummer = { groep :: String          // De naam van de groep
             , album :: String          // De naam van het album
             , jaar  :: Int             // Het jaar van uitgave
             , track :: Int             // Track nummer van het nummer (1 ..)
             , titel :: String          // Naam van het nummer
             , tags  :: [String]        // Beschrijvende tags van het nummer / album
             , lengte:: T               // Lengte van het nummer
             , land  :: [String]        // Land van oorsprong van groep
             }
```

Het *"Nummers.dbs"* bestand heeft de volgende eigenschappen:

- De inhoud is oplopend geordend in volgorde van bovenstaande velden, dus eerst op `groep`, dan op `album`, ... en ten slotte op `land`. Veldwaarden worden als `String` behandeld, en geordend volgens de `<` instance op `String` waarden.
- Ieder `album` van iedere `groep` heeft een unieke titel. Andersom geldt niet: er zijn `album` titels die door verschillende `groep`en gebruikt zijn (bijvoorbeeld *Up* is gebruikt door *Peter Gabriel* en *R.E.M.*).

Hieronder staan een aantal queries geformuleerd. Schrijf voor iedere query een functie die de query beantwoordt. Gebruik zoveel mogelijk lijst comprehensions, hogere-orde functies en standaard functies.

1. Schrijf de functie `alle_groepen :: [Nummer] -> [String]` die alle `groep`en opsomt zonder duplicaten en in oplopende volgorde gesorteerd volgens de `<` instance op `String` waarden.

2. Schrijf de functie `alle_jaarblokken :: [Nummer] -> [String]` die alle `jaar` van uitgave opsomt in oplopende volgorde *en* in verkorte notatie. De verkorte notatie houdt in dat van elk blok van tenminste twee opeenvolgende jaren alleen het beginjaar - eindjaar genoemd wordt; overige jaren worden individueel genoemd.

   Voorbeeld: stel dat de albums uit de volgende jaren komen: 1967, 1968, 1969, 1972, 1973, 1975, 1978. Dan is de uitvoer van deze functie: `["1967-1969", "1972-1973", "1975", "1978"]`.

3. Schrijf de functie `alle_albums_van :: String [Nummer] -> [(Int,String)]` die alle albums opsomt die een gegeven `groep` gemaakt heeft. Lever van deze albums het `jaar` van uitgave en `album` titel op zonder duplicaten, en sorteer in oplopende volgorde naar jaar van uitgave.

4. Schrijf de functie `alle_tracks :: String String [Nummer] -> [(Int,String,T)]` die alle tracks opsomt van een gegeven `album` en `groep`. Lever van iedere track het `track` nummer, de `titel` en de `lengte` op. Deze lijst moet gesorteerd zijn in oplopende volgorde van `track` nummer.

5. Schrijf de functie `speelduur_albums :: [Nummer] -> [(T,String,String)]` die alle albums van alle groepen oplevert. In ieder resultaat (a, b, c) is a de totale speelduur van het album, b de `groep` en c het `album`. Sorteer deze lijst in oplopende volgorde van totale speelduur. Welk album is het kortst en welk album is het langst?

6. Schrijf de functie `totale_speelduur :: [Nummer] -> T` die de totale speelduur van alle nummers samen oplevert. Druk deze af in $m^+ : ss$ notatie van opdracht 6.4, maar ook in aantal dagen uitgedrukt als een `Real`.

7. Schrijf de functie `nederlandse_metal :: [Nummer] -> [String]` die alle `groep`en oplevert zonder duplicaten waarin een van de `tags` `"metal"` is, en een van de `landen` `"Netherlands"`.

5.18 Pseudo-random getallen (gebruikt in 5.19, 7.4, 7.5, 7.12, 7.13)

Main module:	*RandomGetallen.icl*
Environment:	*Object IO*

De module `Random` geeft je een functie, `random`, om pseudo-random getallen mee te maken:

```
random :: RandomSeed -> .(Int,RandomSeed)
```

Deze functie krijgt als argument een `RandomSeed` waarde waarmee een pseudo-random getal berekend wordt van type `Int` en een nieuwe `RandomSeed` waarde. Met deze nieuwe `RandomSeed` waarde kun je een volgend pseudo-random getal berekenen, enzovoort.

Omdat we in een zuivere functionele taal werken, zal (`random rs`) altijd hetzelfde pseudo-random getal en nieuwe `RandomSeed` waarde opleveren. Een min of meer willekeurige begin `RandomSeed` verkrijg je met de functie `getNewRandomSeed`:

```
getNewRandomSeed :: *env -> (RandomSeed, *env) | TimeEnv env
```

Deze gebruikt zijn `env` argument om de huidige tijd te lezen, en maakt daarvan een initiële `RandomSeed` waarde. Je kunt de `*World` gebruiken die je bij de `Start` regel mee kunt krijgen als instantie van de overloaded klasse `TimeEnv`:

```
Start :: *World -> ...
Start world
# (rs,world) = getNewRandomSeed world
= ...
```

De `Random` module gebruikt de module `StdTime` uit *Object IO*. Zet daarom de environment op `Object IO`.

N **pseudo-random getallen** Schrijf de functie `random_n :: Int RandomSeed -> ([Int], RandomSeed)` die, gegeven een positief getal n, en een initiële `RandomSeed` waarde een lijst van n achtereenvolgende pseudo-random getallen oplevert. Maak gebruik van de functie `seqList` uit module `StdFunc` en `repeatn` uit `StdList`.

∞ **pseudo-random getallen** Schrijf de functie `random_inf :: RandomSeed -> [Int]` die *alle* achtereenvolgende pseudo-random getallen oplevert. Schrijf hiervoor eerst een algemene functie `iterateSt :: (s -> (a,s)) s -> [a]` die lijkt op `iterate` uit `StdList`.

Schudden In veel programma's (simulaties, spelletjes) moet je soms een gegeven reeks van waarden willekeurig ordenen. Schrijf de functie `shuffle :: [a] RandomSeed -> [a]` die een eindige lijst `xs` krijgt, en een `RandomSeed` waarde `rs` zodanig dat (`shuffle xs rs`) een willekeurige permutatie van `xs` oplevert (zie voor uitleg van het begrip permutatie opdracht 3.16).

5.19 `return en bind`

Main module:	*ReturnEnBind.icl*
Environment:	*Object IO*

Bestudeer de functies `return` en `bind` uit module `StdFunc`.

(`id`,`o`) versus (`return`,`bind`) Vergelijk `id` met `return` en `o` met `bind`. Wat is het verband tussen deze functies? Schrijf `bind` opnieuw, maar maak nu gebruik van `o` en `uncurry`. Maak dus de volgende definitie af:

```
(`bind1`) infix 0 :: (St s a) (a -> (St s b)) -> St s b
(`bind1`) f1 f2 = ... o ...
```

Toepassingen Maak met behulp van `return` en `bind1` onderstaande functies:

- Gebruik de `random` functie zoals uitgelegd in opdracht 5.18 en maak de functie (let op de haakjes!) `som2 :: (RandomSeed -> (Int,RandomSeed))` die twee randomgetallen genereert met behulp van het `RandomSeed` argument en hun som oplevert, samen met de gewijzigde `RandomSeed` waarde.
- Geef een andere implementatie van `seqList` uit `StdFunc`. Maak dus de volgende definitie af (let op het type!):

```
seqList1 :: [St s a] -> St s [a]
```

Hoofdstuk 6

Data abstractie

Data abstractie is een kernbegrip in programmeertalen. Data abstractie stelt je in staat de actuele realisatie van een data type te verbergen, en verplicht je goed na te denken over de toegangsfuncties (de interface) tot die data structuur. In Clean wordt data abstractie op twee manieren aangeboden. De eerste is met behulp van *modulen*, de tweede met behulp van *existentiële types*.

6.1 Breuken

Main module:	*StdQ.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `StdQ.dcl`. Het abstracte type `Q` implementeert de rationale getallen $\mathbb{Q}$, ofwel breuken. De volgende eigenschappen dienen te gelden voor de operaties op rationale getallen:

- De operaties `==` en `<` testen op gelijkheid en kleiner-dan.
- De operaties `+`, `-`, `*` en `/` zijn de gangbare rekenkundige operaties op rationale getallen. De waarden `zero` en `one` zijn de neutrale elementen van respectievelijk de optelling en vermenigvuldiging.
- `abs` neemt de absolute waarde; `sign` levert het teken op van het argument; `~` keert het teken om van het argument.
- `isInt` test of het argument correspondeert met een geheel getal. Als deze waar is, dan levert de `Q` instantie voor `toInt` exact hetzelfde getal op (dus zonder afronding). `toReal` zet een rationaal getal om naar een `Real` (bij benadering).

  De instanties van de klasse `toQ` hebben de omgekeerde functionaliteit: de `Int` en `Real` instantie zetten respectievelijk een `Int` en een `Real` (bij benadering) om naar een rationaal getal. Voor de `(Int,Int)` instantie geldt dat `(toQ (t,n))` het rationale getal $\frac{\mathtt{t}}{\mathtt{n}}$ maakt. Voor de `(Int,Int,Int)` instantie geldt dat `(toQ (d,t,n))` het rationale getal $\mathtt{d}\frac{\mathtt{t}}{\mathtt{n}}$ maakt. Voor de laatste twee instanties geldt tevens dat ze een foutmelding mogen geven als $\mathtt{n} = 0$.

Je kunt in je implementatie gebruik maken van de `Int` instantie van de functie `gcd` die de *grootste gemeenschappelijke deler* van twee gehele positieve getallen bepaalt. Deze is gedefinieerd in de module `StdInt` die je vanzelf krijgt als je `StdEnv` importeert.

6.2 Getallen

Main module:	*StdNum.icl*
Environment:	*StdEnv*

In Clean wordt middels het typeringsysteem onderscheid gemaakt tussen gehele getallen (`Int`) en floating point getallen (`Real`). Realiseer de implementatie module die hoort bij `StdNum.dcl`. Hierin worden operaties beschikbaar gesteld voor het werken met waarden die ofwel een `Int` zijn, ofwel een `Real`.

> **Optioneel:** Als je opdracht 6.1 gedaan hebt, dan kun je ook breuk waarden erbij nemen.

Kies bij de implementaties telkens de meest precieze representatie (converteer dus niet alles eerst naar `Real`s).

6.3 Uniforme verzamelingen

Main module:	*StdSet.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `StdSet.dcl` voor het werken met eindige verzamelingen van elementen van hetzelfde type. De volgende eigenschappen dienen te gelden voor de operaties op verzamelingen:

- De operaties `toSet` en `fromSet` zetten een lijst van elementen om in een verzameling. Verzamelingen bevatten geen dubbele elementen, dus de lijst die uit `fromSet` komt hoort ook geen dubbele elementen te bevatten.
- De predicaten `isEmptySet`, `isDisjoint` en `memberOfSet` testen achtereenvolgens of een gegeven verzameling leeg is (geen elementen bevat), of twee verzamelingen disjunct zijn (geen gemeenschappelijke elementen bevatten), en of een gegeven waarde voorkomt in de verzameling.
- De `Set` instantie van `zero` genereert de lege verzameling. De lege verzameling heeft geen elementen. De `Set` instantie van `==` test of twee verzamelingen identiek zijn (bevatten exact dezelfde elementen). De functie `nrOfElements` levert het aantal elementen op van een verzameling.
- Er zijn twee predicaten die een deelverzameling-relatie testen: het predicaat `isStrictSubset` bepaalt of het eerste argument een *strikte* deelverzameling is van het tweede argument (het tweede argument bevat elementen die niet in het eerste argument voorkomen) en het predicaat `isSubset` bepaalt of het eerste argument een deelverzameling is van het tweede element (alle elementen van het eerste argument zijn ook element van het tweede argument). Iedere verzameling is altijd een deelverzameling van zichzelf, maar geen *strikte* deelverzameling van zichzelf.
- De operaties `union` en `intersection` berekenen achtereenvolgens de vereniging en doorsnede van twee verzamelingen. De operatie `without` verwijdert alle elementen die in het tweede argument zitten uit het eerste argument. De functie `product` berekent het cartesisch product van twee verzameling.
- De functie `powerSet` berekent de machtverzameling (verzameling van alle deelverzamelingen) van de gegeven verzameling.

Maak naar eigen inzicht gebruik van *ZF*-expressies, ..-expressies of recursieve functies. Probeer een zo klein mogelijke kern van functies te ontwikkelen binnen deze module waarmee de overige functies in deze modulen mee gerealiseerd kunnen worden (bijvoorbeeld: twee verzamelingen zijn gelijk als ze elkaars deelverzameling zijn).

6.4 Tijd (*gebruikt in 5.17*)

Main module:	*StdT.icl*
Environment:	*StdEnv*

De lengtes van films en muziekstukken worden vaak in $m^+ : ss$ formaat weergegeven, waarbij m^+ het aantal minuten is, en ss het aantal seconden. Zo is bijvoorbeeld de lengte van de bioscoopfilm *"The Lord of the Rings: The Fellowship of the Ring"* 178:00, dus twee uur en 58 minuten, en de lengte van het muzieknummer *"Tea"* van Sam Brown op het album *"Stop!"* 0:41, dus 41 seconden.

Implementeer de module behorende bij `StdT.dcl` die het abstracte type `T` exporteert en de volgende functies:

- Vergelijkings operaties: `==`, `<`.
- Rekenkundige operaties: `zero`, `+`, `-` (waarbij $t_1 \leq t_2 \Rightarrow t_1 - t_2 =$ `zero`).
- Omzetten naar seconden: `toInt`, `fromInt` (waarbij $t < 0 \Rightarrow$ `fromInt` $t =$ `zero`).
- Omzetten naar tekst: `toString`, `fromString`. Als een tekst s niet voldoet aan het $m^+ : ss$ formaat, dan is `fromString` $s =$ `zero`.

6.5 Stack (*gebruikt in 5.16,6.9*)

Main module:	*StdStack.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `StdStack.dcl` voor het werken met stacks. De gebruikelijke eigenschappen dienen te gelden voor de operaties:

- `newStack` creëert een lege stack.
- (`push` x s) plaatst x bovenop stack s, en (`pushes` $[x_0 \ldots x_n]$ s) plaatst x_0, x_1, $\ldots x_n$ achtereenvolgens bovenop de stack. Aan het eind staat dus x_n op de top van de stack.
- (`pop` s) verwijdert het top-element van de stack s indien mogelijk, en (`popn` n s) verwijdert de bovenste n elementen van de stack s indien mogelijk.
- (`top` s) levert het top-element van de stack s indien aanwezig, en een run-time error als s leeg is. (`topn` n s) levert de bovenste n elementen van de stack s indien aanwezig, en een run-time error als er te weinig elementen zijn.
- (`elements` s) levert alle elementen van de stack s op, van de top van de stack naar de bodem. (`count` s) levert het aantal elementen op de stack s.

Probeer een zo klein mogelijke kern van functies te ontwikkelen binnen deze module waarmee de overige functies in deze modulen mee gerealiseerd kunnen worden (bijvoorbeeld: probeer `pushes` te definiëren met behulp van `push`).

6.6 Gesorteerde Lijst

Main module:	*StdSortList.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `StdSortList.dcl` die operaties aanbiedt voor het werken met lijsten waarvan de elementen gesorteerd zijn. De volgende eigenschappen dienen te gelden voor de operaties (let op de complexiteits-eis hieronder!):

- (`minimum` l) en (`maximum` l) leveren de minimum en maximum waarde op van l, en genereren een runtime error als l leeg is. De complexiteit van beide functies moet $\mathcal{O}(1)$ (constante tijd) zijn.
- `newSortList` creëert een lege, gesorteerde, lijst.
- (`memberSort` x l) test of waarde x in l voorkomt. De uitspraak (`memberSort` x `newSortList`) is dus altijd onwaar voor iedere x.
- (`insertSort` x l) voegt waarde x toe aan l op de juiste positie. De uitspraak (`memberSort` x `(insertSort` x l`))` is dus altijd waar voor iedere x en l.
- (`removeFirst` x l) verwijdert het eerste voorkomen van x uit l (er mogen dus duplicate elementen in l voorkomen). (`removeAll` x l) verwijdert alle voorkomens van x uit l. De uitspraak (`memberSort` x (`removeAll` x l)) is dus altijd onwaar voor iedere x en l.
- (`elements` l) levert alle elementen van l op. Deze lijst is gesorteerd en mag duplicate elementen bevatten. (`count` l) levert het aantal elementen van l op. Er geldt dus: `length (elements` l`)` $=$ `count` l.
- (`mergeSortList` l_1 l_2) voegt l_1 en l_2 samen tot een nieuwe gesorteerde lijst. Er geldt: `count` l_1+ `count` l_2 $=$ `count (mergeSortList` l_1 l_2`)`; als (`memberSort` x l_1), dan geldt ook (`memberSort` x `(mergeSortList` l_1 l_2`))` en (`memberSort` x `(mergeSortList` l_2 l_1`))`.

6.7 Associatie Lijst

Main module:	*StdAssocList.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `StdAssocList.dcl`. In een associatie lijst van type `AssocList k a` worden elementen van type `a` m.b.v. een *key* van type `k` opgeslagen. Per *key* wordt één element opgeslagen. De volgende eigenschappen dienen te gelden:

- `newAssocList` creëert een lege associatie-lijst.
- (`countValues` l) levert het aantal opgeslagen elementen op.
- (`lookupKey` k l) levert de singleton-lijst met waarde v op indien het *key-value* paar (k, v) in l aanwezig was, en een lege lijst in het andere geval.
- (`updateKey` k v l) voegt het *key-value* paar (k, v) toe aan l indien k nog niet aanwezig was in l, en wijzigt de voorgaande waarde in v indien al een waarde geassocieerd was met k.

- (`removeKey` k l) verwijdert het *key-value* paar dat geassocieerd is met k indien aanwezig, en laat l ongemoeid in het andere geval.

6.8 Tekstcompositie^(*gebruikt in 8.2,8.7,8.9*)

Main module:	*TextCompose.icl*
Environment:	*StdEnv*

Realiseer de implementatie module die hoort bij `TextCompose.dcl`. Deze module biedt een abstract type `Text` en bijbehorende operaties aan waarmee je gemakkelijk ASCII-tekstblokken kunt samenstellen. Een dergelijk tekstblok bestaat uit een aantal kolommen (de 'breedte') en regels (de 'hoogte'). Deze `Text` waarden kun je naast elkaar plaatsen (terwijl je de verticale *alignment* controleert) en onder elkaar (terwijl je de horizontale *alignment* controleert). Hiermee kun je op een relatief eenvoudige wijze op een ruimtelijke manier de structuur van recursieve data structuren weergeven. Dat is erg handig om je functies te testen.

De voorgegeven type definities zijn:

```
:: Text                                      // het abstracte type voor een tekstblok
:: AlignH     =   LeftH | CenterH | RightH   // align left, center, right horizontally
:: AlignV     =   TopV  | CenterV | BottomV  // align top, center, bottom vertically
:: Width      :== NrOfChars
:: Height     :== NrOfLines
:: NrOfChars  :== Int                        // 0 ≤ nr of chars
:: NrOfLines  :== Int                        // 0 ≤ nr of lines
```

Implementeer de volgende operaties:

- de `Text` instantie van de overloaded `zero` functie genereert een tekstblok dat nul tekens breed is en nul regels tekst bevat.
- De bewerking `(toText (ah,reqw) (av,reqh) a)` transformeert de waarde `a` eerst naar een 'gewone' `String` met behulp van de standaard klasse `toString`. Vervolgens wordt een tekstblok gemaakt dat *tenminste* `reqw` tekens breed is en *tenminste* `reqh` regels bevat. Als de tekst meer tekens dan wel regels nodig heeft, dan worden deze gegenereerd. De waarde `ah` bepaalt de horizontale alignment, en de waarde `av` de verticale.
- De `Text` instantie van de standaard `toString` overloaded functie transformeert een tekstblok naar een `String` waarin met behulp van *newline* tekens de regel-overgangen geïmplementeerd zijn.
- De overloaded functie `sizeOf` bepaalt van zijn argument het aantal tekens en regels waaruit deze bestaat. De `String` instance behandelt zijn argument als een tekstblok van één regel en de `Text` instantie levert diens dimensies op.
- De bewerking `(horz av ts)` zet alle tekstblokken in `ts` *naast elkaar*, en zorgt ervoor dat hun alignment overeenkomt met `av`. De breedte van het resultaat tekstblok is dus de som van de breedtes van de elementen uit `ts`, en de hoogte is de maximum hoogte van de elementen uit `ts`. De bewerking `(vert hv ts)` is analoog, maar plaatst alle tekstblokken *onder elkaar*, gebruik makend van de alignment waarde `hv`. De breedte is dus de maximum breedte van de elementen uit `ts` en de hoogte is de som van de hoogten van deze elementen.

- De bewerking `(repeath n c)` creëert een tekstblok van één regel die bestaat uit `n c` tekens. De bewerking `(repeatv n c)` creëert een tekstblok van één kolom die bestaat uit `n c` tekens.
- De bewerking `(frame t)` maakt een ASCII-art tekstframe om het tekstblok `t`.

Voorbeeld:
`toString (frame (toText (CenterH,10) (BottomV,4) 42))`
produceert:

```
------------
|          |
|          |
|          |
|    42    |
------------
```

Voorbeeld:
```
toString
    (frame
        (toText (RightH,16)
                (CenterV,6)
                "Functional\nProgramming\nis\nFUN!"))
```
produceert:

```
------------------
|                |
|      Functional|
|     Programming|
|              is|
|            FUN!|
|                |
------------------
```

6.9 Stack, deel II

Main module:	*TestStdStack2.icl*
Environment:	*StdEnv*

In opdracht 6.5 heb je met behulp van een module een abstract data type (`Stack a`) gemaakt. In deze opdracht maken we opnieuw een abstract stack data type (`Stack2 a`) gebruik makend van existentiële quantificatie (`E.`). De signatuur van dit nieuwe type stack is:

```
definition module StdStack2

:: Stack2 elem = E. impl: { stack    :: impl
                          , push     :: elem impl ->  impl
                          , pop      ::      impl ->  impl
                          , top      ::      impl ->  elem
                          , elements ::      impl -> [elem]
                          }
```

Een (`Stack2 elem`) is een stack waarin elementen van type `elem` opgeslagen worden. Het type van de implementatie (`impl`) is verborgen door de existentiële quantifier.

`Stack2` instanties

Breid de module `TestStdStack2` uit met twee functies die nieuwe stack objecten aanmaken:

```
listStack :: Stack2 a
charStack :: Stack2 Char
```

De implementatie van `listStack` moet met lijsten van type `[a]` werken, en de implementatie van `charStack` moet met een `String` werken. Maak dus de volgende implementaties af:

```
listStack = { stack = [], ... }
charStack = { stack = "", ... }
```

`Stack2` toegangsfuncties, deel I

De existentiële quantifier werkt als een zwarte doos. Voer de volgende expressie in in de module `StdStack2.icl` en compileer je module opnieuw.

```
test nieuw stack2 = { stack2 & stack = stack2.push nieuw stack2.stack }
```

Dit zal tot een typeringsfoutmelding leiden. Leg uit waarom.

`Stack2` toegangsfuncties, deel II

Pattern matching is de enige toegestane manier om 'in te breken' in de encapsulatie van een existentiële quantifier. Om een waarde `nieuw :: elem` op een stack `stack2 :: Stack2 elem` te *push*-en gebruik je een expressie van de volgende vorm:

```
test nieuw stack2 = case stack2 of
                      { stack, push } = { stack2 & stack = push nieuw stack }
```

of:

```
test nieuw stack2=:{ stack, push } = { stack2 & stack = push nieuw stack }
```

Dat is nogal omslachtig. Voeg de volgende toegangsfuncties toe aan `StdStack2.icl` en implementeer en exporteer deze.

```
push     :: elem (Stack2 elem) -> Stack2 elem
pop      ::      (Stack2 elem) -> Stack2 elem
top      ::      (Stack2 elem) ->  elem
elements ::      (Stack2 elem) -> [elem]
```

Vergelijk de modulen `StdStack` en `StdStack2`. Zijn ze hetzelfde? Leg uit waarom je dat vindt.

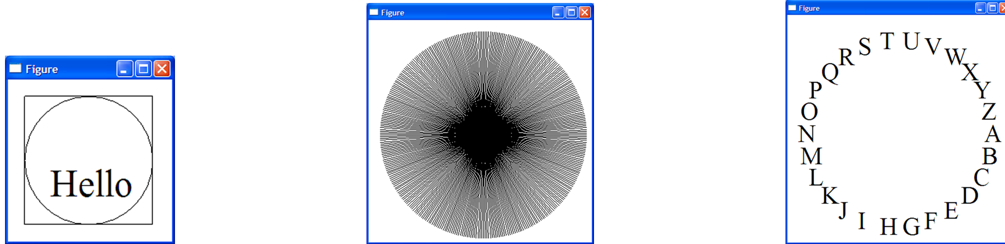
6.10 Eenvoudige graphics

Main module:	*TestFigure.icl*
Environment:	*Object IO*

Met de operaties uit de module `Figure` kun je eenvoudige, statische, afbeeldingen maken. De basis-elementen zijn het tekenen van een lijn van punt A naar punt B (`line` A B), een rechthoek met diagonaal-hoekpunten A en B (`rectangle` A B), een ellips die binnen een dergelijke rechthoek past (`ellips` A B) en een tekst t waarvan de linkerbovenhoek van de omvattende rechthoek positie A heeft (`text` t A).

Een reeks figuren $f_0 \dots f_n$ wordt samengesteld tot een nieuwe figuur m.b.v. `mkFigures` $[f_0, \dots, f_n]$, waarbij de figuren in oplopende index-volgorde getekend worden.

De functie `drawFigure` f neemt een figuur f, en opent een *window* waarin f getekend wordt. In de module `TestFigure.icl` vind je een aantal voorbeelden van figuren (`figure1` tot en met `figure5`). Figuur 6.1 toont een aantal van deze figuren.



Figuur 6.1: `figure1`, `figure4` en `figure5`

Kleur en penbreedte Bestudeer de implementatie van `Figure.icl`. Breid de module uit met de mogelijkheid om figuren in *kleur* te tekenen (`pencolour :: Colour -> Figure`) en om de *penbreedte* te veranderen (`pensize :: Int -> Figure`). De basisfuncties die je hiervoor nodig hebt vind je in `StdPicture.dcl`, n.l. `setPenColour` en `setPenSize`.

Toepassingen Gebruik bovenstaande uitbreidingen om elke letter in `figuur5` in een andere kleur te tekenen, en om een reeks gelijkbenige n-hoeken te tekenen ($n = 3, 4, 5, \ldots$).

Hoofdstuk 7

Console en File I/O

In dit hoofdstuk houden we ons bezig met eenvoudige console en file I/O. Onder console I/O verstaan we tekstgebaseerde invoer en uitvoer van en naar het console window. Met file I/O kunnen we bestanden benaderen in tekstformaat en binair formaat. Beide vormen van I/O zijn gebaseerd op het *uniqueness typeringssysteem* van Clean, en gebruiken dezelfde abstractie, n.l. `File` en `*File`.

7.1 Echo

Main module:	*Echo.icl*
Environment:	*StdEnv*

Schrijf een console I/O programma dat achtereenvolgens de volgende acties uitvoert:

- de console file `stdio` wordt geopend uit de `world`;
- er wordt herhaaldelijk een regel invoer ingelezen van `stdio` en ge-echoot via `stdio`;
- als de invoer leeg is, dan stopt het programma en sluit het de console file `stdio` netjes af in de `world`.

7.2 Oh dennenboom, deel II

Main module:	*OhDennenboom2.icl*
Environment:	*StdEnv*

Schrijf een console I/O programma dat herhaaldelijk als invoer een getalswaarde leest en de bijbehorende dennenboom zoals beschreven in opdracht 2.8 naar de console schrijft. Als je opdracht 2.8 of 6.8 gedaan hebt, dan mag je uiteraard die functies gebruiken. Bij een niet-numerieke invoer stopt het programma.
Let op: de functie `toInt` die een `String` converteert naar `Int` faalt als de string niet precies een geheel getal is. Dit gaat in het bijzonder mis met invoer van de console die met een newline tekent eindigt. Gebruik de `String` instantie van de overloaded functie `%` uit `StdString` om het newline teken van een invoerstring af te halen.

7.3 Zinspelingen

Main module:	*Zinspelingen.icl*
Environment:	*StdEnv*

Schrijf in deze opdracht een console I/O programma dat herhaaldelijk een commando (hieronder beschreven) herkent en de bijbehorende uitvoer naar de console schrijft. Maak gebruik van de `words` functie uit opdracht 5.8.

Beginstukken van een zin Het commando `firsts` gevolgd door een zin z, bestaande uit woorden $w_0 \ldots w_n$ (afgesloten met een newline teken) toont onder elkaar de zinnen met de woorden w_0, w_0 en w_1, w_0 t/m w_3 $\ldots w_0$ t/m w_n. Gebruik de functie `firsts` uit opdracht 3.13.

Fragmenten van een zin Het commando `frags` gevolgd door een zin z, bestaande uit woorden $w_0 \ldots w_n$ (afgesloten met een newline teken) toont onder elkaar de zinnen met alle woordfragmenten. Gebruik de functie `frags` uit opdracht 3.14.

Deelwoorden van een zin Het commando `subs` gevolgd door een zin z, bestaande uit woorden $w_0 \ldots w_n$ (afgesloten met een newline teken) toont onder elkaar de zinnen met alle deelwoorden. Gebruik de functie `subs` uit opdracht 3.15.

Permutaties van een zin Het commando `perms` gevolgd door een zin z, bestaande uit woorden $w_0 \ldots w_n$ (afgesloten met een newline teken) toont onder elkaar de zinnen met alle woord permutaties. Gebruik de functie `perms` uit opdracht 3.16.

7.4 Mastermind

Main module:	*Mastermind.icl*
Environment:	*StdEnv*

Mastermind is een spel voor twee spelers: de *codemaker* en de *codebreker*:

- De *codemaker* bedenkt een code die bestaat uit *vier* mogelijke kleuren. Hij kan kiezen uit *acht* kleuren: *wit*, *zilver*, *groen*, *rood*, *oranje*, *roze*, *geel* en *blauw*. Er zijn geen beperkingen aan de code: deze mag bestaan uit vier dezelfde kleuren, maar ook vier verschillende kleuren. Er zijn dus in totaal $8^4 = 4096$ combinaties mogelijk.
- De *codebreker* probeert de code te raden in zo min mogelijk pogingen. Als hij meer dan *twaalf* pogingen nodig heeft, heeft hij verloren. Een poging bestaat uit het geven van een concrete kleurcode. De *codemaker* geeft twee antwoorden:
  - Het aantal kleuren dat precies klopt, d.w.z.: ze hebben de juiste kleur én staan op de goede plek.
  - Het aantal kleuren (afgezien van het aantal kleuren die precies kloppen), die wel in de code voorkomen maar niet op de juiste plek staan.

Schrijf een console I/O programma dat het spel *Mastermind* implementeert waarbij de gebruiker de rol van *codebreker* heeft, en het programma de rol van *codemaker*. Voor het genereren van willekeurige codes kun je met de *pseudo-randomgenerator* werken uit opdracht 5.18.

Maak de implementatie zodanig dat deze abstraheert van de constante waarden die hierboven genoemd zijn: het moet dus voor ieder aantal te selecteren kleuren werken, voor iedere code-lengte, en voor ieder maximum aantal pogingen.

7.5 Pesten

Main module:	*Pesten.icl*
Environment:	*StdEnv*

Pesten is een kaartspel dat door meerdere spelers kan worden gespeeld. Het kan met één of meer complete kaartspelen gespeeld worden, inclusief jokers. Het spelverloop is als volgt.

Start: De kaarten worden geschud, en iedere speler krijgt 7 kaarten. De overige kaarten worden op de kop (dus de waarde en kleur is onzichtbaar) op de *afneemstapel* geplaatst. De bovenste kaart wordt van de afneemstapel genomen en zichtbaar neergelegd. Dit wordt de *aflegstapel*.

Speelrichting: De volgorde van spelen is alsof de spelers in een kring zitten: initiëel wordt er met de klok mee gespeeld. De speelrichting kan echter meerdere malen omkeren tijdens het spel door het spelen van de aas. Spelers veranderen niet van plaats tijdens een spel.

Speelactie zonder pestregels: De speler die aan de beurt is kijkt of hij een kaart in zijn bezit heeft met ofwel dezelfde kleur ofwel dezelfde waarde (zie voor uitleg terminologie opdracht 4.3) als de kaart die bovenop de aflegstapel ligt. Als dit zo is, mag hij die kaart op de aflegstapel leggen. Als dit niet zo is, moet hij een kaart van de afneemstapel pakken.

Zodra de afneemstapel leeg is, dan blijft de bovenste kaart van de aflegstapel liggen en alle andere kaarten worden opnieuw geschud en op de kop neergelegd als nieuwe afneemstapel. De speler neemt alsnog de bovenste kaart van de nieuwe afneemstapel.

Winnaar: Dit gaat zo door tot de eerste speler al zijn kaarten kwijt is. Deze wint het spel.

Speelactie met pestregels: Het spel heet niet voor niets pesten: je kunt je tegenstanders dwingen kaarten af te nemen van de afneemstapel. Er bestaan allerlei varianten van pestkaarten. Een er van is de volgende:

- Een kaart met waarde 2 verplicht de volgende speler tot het nemen van twee kaarten van de afneemstapel tenzij hij ook een kaart met waarde 2 speelt (en dit gaat door voor volgende spelers tot er geen kaart met waarde 2 gespeeld wordt). De eerste speler die geen kaart met waarde 2 heeft gespeeld moet kaarten van de afneemstapel pakken, en wel *twee* maal het aantal spelers dat achtereenvolgens een 2 gespeeld heeft. Hij mag zelf niet meer spelen.
- Een kaart met waarde 7 ("zeven blijft kleven") stelt de speler in staat nóg een kaart met dezelfde kleur of waarde te spelen. Als hij dit niet kan, dan moet hij zelf een kaart van de afneemstapel pakken.
- Een kaart met waarde 8 ("acht wacht") verplicht de volgende speler de beurt over te slaan.

- Een boer (ongeacht de kleur) stelt de huidige speler in staan een nieuwe kleur te kiezen waarmee verder gespeeld moet worden.
- Een kaart met waarde aas keert de speelrichting om.
- Een joker verplicht de volgende speler tot het nemen van 5 kaarten van de afneemstapel tenzij hij ook een joker speelt (en dit gaat door voor volgende spelers tot er geen joker gespeeld wordt). De eerste speler die geen joker heeft gespeeld moet kaarten van de afneemstapel pakken, en wel *vijf* maal het aantal spelers dat achtereenvolgens een joker gespeeld heeft. Hij mag zelf niet meer spelen, maar mag wel, zoals bij de boer, bepalen wat de nieuwe kleur wordt.

Schrijf een console I/O programma dat *pesten* implementeert waarbij de gebruiker telkens de beginspeler is, en het programma alle overige spelers voor zijn rekening neemt. Voor het schudden van de speelkaarten kun je met de `shuffle` functie werken uit opdracht 5.18.

Maak de implementatie zodanig dat deze werkt voor een willekeurig aantal kaartensets (tenminste één) en het aantal spelers (twee tot en met zes). Gebruik de implementatie van kaarten uit opdracht 4.3 en breid deze uit met jokers.

7.6 Een module voor file I/O

(gebruikt in 7.8,7.9,7.12,7.13,7.14,12.1)

Main module:	*TestSimpleFileIO.icl*
Environment:	*Object IO*

Om met bestanden te kunnen werken kun je de standaard Clean module `StdFile` gebruiken. In veel gevallen heb je alleen maar behoefte aan het volledig inlezen van een invoerbestand en het wegschrijven van een tekstbestand.

Basismodule

Ontwikkel een module `SimpleFileIO` die de volgende signatuur heeft:

```
definition module SimpleFileIO

import StdFile, StdMaybe

readFile  :: String        *env -> (Maybe String,*env) | FileSystem env
writeFile :: String String *env -> (Bool,        *env) | FileSystem env
```

Het eerste argument van `readFile` en `writeFile` is een padnaam naar een bestand, in hetzelfde formaat dat nodig is voor de functies `fopen` en `sfopen` uit `StdFile`. Het laatste argument van beide functies is een geschikte omgeving waarop de standaard file I/O operaties geïnstantieerd zijn (bijvoorbeeld de `*World`). Uiteraard is de resultaatwaarde van dit type de gewijzigde omgeving van de functie na aanroep. De functie `readFile` leest de complete inhoud *txt* van het gegeven bestand in en levert het resultaat op als (`Just` *txt*). Het desbetreffende bestand dient geopend, gelezen en gesloten te worden. Mocht er iets misgaan, dan wordt als eerste resultaat `Nothing` opgeleverd. De functie `writeFile` krijgt als tweede parameter de te schrijven nieuwe inhoud van het opgegeven bestand. Als het openen, schrijven en sluiten van het bestand allemaal lukt, dan is het `Bool` resultaat `True`; anders is het `False`.

Je kunt deze module testen met de eerste `Start` regel uit de main module `TestSimpleFileIO.icl`. Deze kopiëert zichzelf naar een bestand met de naam `"TestSimpleFileIO1.icl"`.

Regel voor regel

Breid deze module uit met een lees-functie en een schrijf-functie die de inhoud van het bestand als een lijst van regels (afgesloten met `'\n'` op eventueel de laatste regel na) beschouwt:

```
readLines  :: String          *env -> (Maybe [String],*env) | FileSystem env
writeLines :: String [String] *env -> (Bool,          *env) | FileSystem env
```

Je kunt deze nieuwe functies testen met de tweede `Start` regel uit de main module `TestSimpleFileIO.icl`. Deze slaat de regels uit `TestSimpleFileIO1.icl` in omgekeerde volgorde op in `TestSimpleFileIO2.icl`.

Een overloaded interface

Breid deze module uit met een overloaded functie `mapFile`:

```
mapFile :: String String (a -> b) *env -> (Bool,*env) | FileSystem env
                                                      & ... a
                                                      & ... b
```

Het eerste argument van `mapFile` is een padnaam naar een bestand waarvan de inhoud gelezen wordt, net zoals `readFile` hierboven. Het tweede argument van `mapFile` is een padnaam naar een bestand dat geschreven wordt, net zoals `writeFile` hierboven. Het derde argument, een functie van type `a -> b`, moet worden toegepast op de inhoud van het gelezen bestand, uiteraard na de benodigde conversies. Het resultaat hiervan moet geschreven worden naar het doelbestand, uiteraard na de benodigde conversies. Bedenk zelf welke *class* restricties benodigd zijn op `a` en `b`, en ontwikkel hiervoor zelf de eventueel benodigde *class*en.

Je kunt deze nieuwe functie testen met de derde `Start` regel uit de main module `TestSimpleFileIO.icl`. Deze verandert alle tekens uit `"TestSimpleFileIO2.icl"` in hoofdlettertekens en schrijft deze naar een bestand met de naam `"TestSimpleFileIO3.icl"`.

7.7 Monadische Console en File I/O

Main module:	*StdIOMonad.icl*
Environment:	*StdEnv*

In de practicum-folder vind je de module `StdMonad` waarin de *monadische operaties* `return` en `>>=` worden gedefinieerd. Daarnaast tref je het `fail` pattern aan. In de practicum-folder vind je ook de modulen `StdMaybeMonad`, `StdListMonad` en `StdStateMonad` die de instanties van deze operaties definiëren voor de respectievelijke type constructoren `Maybe`, `[]` en `ST`.

In deze opdracht ontwikkel je de module `StdIOMonad` die het type `IO` als instantie van de monadische operaties aanbiedt. Deze module maakt monadische versies van een klein aantal console en file functies:

- Het lezen van een invoerregel (inclusief *newline*) van de console (`read`), en het schrijven van een string naar de console (`write`).

- Het openen en sluiten van een file (`open` en `close`). De file wordt geïdentificeerd middels zijn file-naam. We beperken ons tot tekst-files en bieden daarom alleen de file-modi `Lees` en `Schrijf` aan. Een file die open is op het moment van evalueren van de `open` functie kan niet geopend worden. Als de `open` functie een file succesvol opent, dan wordt er een *file handle* (`Filehandle`) naar die file teruggegeven die gebruikt moet worden voor de verdere operaties op de file. Alleen een file die open is op het moment van evalueren van de `close` functie kan succesvol gesloten worden. Vanaf dat moment is de *file handle* onbruikbaar. De file zelf kan opnieuw bewerkt worden door hem weer te openen, hetgeen een nieuwe *file handle* oplevert.
- Het lezen en schrijven van een file (`eof`, `readline` en `writeline`). De functie `eof` bepaalt of alle tekst data uit de geïdentificeerde file gelezen is (mits de file open is op het moment van aanroepen). De functie `readline` leest de eerstvolgende tekstregel, inclusief *newline* teken mits aanwezig, uit de geïdentificeerde file. De functie `writeline` schrijft de tekstregel naar de geïdentificeerde file.

Tenslotte is er de functie `doIO` die een monadische berekening van type `(IO a)` omzet naar een Clean functie die de `*World` manipuleert. Als m een monadisch programma is, dan is `Start world = doIO` m `world` het equivalente Clean programma.

7.8 Woordenlijst, deel II

Main module:	*WC.icl*
Environment:	*StdEnv*

Schrijf een functie `wc` (*word count*) die als argument de naam van een tekstbestand krijgt en het aantal woorden telt in dat bestand. Gebruik voor het bepalen van woorden de simplistische functie `words` uit opdracht 5.8. Gebruik voor het lezen van een bestand de module `SimpleFileIO` uit opdracht 7.6.

7.9 Woordfrequentie, deel II

Main module:	*WF.icl*
Environment:	*Object IO*

Schrijf een functie `wf` (*woord frequentie*) die als argument de naam van een tekstbestand krijgt en de frequentietabel oplevert van de woorden in dat bestand zoals beschreven in opdracht 5.9. Gebruik voor het lezen van een bestand de module `SimpleFileIO` uit opdracht 7.6. Als je de optionele opdracht 5.9.1 gedaan hebt, dan kun je beter voor de visualisatie de functie `toonFrequentielijst2` gebruiken.

7.10 Gesorteerde files en bomen(gebruikt in 7.11)

Main module:	*GesorteerdBestandNaarBoom.icl*
Environment:	*Object IO*

Bestanden zijn vaak gesorteerd. Neem bijvoorbeeld het bestand *"Nederlands_lexicon"* dat je in de Clean-distributie kunt vinden in de directory:

```
Examples\ObjectIO Examples\scrabble\Nederlands
```

Schrijf de functie `readSortedFile` die als argument het padnaam van een gesorteerd tekstbestand krijgt, en die deze inleest als een *gebalanceerde binaire zoekboom*. Iedere regel (afgesloten met `'\n'`) in het tekstbestand is één element (zonder `'\n'`). Als je opdracht 8.10 gedaan hebt (*AVL* bomen), dan mag je deze gebruiken. Maak anders gebruik van het feit dat het bestand gesorteerd is.

Test je functie op het hierboven genoemde bestand. Pas hiertoe de waarde aan van de `String cleanpad`. Deze moet het padnaam krijgen van je Clean-distributie (de directory waarin je *CleanIDE.exe* staat). Je zult waarschijnlijk de *Maximum Heap Size* van je applicatie moeten vergroten. Anders krijg je de *run-time* foutmelding *Heap full*. Zet in "Project:Project Options...:" de "Maximum Heap Size" op 1M en hercompileer je applicatie.

Als dit werkt, schrijf dan de inverse functie `writeSortedFile` die een binaire zoekboom naar een gesorteerd tekstbestand schrijft.

7.11 Boggle

Main module:	*Boggle.icl*
Environment:	*Object IO*

Boggle (`http://en.wikipedia.org/wiki/Boggle`) is een woordspel dat bestaat uit 16 dobbelstenen waarop letters staan in plaats van ogen. De dobbelstenen worden geschud en vervolgens in een 4×4 matrix geplaatst. Spelers krijgen 3 minuten de tijd om zoveel mogelijk woorden uit deze letters te halen. Een woord moet tenminste uit 3 letters bestaan. Woorden worden gevormd door vanuit een willekeurige letter een buur te nemen die boven, onder, links, rechts, of schuin in alle richtingen staan. Iedere dobbelsteen mag ten hoogste één keer gebruikt worden. Het langst mogelijke woord bestaat dus uit 16 letters. Woorden mogen in enkelvoud en meervoud voorkomen. Werkwoorden mogen niet worden vervoegd. Voltooid verleden tijd is wel toegestaan. Verder zijn eigennamen, buitenlandse woorden, afkortingen en samengestelde woorden niet toegestaan.

Na afloop worden punten toegekend aan de gevonden woorden. Woorden met 3 en 4 letters zijn 1 punt waard, 5 letters 2 punten, 6 letters 3 punten, 7 letters 5 punten en 8 of meer letters 11 punten.

Implementeer een algoritme dat gegeven een willekeurige 4×4 matrix van letters en een woordenlijst alle mogelijke woorden van tenminste 3 letters opzoekt. Officiëel moeten de woorden in een Nederlands woordenboek terug te vinden zijn. Voor deze opdracht kun je het *"Nederlands_lexicon"* gebruiken in de directory:

```
Examples\ObjectIO Examples\scrabble\Nederlands
```

Pas de *Maximum Heap Size* aan van je applicatie indien je de *run-time* foutmelding *Heap full* krijgt. Om efficiënt woorden te kunnen vinden, kun je deze het beste inlezen als een gebalanceerde binaire zoekboom. Gebruik hiervoor de functie die je ontwikkeld hebt in opdracht 7.10.

Terzijde: het Nederlandstalige Boggle spel heeft de volgende letterverdeling op de dobbelstenen:

```
A A E I O W     D E N O S T     E E F H I S     E K N O T Z
B D J M O Q     E I G N T V     E H I N R S     E G L R U W
A I M O R S     E G J N U Y     A B I L N T     A F I K R X
A D E N V Z     A C E H R S     E L N P T U     A C D E M P
```

7.12 Galgje

Main module:	*Galgje.icl*
Environment:	*Object IO*

Het spel *galgje* is een woordraadspel voor twee spelers. Eén speler bedenkt een woord bestaande uit tenminste twee letters dat geraden moet worden door de andere speler, de *rader*, en geeft aan uit hoeveel letters het te raden woord bestaat. De andere speler probeert in zo min mogelijk aantal beurten het woord te raden. In iedere beurt mag hij één van de volgende dingen doen:

- Hij noemt een letter. De bedenker van het woord geeft aan waar de letter in het woord voorkomt. Als dit aantal 0 is, dan is de poging fout en krijgt de rader een strafpunt.
- Hij noemt een woord. De bedenker geeft aan of het woord geraden is. Als het klopt heeft de rader gewonnen. Als het niet klopt dan krijgt de rader een strafpunt.

Na iedere beurt waarin de rader een strafpunt gekregen heeft wordt een deel van een galgje getekend. Dit kan m.b.v. ASCII-*art* gedaan worden. Een voorbeeld:

```
                        ------     ------     ------     ------     ------
                  |        \ |     |  \ |     |  \ |     |  \ |     |  \ |
                  |         \|         \|     o   \|     o   \|     o   \|
                  |          |          |          |    /O\   |    /O\   |
                  |          |          |          |          |    / \   |
                  |          |          |          |          |          |
--------   --------   --------   --------   --------   --------   --------
```

Als de hele galg af is, dan heeft de rader het spel verloren.

Schrijf een console I/O programma dat het spel *galgje* implementeert waarbij de gebruiker de rol van *rader* heeft, en het programma de rol van bedenker van woorden. Je kunt het *"Nederlands_lexicon"* gebruiken in de directory:

```
Examples\ObjectIO Examples\scrabble\Nederlands
```

van de Clean-distributie. Gebruik voor het inlezen van woorden de module `SimpleFileIO` die je in opdracht 7.6 gemaakt hebt. Voor het kiezen van willekeurige woorden kun je met de *pseudo-randomgenerator* werken uit opdracht 5.18.

7.13 One-Time Pad (OTP) encryptie

Main module:	*OTP.icl*
Environment:	*Object IO*

In deze opdracht werk je aan een programma dat arbitrair grote tekstbestanden kan versleutelen en ontcijferen volgens de *one-time pad* methode[1]. Hiervoor zijn randomstreams nodig, dus je zult ook met randomgetallen moeten werken (zie opdracht 5.18). We geven eerst wat achtergrondinformatie over de one-time pad encryptiemethode.

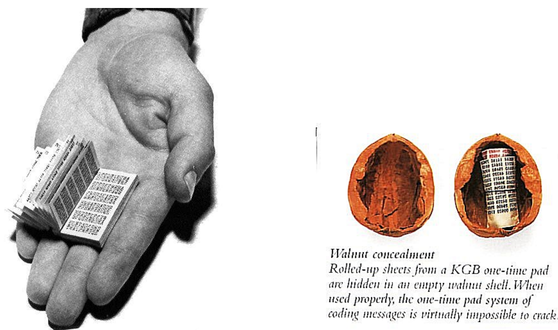
De one-time pad methode is een encryptiemethode die in het ideale geval onbreekbaar is. Het idee is dat als je kunt beschikken over *zuivere* randomstreams van getallen, je deze kunt gebruiken om een boodschap zodanig te manipuleren (m.b.v. het XOR-en van bitjes of optellen modulo 128) dat deze enkel nog ambigue informatie bevat. Om de boodschap terug te krijgen heb je een *exacte kopie* nodig van de zuivere randomstream van getallen om via de inverse bewerking de originele boodschap terug te krijgen. In

[1] bron: `http://en.wikipedia.org/wiki/One-time_pad`

theorie is deze methode onkraakbaar. In de praktijk kleven er echter veel haken en ogen aan:

- Het genereren van zuivere randomstreams is geen sinecure: de meeste pseudo-random generatoren die je in programmeertalen vindt zijn niet zuiver.
- Elke randomstream mag slechts één maal gebruikt worden, en moet daarna vernietigd worden. Het vernietigen van data op bijvoorbeeld gangbare computer media is een moeilijk probleem.
- De zender en de ontvanger hebben een exacte kopie nodig van de randomstream. Dit betekent dat het systeem kwetsbaar is voor beschadiging, diefstal en kopieëren.

Ondanks bovenstaande bezwaren zijn OTP encryptie methoden in de praktijk gebruikt door bijvoorbeeld de KGB (de voormalige inlichtingendienst van de voormalige Sovjet-Unie). Deze gebruikten zuivere randomstreams gedrukt op zeer klein formaat zodat deze goed verstopt konden worden (zie figuur 7.1):



Figuur 7.1: Zuivere randomstreambronnen voor OTP encryptie, gebruikt voor spionage activiteiten. Bron: `http://www.ranum.com/security/computer_security/-papers/otp-faq/` (afbeelding links); `http://home.egge.net/~savory/chiffre9.htm` (afbeelding rechts).

In deze opdracht implementeer je de OTP encryptie-methode voor het versleutelen van bestanden. Omdat we geen zuivere randomstreams hebben, werk je met de *pseudo-random generatoren* uit opdracht 5.18. Kies voor je eigen OTP encryptiemethode een integer $\mathcal{R}$ die je geheim houdt (dit moet doorgaan voor je zuivere randomstream). Gebruik deze $\mathcal{R}$ als `RandomSeed` waarde voor het genereren van de reeks van pseudo-random getallen $r_0, r_1, \ldots$.

One-time pad versleutelen en ontcijferen

Het versleutelen werkt als volgt. Laat A het invoertekstbestand zijn dat je wilt versleutelen naar een uitvoertekstbestand B. Lees nu, stap voor stap de ASCII tekens $a_0, a_1, \ldots, a_N$ in van A, en schrijf dat achtereenvolgens weg naar B als $b_0, b_1, \ldots, b_N$.

$$b_i = \begin{cases} a_i & \textbf{als } a_i < 32 \\ (a_i - 32 + r_i) mod (128 - 32) + 32 & \textbf{als } a_i \geq 32 \end{cases}$$

Deze tweedeling zorgt ervoor dat non-printable ASCII tekens ($0 \leq a_i < 32$) onveranderd blijven, en dat de overige ASCII tekens ($32 \leq a_i < 128$) wel versleuteld worden.

Ontcijferen werkt hetzelfde, behalve dat nu B het invoer-bestand is, en dat nu niet de code van b_i met r_i verhoogd dient te worden, maar *verlaagd*. Let op negatieve waardes!

Schrijf een console I/O programma dat de invoer `otp` A B accepteert waarbij A en B de padnamen van tekstbestanden zijn, en die de inhoud van A versleuteld wegschrijft naar B. De invoer `pto` A B doet het omgekeerde: het ontsleutelt bestand A en schrijft het weg naar B. Test je programma op een representatief invoerbestand A, dat je versleutelt en vervolgens ontcijfert. Controleer of het ontcijferde bestand identiek is aan A.

Maak gebruik van de module `SimpleFileIO` die je ontwikkeld hebt in opdracht 7.6.

7.14 Huffman codering

Main module:	*Huffman.icl*
Environment:	*Object IO*

In deze opdracht maak je een compressie- en decompressieprogramma dat gebruik maakt van *Huffman-codering*[2]. Huffman-codering is in 1952 bedacht door David Huffman[3]. Tekens in een ASCII bestand worden elk gerepresenteerd door een even lange code. Dat betekent dat zeldzaam voorkomende tekens evenveel representatie-ruimte in beslag nemen als veel voorkomende tekens. Het basis-idee van Huffman codering is om veelvoorkomende tekens te vervangen door een korte code, en weinig voorkomende tekens door een langere code. Dat betekent dat deze methode afhankelijk is van een z.g.n. *frequentietabel*. Deze is verschillend per taal (bijv. Nederlands vs. Engels). In het algemeen zul je dus ook de frequentietabel moeten opslaan in de gecomprimeerde data. Tevens betekent dit dat het resultaatbestand in *binair* formaat is. Als het toepassingsgebied vast ligt, is het niet nodig om een frequentietabel op te slaan, maar kun je een vaste tabel afspreken. Voor het Nederlands is de frequentietabel uit figuur 7.2 bruikbaar. Deze tabel kun je vinden in het bestand `freqNL.txt`.

	%		%		%		%		%		%		%
E	18,91	N	10,03	A	7,49	T	6,79	I	6,50	R	6,41	O	6,06
D	5,93	S	3,73	L	3,57	G	3,40	V	2,85	H	2,38	K	2,25
M	2,21	U	1,99	B	1,58	P	1,57	W	1,52	J	1,46	Z	1,39
C	1,24	F	0,81	X	0,04	Y	0,03	Q	0,01				

Figuur 7.2: De frequentietabel van het Nederlandse alfabet. (`http://nl.wikipedia.org/wiki/Huffmancodering`)

Een andere consequentie van het gebruik van codes van verschillende lengte is dat de code van het ene teken geen *prefix* mag zijn van de code van een ander teken. Deze eis heet *prefix free code*. Het algoritme dat door Huffman bedacht is om *prefix free code* te genereren staat goed uitgelegd op:

[2] Bron: Wikipedia; `http://nl.wikipedia.org/wiki/Huffmancodering`

[3] "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., sept 1952, pp. 1098-1102.

`http://nl.wikipedia.org/wiki/Huffmancodering` (bondige uitleg)
`http://en.wikipedia.org/wiki/Huffman_coding` (uitgebreidere uitleg)

Bestudeer dit algoritme.

In deze opdracht maak je een console I/O programma dat de commando's implementeert die hieronder staan beschreven. Gebruik voor het inlezen en wegschrijven van bestanden de module `SimpleFileIO` uit opdracht 7.6.

Frequentietabel → Huffman-code tabel

Het commando `tabel freq`*XX*`.txt` leest een frequentietabel `freq`*XX*`.txt` in (in hetzelfde formaat als gebruikt voor `freqNL.txt` op de site), en genereert een Huffman-code tabel en schrijft deze naar een bestand met de naam `huffman`*XX*`.txt`. Om te controleren of je oplossing goed is, staat op de site ook de oplossing voor de Nederlandse frequentietabel, dus onder de naam `huffmanNL.txt`.

Huffman-code tabel × tekstbestand → codering

Het commando `codeer huffman`*XX*`.txt` *file*`.txt` leest de code tabel uit bestand `huffman`*XX*`.txt` in en een tekstbestand *file*`.txt` dat gecodeerd dient te worden. De inhoud van *file*`.txt` wordt gecodeerd met behulp van de `huffman`*XX*`.txt` code. Het resultaat wordt bewaard in een nieuw bestand met de naam *file*`.huf`.

Huffman-code tabel × codering → tekstbestand

Het commando `decodeer huffman`*XX*`.txt` *file*`.huf` leest de code tabel uit bestand `huffman`*XX*`.txt` in en een gecodeerd *file*`.huf` bestand dat gedecodeerd dient te worden. De inhoud van *file*`.huf` wordt gedecodeerd met behulp van de `huffman`*XX*`.txt` code. Het resultaat wordt bewaard in een nieuw bestand met de naam *file*`.txt`.

Hoofdstuk 8

Boomstructuren

Tot nu toe hebben we, afgezien van lijsten, in de opdrachten geen gebruik gemaakt van recursieve data structuren. Recursieve data structuren kunnen met algebraïsche types en records gemaakt worden. Lijsten zijn een bijzonder geval. Recursieve data structuren worden ook wel *boomstructuren* genoemd omdat ze een beginpunt hebben die *wortel* (*root*) genoemd wordt en een vertakkingsstructuur (*branching structure*). De principes die je gebruikt hebt om met lijsten te werken zijn ook van toepassing op boomstructuren. Dat betekent dat je ze onbegrensd groot kunt maken, en dat je er berekeningen in kunt wegzetten die je wellicht in de toekomst wilt laten uitrekenen.

8.1 Binaire bomen(*gebruikt in 8.2,8.3,8.20*)

Main module:	*BinTree.icl*
Environment:	*StdEnv*

Neem de volgende type definitie van een binaire boom (diktaat, sectie 3.6.1, blz. 71):

```
::  Tree a = Node a (Tree a) (Tree a) | Leaf
```

Tekenen Teken de bomen `t0` ... `t7` die als volgt ontstaan:

```
t0 = Leaf
t1 = Node 4 t0 t0
t2 = Node 2 t0 t1
t3 = Node 5 t2 t0
t4 = Node 5 t2 t2
t5 = Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
t6 = Node 1 (Node 2 (Node 3 (Node 4 Leaf Leaf) Leaf) Leaf) Leaf
t7 = Node 4 (Node 1 Leaf Leaf) (Node 5 (Node 2 Leaf Leaf) Leaf)
```

Boom metrieken Implementeer de functies `nodes`, `leaves` en `diepte`: `nodes` telt het aantal `Nodes`, `leaves` telt het aantal `Leafs`, en `diepte` is het *maximum* aantal `Nodes` dat je tegenkomt als je van de *wortel* van een boom naar een van zijn *bladen* loopt. Dus:

`nodes t0`	=	0	`leaves t0`	=	1	`diepte t0`	=	0
`nodes t1`	=	1	`leaves t1`	=	2	`diepte t1`	=	1
`nodes t2`	=	2	`leaves t2`	=	3	`diepte t2`	=	2
`nodes t3`	=	3	`leaves t3`	=	4	`diepte t3`	=	3
`nodes t4`	=	5	`leaves t4`	=	6	`diepte t4`	=	3
`nodes t5`	=	4	`leaves t5`	=	5	`diepte t5`	=	4
`nodes t6`	=	4	`leaves t6`	=	5	`diepte t6`	=	4
`nodes t7`	=	4	`leaves t7`	=	5	`diepte t7`	=	3

Aantal nodes versus aantal leaves versus diepte De metrieken van binaire bomen hebben onderlinge relaties. Toon deze aan voor iedere eindige boom t met behulp van inductie naar de structuur van t.

1. `nodes` $t =$ `leaves` $t - 1$.
2. `leaves` $t \leq 2^{\texttt{diepte}\ t}$.

8.2 Bomen tonen*(gebruikt in 8.3)*

Main module:	*BinTreePrint.icl*
Environment:	*StdEnv*

In opdracht 8.1 heb je met de hand een aantal bomen getekend. Om te kunnen controleren of je daadwerkelijk de juiste bomen getekend hebt, is het handiger als je daar een functie voor hebt. In deze opdracht ontwikkel je twee algoritmen om dat te doen:

1. een functie `indentTree` die de boom afdrukt door systematisch *in te springen* afhankelijk van de diepte van de af te drukken knoop en diens onderbomen;
2. een functie `tree2D` die de module `TextCompose` uit opdracht 6.8 gebruikt om een meer traditionele weergave van de boomstructuur te berekenen.

De module `BinTreePrint` exporteert deze functionaliteit als instantie van de standaard `toString` klasse van Clean:

```
instance toString (Tree a) | toString a where
    toString tree = indentTree tree
    toString tree = tree2D     tree
```

Kies de instantie die je wilt implementeren, testen of gebruiken door de ongewenste instantie in commentaar te zetten. In deze opdracht kun je hem gebruiken om te beoordelen of je de juiste plaatjes getekend hebt in opdracht 8.1.

8.2.1 Afdrukken met inspringen

De functie `indentTree` krijgt een binaire boomstructuur, zoals bijvoorbeeld `t7` uit opdracht 8.1, en drukt deze als volgt af:

```
(Node 4
    (Node 1
        Leaf
        Leaf
    )
    (Node 5
        (Node 2
            Leaf
            Leaf
        )
        Leaf
    )
)
```

Aan deze uitvoer zie je dat de afstand van de knopen tot de linkerkolom lineair afhangt van hun diepte in de boom. De *root* node wordt op de linkerkolom gezet, diens kinder-knopen één niveau diep ingesprongen, daar weer de kinderen van twéé niveaus ingesprongen, enz. De haakjesstructuur geeft verder aan welke elementen bij elkaar horen.

Implementeer de functie `indentTree` die een binaire boomstructuur afdrukt zoals hierboven beschreven. Test je functie met de bomen `t0` tot en met `t7` uit opdracht 8.1.

8.2.2 2D afdrukken

De uitvoer die gegenereerd wordt door `indentTree` is niet altijd even leesbaar omdat knopen van dezelfde diepte erg ver van elkaar getoond kunnen worden. Het is soms duidelijker om deze knopen *naast elkaar* te tonen. De functie `tree2D` krijgt een binaire boomstructuur, zoals bijvoorbeeld `t7` uit opdracht 8.1, en drukt deze als volgt af:

```
          4
          |
    --------------
    |            |
    1            5
    |            |
  -----      --------
  |   |      |      |
Leaf Leaf    2     Leaf
             |
           -----
           |   |
         Leaf Leaf
```

Aan deze uitvoer zie je dat knopen van dezelfde diepte naast elkaar getoond worden. De twee onderbomen van een knoop worden als twee *tekstblokken* naast elkaar gezet. Dit suggereert dat de module `TextCompose` die je ontwikkeld hebt in opdracht 6.8 hier goed toegepast kan worden.
Een uitdagend aspect van het tekenen van boomstructuren op deze manier is dat de afmeting van de linker-onderboom in het algemeen niet gelijk is aan die van de rechter-onderboom. Dit heeft gevolgen voor het bepalen van de positie van de bijbehorende knoop.

Implementeer de functie `tree2D` die een binaire boomstructuur afdrukt zoals hierboven beschreven. Test je functie met de bomen `t0` tot en met `t7` uit opdracht 8.1.

8.3 Binaire zoekbomen

Main module:	*BinSearchTree.icl*
Environment:	*StdEnv*

Neem de type definitie van een binaire zoekboom (diktaat 3.6.2, blz.73).

insert Bestudeer de functie `insertTree` (diktaat 3.6.2, blz.73). Teken de bomen $z_0 \ldots z_7$:

```
z0 = Leaf
z1 = insertTree 50 z0
z2 = insertTree 10 z1
z3 = insertTree 75 z2
```

```
z4 = insertTree 80 z3
z5 = insertTree 77 z4
z6 = insertTree 10 z5
z7 = insertTree 75 z6
```

Als je opdracht 8.2 gemaakt hebt, controleer dan je antwoorden met behulp van de door jou geïmplenteerde `toString` instantie voor binaire bomen.

delete Bestudeer de functie `deleteTree` (diktaat 3.6.4, blz.75). Teken de boom z_8:

```
z8 = deleteTree 50 z7
```

Als je opdracht 8.2 gemaakt hebt, controleer dan je antwoord met behulp van de door jou geïmplenteerde `toString` instantie voor binaire bomen.

geordend Een binaire boom is *geordend* als voor elke knoop met element x geldt dat alle elementen in de linker-onderboom *kleiner of gelijk* zijn dan x **en** dat alle elementen in de rechter-onderboom *groter* zijn dan x. De ordening op elementen is bepaald door de `<` instantie van het element type. Schrijf de functie `is_geordend` die alleen `True` oplevert als een boom *geordend* is. Dus (met de bomen $t_0 \dots t_7$ uit opdracht 8.1):

is_geordend t0	=	True	is_geordend t4	=	False
is_geordend t1	=	True	is_geordend t5	=	True
is_geordend t2	=	True	is_geordend t6	=	False
is_geordend t3	=	True	is_geordend t7	=	False

gebalanceerd Een binaire boom is *gebalanceerd* als voor elke knoop in de boom geldt dat het verschil tussen de *diepte* (zie opdracht 8.1) van diens linker- en rechter-onderboom ten hoogste 1 is. Schrijf de functie `is_gebalanceerd` die alleen `True` oplevert als een boom *gebalanceerd* is. Dus (met de bomen $t_0 \dots t_7$ uit opdracht 8.1):

is_gebalanceerd t0	=	True	is_gebalanceerd t4	=	True
is_gebalanceerd t1	=	True	is_gebalanceerd t5	=	False
is_gebalanceerd t2	=	True	is_gebalanceerd t6	=	False
is_gebalanceerd t3	=	False	is_gebalanceerd t7	=	True

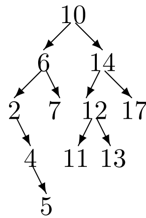
8.4 Bomen aflopen

Main module:	*BinTreeTraversal.icl*
Environment:	*StdEnv*

In deze opgave maken we gebruik van het volgende data type voor binaire zoekbomen:

```
::  Tree a = Node a (Tree a) (Tree a) | Leaf
```

Een voorbeeld van een binaire zoekboom is (`Node`s en `Leaf`s zijn niet getekend):



Zoekbomen kunnen op verschillende manieren omgezet worden naar lijsten. Schrijf de drie recursieve functies `lijstOplopend`, `lijstAflopend` en `lijstNaarBladen`, alle van type `(Tree a) -> [a]`. De eisen van de functies zijn:

- `lijstOplopend`: levert de elementen in oplopende volgorde van grootte op, te beginnen met het kleinste element.

  Bijv.: de boom hierboven wordt omgezet naar `[2,4,5,6,7,10,11,12,13,14,17]`.

  NB: implementeer `lijstOplopend` niet als `reverse lijstAflopend`.

- `lijstAflopend`: levert de elementen in aflopende volgorde van grootte op, te beginnen met het grootste element.

  Bijv.: de boom hierboven wordt omgezet naar `[17,14,13,12,11,10,7,6,5,4,2]`.

  NB: implementeer `lijstAflopend` niet als `reverse lijstOplopend`.

- `lijstNaarBladen`: levert de elementen op in volgorde van afstand tot de wortel van de boom. Eerst de wortel, dan alle elementen daaronder (van links naar rechts), dan alle elementen daar weer onder (van links naar rechts), enzovoort.

  Bijv.: de boom hierboven wordt omgezet naar `[10,6,14,2,7,12,17,4,11,13,5]`.

8.5 Map en fold over bomen

Main module:	*BinTreeMapEnFold.icl*
Environment:	*StdEnv*

Beschouw het volgende type voor binaire bomen en een aantal functies:

```
::  BTree a = Tip a | Bin (BTree a) (BTree a)

mapbtree :: (a -> b) (BTree a) -> BTree b
mapbtree f (Tip a)     = Tip (f a)
mapbtree f (Bin t1 t2) = Bin (mapbtree f t1) (mapbtree f t2)

foldbtree :: (a a -> a) (BTree a) -> a
foldbtree f (Tip a)     = a
foldbtree f (Bin t1 t2) = f (foldbtree f t1) (foldbtree f t2)

const :: a b -> a
const x _ = x
```

Geef van elk van de onderstaande functies het meest algemene type en vertel wat de functies uitrekenen:

```
f1 = foldbtree (+)
f2 = foldbtree (+) o (mapbtree (const 1))
f3 = foldbtree (\x y -> 1 + max x y) o (mapbtree (const 0))
f4 = foldbtree (++) o (mapbtree (\x -> [x]))
```

8.6 Gegeneraliseerde bomen*(gebruikt in 8.7,8.8)*

Main module:	*GenTree.icl*
Environment:	*StdEnv*

In deze opdracht ga je aan de slag met gegeneraliseerde bomen:

```
::  GenTree a b    = Node a [GenTree a b] | Leaf b
```

Ze zijn algemener dan de binaire bomen die we tot nu toe gezien hebben omdat de types van de elementen in de `Node`s anders kunnen zijn dan die in de `Leaf`s. Bovendien kan een knoop 0 of meer onderbomen hebben. Schrijf voor deze bomen de volgende functies:

```
::  Either a b  = This a | That b

root         ::   (GenTree a b) -> Either a b
trees        ::   (GenTree a b) -> [GenTree a b]

isNodeMember :: a (GenTree a b) -> Bool | Eq a
isLeafMember :: b (GenTree a b) -> Bool | Eq b
allNodes     ::   (GenTree a b) -> [a]
allLeaves    ::   (GenTree a b) -> [b]
allMembers   ::   (GenTree a a) -> [a]

map2         :: (a -> c,b -> d) (GenTree a b) -> GenTree c d
```

met de volgende betekenissen:

- `root` levert het element van de wortel op. Het type hangt af van het feit of het een knoop-element is of een blad-element. Daarom wordt het `Either` type gebruikt.
- `trees` levert de direkte onderbomen van de boom op. Van een `Leaf` is deze leeg, van een (`Node _ ts`) is het `ts`.
- `is(Node/Leaf)Member` test of het eerste argument voorkomt als knoop-element (blad-element).
- `all(Nodes/Leaves)` levert alle knoop-elementen (blad-elementen) op. Als knopen en bladen hetzelfde type hebben, dan kan `allMembers` gebruikt worden om ze alle op te leveren.
- `map2 (f,g)` past `f` toe op alle knoop-elementen, en `g` toe op alle blad-elementen.

8.7 Gegeneraliseerde bomen tonen(*gebruikt in 8.8*)

Main module:	*GenTreePrint.icl*
Environment:	*StdEnv*

Ontwikkel in deze opdracht op analoge wijze als beschreven in opdracht 8.2 twee mogelijke algoritmen om gegeneraliseerde bomen af te drukken:

1. `indentTree` die middels systematisch inspringen knopen van dezelfde diepte afdrukt met dezelfde mate van inspringen, en
2. `tree2D` die met behulp van `TextCompose` uit opdracht 6.8 knopen van dezelfde diepte *naast elkaar* afdrukt.

Laten we als voorbeeld de volgende, ietwat vreemd-gevormde boomstructuur nemen:

```
:: Void   = Void
instance toString Void where toString _ = "."

pyramid :: Int -> GenTree Int Void
pyramid 1 = Leaf Void
pyramid n = Node n [pyramid (n-1) \\ i <- [1 .. n]]
```

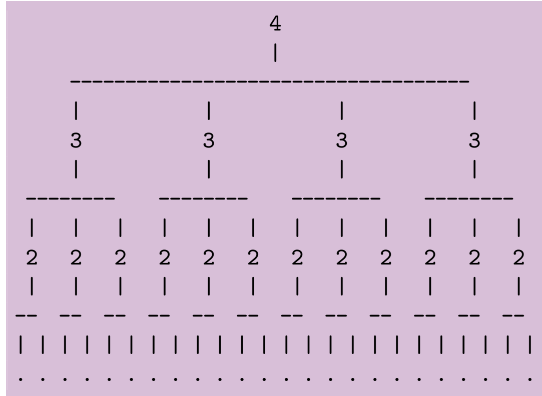
Deze gegeneralizeerde boomstructuur heeft onderbomen die telkens één onderboom minder hebben. Onderaan de boom vind je de bladeren van de boom die verder geen nuttige informatie bevatten. Dat wordt weergegeven door het `Void` type dat als een eenvoudige . afgedrukt wordt.

8.7.1 Afdrukken met inspringen

```
Node 3
    [Node 2
        [Leaf .
        ,Leaf .
        ]
    ,Node 2
        [Leaf .
        ,Leaf .
        ]
    ,Node 2
        [Leaf .
        ,Leaf .
        ]
    ]
```

Ontwikkel op soortgelijke wijze als in opdracht 8.2.1 een algoritme dat hetzelfde doet voor gegeneraliseerde bomen. De uitvoer voor bijvoorbeeld `pyramid 3` staat hier links. De uitvoer voor `pyramid 4` laten we hier niet zien: deze is meer geschikt als alternatieve versiering van toiletpapier, probeer het maar eens zelf.

8.7.2 2D afdrukken



Ontwikkel op soortgelijke wijze als in opdracht 8.2.2 een algoritme dat hetzelfde doet voor gegeneraliseerde bomen. De uitvoer voor bijvoorbeeld `pyramid 4` staat hier links. In tegenstelling tot het 'inspring'-algoritme levert dit een beter verdeelde uitvoer op.

8.8 Stambomen(gebruikt in 8.9)

Main module:	*StamBoom.icl*
Environment:	*StdEnv*

Stambomen kun je als volgt definiëren als een specialisatie van gegeneraliseerde bomen zoals gedefinieerd in opdracht 8.6:

```
:: FamilyTree  :== GenTree Couple Single
:: Couple      =   Couple Person Person
:: Single      =   Single Person
:: Person      =   Person DateOfBirth Gender String
:: Gender      =   Male | Female
:: DateOfBirth =   DoB Year Month Day
:: Year        :== Int
:: Month       :== Int
:: Day         :== Int
```

We gaan voor het gemak uit van een erg conservatieve situatie: mensen krijgen pas kinderen nadat ze getrouwd zijn, en dus een paar vormen. Mensen hertrouwen niet. Een persoon p wordt geboren als een (`Single` p) en gaat na trouwen met persoon q door het leven als het paar (`Couple` p q). De persoon q is *aangetrouwd* en hoort niet bij de nazaten (*offspring*) van iemand.

Schrijf voor stambomen de volgende functies:

```
okFamilyTree ::               FamilyTree -> Bool
rootAncestor ::               FamilyTree -> Person
inFamilyTree :: Person        FamilyTree -> Bool
marry        :: Person Person FamilyTree -> FamilyTree
addChild     :: Person Couple FamilyTree -> FamilyTree
children     :: Person        FamilyTree -> [Person]
offspring    :: Person        FamilyTree -> [Person]
```

- `okFamilyTree` t controleert of een stamboom t voldoet aan de volgende criteria:
  - kinderen zijn jonger dan hun ouders;

  – broers/zussen moeten geordend zijn van oudste naar jongste;
  – een persoon mag niet meerdere keren voorkomen in dezelfde stamboom.

- `rootAncestor` t levert de stamoudste van t op.
- `inFamilyTree` p t test of een gegeven persoon p (afstammeling dan wel aangetrouwd) voorkomt in stamboom t.
- `marry` p q t maakt van een ongetrouwd persoon (`Single` p) in t het getrouwde paar (`Couple` p q). Als persoon p niet voorkomt in t, of al getrouwd is, dan blijft t onveranderd.
- `addChild` p c t voegt een kind p toe aan een stel ouders c in t. Als het paar c niet voorkomt in t, dan blijft t onveranderd.
- `children` p t levert alle kinderen op van een gegeven persoon p in t; p kan zowel een afstammeling als aangetrouwd persoon zijn. Als p niet voorkomt in t, dan worden geen personen opgeleverd.
- `offspring` p t levert alle afstammelingen van een gegeven persoon p in t op; p kan zowel een afstammeling als aangetrouwd persoon zijn, maar diens afstammelingen zijn nooit aangetrouwde personen.

Bedenk zelf wat een handige volgorde is om bovenstaande functies te definiëren. Als je opdracht 8.6 gemaakt hebt, is het verstandig zo veel mogelijk gebruik te maken van de functionaliteit die je daar geïmplementeerd hebt.

8.9 Stambomen afdrukken

Main module:	*StamboomPrint.icl*
Environment:	*StdEnv*

Gebruik de afdrukmogelijkheden van gegeneraliseerde boomstructuren uit opdracht 8.7 om ook stambomen af te kunnen drukken. Dit leidt al snel tot brede uitvoer. Je kunt de uitvoer eenvoudig naar een tekstbestand *redirecten* vanaf de *command promt*:

```
> StamboomPrint.exe >uitvoer.txt
```

8.10 AVL-bomen

Main module:	*AVLTreeTest.icl*
Environment:	*StdEnv*

Implementeer de module `StdAVLTree.icl` die *AVL* bomen realiseert zoals uitgewerkt op het college. De *AVL* boom moet in de boom zelf de diepte opslaan, zodat deze niet elke keer opnieuw berekend moet worden. De bijbehorend `StdAVLTree.dcl` ziet er als volgt uit:

```
definition module StdAVLTree

import StdClass

::  AVLTree a
```

```
mkAVLLeaf       ::                  AVLTree a
mkAVLNode       :: a            -> AVLTree a
isMemberAVLTree :: a (AVLTree a) -> Bool        | Eq, Ord a
insertAVLTree   :: a (AVLTree a) -> AVLTree a   | Eq, Ord a
deleteAVLTree   :: a (AVLTree a) -> AVLTree a   | Eq, Ord a
isAVLTree       ::   (AVLTree a) -> Bool        | Eq, Ord a
```

De functies `mkAVL(Leaf/Node)` creëren een lege *AVL* boom / knoop met gegeven waarde en lege onderbomen; de functie `isMemberAVLTree` test of een gegeven waarde in de *AVL* boom voorkomt; `insertAVLTree` die een element in de *AVL* boom toevoegt; `deleteAVLTree` die een element uit de *AVL* boom verwijdert; `isAVLTree` tenslotte is een predicaat dat test of de gegeven *AVL* boom correct geconstrueerd is. Deze laatste functie hoort normaal gesproken niet in de abstracte signatuur van correct geconstrueerde *AVL* bomen, maar is handig tijdens het ontwikkelen van je functies om te bepalen of er geen fouten zijn opgetreden.

Optioneel

Als je opdracht 8.2 gemaakt hebt, die binaire bomen kan afdrukken, dan is dat ook een handig hulpmiddel om te kunnen *zien* of je *AVL* bomen correct geconstrueerd zijn.

8.11 Propositie-logica

Main module:	*PropositieLogica2.icl*
Environment:	*StdEnv*

8.11.1 Grondtermen

Termen in propositie-logica zonder variabelen (zogenaamde *grondtermen*) zijn als volgt inductief opgebouwd:

1. *waar* en *onwaar* zijn termen;

2. als t een term is, dan is $niet(t)$ ook een term;

3. als t_1 en t_2 termen zijn, dan zijn $en(t_1, t_2)$ en $of(t_1, t_2)$ ook termen.

Een voorbeeld van een term die volgens deze drie regels is opgebouwd is:

$$niet(of(en(onwaar, waar), of(onwaar, en(waar, waar))))$$

Aan iedere term t kun je een *boolean* waarde toekennen. Dit gebeurt met behulp van een interpretatie $[\![t]\!]$. Deze is ook inductief gedefinieerd:

A. $[\![waar]\!] = true$ en $[\![onwaar]\!] = false$;

B. De waarde van $niet(t)$ is de ontkenning van de waarde van t.
Dus: $[\![niet(t)]\!] = \neg[\![t]\!]$;

C. De waarde van $en(t_1, t_2)$ is alleen waar als t_1 én t_2 waar zijn.
Dus: $[\![en(t_1, t_2)]\!] = [\![t_1]\!] \wedge [\![t_2]\!]$.
De waarde van $of(t_1, t_2)$ is alleen onwaar als t_1 én t_2 onwaar zijn.
Dus: $[\![of(t_1, t_2)]\!] = [\![t_1]\!] \vee [\![t_2]\!]$.

Bovenstaande term heeft de volgende interpretatie:

$$
\begin{array}{rl}
& [\![\mathit{niet}(\mathit{of}(\mathit{en}(\mathit{onwaar}, \mathit{waar}), \mathit{of}(\mathit{onwaar}, \mathit{en}(\mathit{waar}, \mathit{waar}))))]\!] \\
= & \neg [\![\mathit{of}(\mathit{en}(\mathit{onwaar}, \mathit{waar}), \mathit{of}(\mathit{onwaar}, \mathit{en}(\mathit{waar}, \mathit{waar})))]\!] \\
= & \neg ([\![\mathit{en}(\mathit{onwaar}, \mathit{waar})]\!] \vee [\![\mathit{of}(\mathit{onwaar}, \mathit{en}(\mathit{waar}, \mathit{waar}))]\!]) \\
= & \neg (([\![\mathit{onwaar}]\!] \wedge [\![\mathit{waar}]\!]) \vee ([\![\mathit{onwaar}]\!] \vee ([\![\mathit{en}(\mathit{waar}, \mathit{waar})]\!]))) \\
= & \neg (([\![\mathit{onwaar}]\!] \wedge [\![\mathit{waar}]\!]) \vee ([\![\mathit{onwaar}]\!] \vee ([\![\mathit{waar}]\!] \wedge [\![\mathit{waar}]\!]))) \\
= & \neg ((\mathit{false} \wedge \mathit{true}) \vee (\mathit{false} \vee (\mathit{true} \wedge \mathit{true})))
\end{array}
$$

Dit is een 'normale' *booleaanse* expressie die als volgt uitgerekend kan worden:

$$
\begin{array}{rl}
= & \neg (\underline{(\mathit{false} \wedge \mathit{true})} \vee (\mathit{false} \vee (\mathit{true} \wedge \mathit{true}))) \\
= & \neg (\underline{\mathit{false} \vee (\mathit{false} \vee (\mathit{true} \wedge \mathit{true}))}) \\
= & \neg (\underline{\mathit{false} \vee (\mathit{true} \wedge \mathit{true})}) \\
= & \neg (\underline{\mathit{true} \wedge \mathit{true}}) \\
= & \underline{\neg (\mathit{true})} \\
= & \mathit{false}
\end{array}
$$

Representatie Ontwikkel met gebruik van algebraïsche types een geschikte representatie voor de hierboven beschreven termen uit de propositie-logica. Noem deze `PropL`.

Printen Schrijf een *instance* van de `toString` *type class* voor het type `PropL`. Deze zet dus `PropL` waarden om in een `String`.

Evaluatie Schrijf een functie `eval1` die een `PropL` term interpreteert zoals hierboven beschreven, en de bijbehorende `Bool` waarde oplevert.

8.11.2 Variabelen

We voegen nu *variabelen* toe aan de inductieve definitie van termen. Er komt de volgende regel bij:

4. de *variabelen* $v_i (i \in \{1, 2, 3, \ldots\})$ zijn termen.

Een voorbeeld van een term *met* variabelen is:

$$\mathit{niet}(\mathit{of}(\mathit{en}(v_1, v_2), \mathit{of}(v_2, \mathit{en}(v_2, \mathit{waar}))))$$

Om een term t met variabelen te interpreteren heb je extra informatie nodig. Een *valuatie* van variabelen is een afbeelding die aan elke variabele één waarde toekent. Voor bovenstaand voorbeeld is $\mathit{val} = \{(v_1, \mathit{true}), (v_2, \mathit{false})\}$ een mogelijke valuatie van de variabelen $\{v_1, v_2\}$. De interpretatie $[\![t]\!]$ breiden we uit met deze valuatie: $[\![t]\!]_{\mathit{val}}$ en de volgende regel:

D. De waarde van v_i, gegeven een valuatie $\mathit{val} = \{\ldots (v_i, b_i) \ldots\}$ is b_i. We noteren dit met $\mathit{val}(v_i)$. Dus: $[\![v_i]\!]_{\mathit{val}} = \mathit{val}(v_i)$.

In de regels **A** tot en met **C** wordt de valuatie alleen maar ongewijzigd doorgegeven.

Representatie Pas `PropL` zodanig aan dat variabelen gerepresenteerd kunnen worden.

Printen Pas de `PropL` *instance* van de `toString` *type class* aan zodat variabelen ook naar `String` omgezet worden.

Evaluatie Schrijf de functie `eval2` die `eval` uitbreidt met een valuatie om termen met variabelen te kunnen interpreteren. Bedenk zelf een geschikte representatie voor een valuatie en noem dit type `Valuatie`.

Variabelen filter Schrijf een functie `vars` die een term t van type `PropL` krijgt en die alle variabelen uit t in een lijst oplevert. Deze lijst mag geen duplicaten bevatten. De variabelen van bovenstaand voorbeeld zijn v_1 en v_2.

Valuaties Schrijf een functie `vals` die, gegeven een lijst van variabelen *zonder duplicaten*, alle mogelijke valuaties voor die variabelen oplevert.

Wanneer waar? Gebruik tenslotte bovenstaande functies om een functie `truths` te schrijven die, gegeven een term t van type `PropL`, alle valuaties berekent die t waar maken. Voor de term die hierboven gedefinieerd is zijn dat $\{(v_1, true), (v_2, false)\}$ en $\{(v_1, false), (v_2, false)\}$.

8.12 Drie-waardige propositie-logica

Main module:	*PropositieLogica3.icl*
Environment:	*StdEnv*

In de *drie-waardige* propositie logica worden termen op dezelfde manier opgebouwd als in opdracht 8.11, die over *twee-waardige* logica gaat. Het verschil is dat de variabelen *drie* waarden kunnen aannemen, namelijk *waar*, *onwaar* en *onbekend*. Rekenen met de waarden *waar*, *onwaar* en de logische operaties blijft ongewijzigd. Rekenen met *onbekend* gaat als volgt:

t_1	t_2	$and(t_1, t_2)$	$or(t_1, t_2)$	$not(t_1)$
	waar	*waar*	*waar*	
waar	*onwaar*	*onwaar*	*waar*	*onwaar*
	onbekend	*onbekend*	*waar*	
	waar	*onwaar*	*waar*	
onwaar	*onwaar*	*onwaar*	*onwaar*	*waar*
	onbekend	*onwaar*	*onbekend*	
	waar	*onbekend*	*waar*	
onbekend	*onwaar*	*onwaar*	*onbekend*	*onbekend*
	onbekend	*onbekend*	*onbekend*	

Ontwikkel datastructuren en functies zoals in opdracht 8.11. Pas nu echter de datastructuren en functies zodanig aan dat deze voor zowel twee-waardige als drie-waardige propositie-logica werken. Gebruik hiervoor overloading.

8.13 Expressies refactoren(*gebruikt in 8.21*)

Main module:	*RefactorX.icl*
Environment:	*StdEnv*

In deze opdracht bewerk je representaties van expressies van de volgende vorm:

E_1 `= ( let x = 42 - 3 in x / 0 ) + ( let y = 6 in y * y )`
// syntactisch correcte Clean expressie met run-time error

E_2 = **let** x = 42 **in** x + (**let** x = 58 **in** x)
// syntactisch correcte Clean expressie met resultaat 100
E_3 = **let** x = 1 **in** **let** y = 2 **in** **let** x = 3 **in** 4
// syntactisch correcte Clean expressie met resultaat 4
E_4 = **let** x = 1 **in** x + y *// syntactisch incorrecte Clean expressie (y onbekend)*
E_5 = (**let** x = 1 **in** x) * x *// syntactisch incorrecte Clean expressie (buitenste x onbekend)*

Deze expressies kun je representeren met behulp van de volgende types:

```
:: Expr     =   NR  Int
            |   VAR Name
            |   OP  Expr Operator Expr
            |   LET Name Expr Expr
:: Name     :== String
:: Operator =   PLUS | MIN | MUL | DIV
```

Een expressie kan dus een getal n zijn (gerepresenteerd met `(NR` n`)`), een variabele met naam x (gerepresenteerd met `(VAR` x`)`), een rekenkundige bewerking (e_1 op e_2) (gerepresenteerd met `(OP` e_1 op e_2`)`), of een declaratie (**let** n = e_1 **in** e_2) (gerepresenteerd met `(LET` n e_1 e_2`)`). De bovenstaande expressies E_i worden dan als volgt gerepresenteerd als `Expr` waarden `E`i:

```
E1 = OP (LET "x" (OP (NR 42) MIN (NR 3)) (OP (VAR "x") DIV (NR 0)))
        PLUS
        (LET "y" (NR 6) (OP (VAR "y") MUL (VAR "y")))
E2 = LET "x" (NR 42) (OP (VAR "x") PLUS (LET "x" (NR 58) (VAR "x")))
E3 = LET "x" (NR 1) (LET "y" (NR 2) (LET "x" (NR 3) (NR 4)))
E4 = LET "x" (NR 1) (OP (VAR "x") PLUS (VAR "y"))
E5 = OP (LET "x" (NR 1) (VAR "x")) MUL (VAR "x")
```

8.13.1 Expressies afdrukken

Maak een *instance* van de overloaded functie `toString` voor `Expr`:

```
instance toString Expr where
    toString ...
```

Deze instance moet `Expr` waarden weergeven zoals hierboven getoond. De uitkomst van `Start = map toString [E1,E2,E3,E4,E5]` is de lijst met de achtereenvolgende strings E_1, E_2, E_3, E_4, E_5.

Let erop dat om argumenten van een berekening die enkel bestaan uit een getal of een variabele *geen* haakjes gezet mogen worden.

8.13.2 Vrije variabelen

Schrijf de functie `free :: Expr -> [Name]` die alle vrije variabelen oplevert die in een expressie voorkomen. Een variabele x wordt gebonden door een **let** x = ... **in** e. Een variabele is vrij als hij niet gebonden is. De uitkomst van `Start = map free [E1,E2,E3,E4,E5]` is `[[], [], [], ["y"], ["x"]]`. Let er op dat in `E5` de variabele met naam `"x"` wél gedefinieerd is in het eerste argument van de vermenigvuldiging, maar niet in het tweede argument.

8.13.3 Ongebruikte variabelen

In expressie `E3` worden de variabelen met naam `"x"` en `"y"` weliswaar gedefinieerd, maar niet gebruikt. Deze expressie kan dus vereenvoudigd worden naar `(NR 4)` met behoud van betekenis. In het algemeen kun je de expressie (**let** $x = e_1$ **in** e_2) vereenvoudigen naar e_2 als x niet vrij voorkomt in e_2. Schrijf de functie `remove_unused_lets :: Expr -> Expr` die deze transformatie uitvoert. De uitkomst van `map remove_unused_lets [E1,E2,E3,E4,E5]` is `[E1,E2,NR 4,E4,E5]`.

8.13.4 Evaluator

De waarde van een expressie wordt berekend door een evaluator. De waarde van een getal is het getal zelf. De waarde van het gebruik van een variabele is zijn definitie. Dat kan dus alleen voor gedefinieerde variabelen, dus expressies E_4 en E_5 hebben geen waarde. De waarde van een rekenkundige bewerking is de rekenkundige bewerking op de waarden van zijn argumenten. We gebruiken berekeningen op integers. Delen door nul is niet toegestaan, dus expressie E_1 heeft geen waarde. De waarde van een variabele definitie is het gebruiken van zijn nieuwe definitie in de rest van de expressie. Let op: dat betekent dat de waarde van E_2 100 moet zijn, en niet 84 of 116.

Het evalueren van expressies kan dus slagen en levert dan een integer waarde op, of falen (vanwege gebruik van ongedefinieerde variabele of deling door nul) en levert dan een ongedefinieerde waarde op. Dit representeren we met:

```
::  Val = Result Int | Undef
```

Schrijf de functie `eval :: Expr -> Val` die de waarde n van een expressie uitrekent en deze oplevert als `(Result` n`)`, mits deze bestaat. Als de expressie geen geldige berekening is, dan dient `Undef` opgeleverd te worden. De uitkomst van `Start = map eval [E1,E2,E3,E4,E5]` is `[Undef,Result 100,Result 4,Undef,Undef]`.

8.14 Een λ-reducer

Main module:	*Lambda.icl*
Environment:	*StdEnv*

De λ-calculus is een sterk vereenvoudigde vorm van een functionele programmeertaal. Termen in de λ-calculus voldoen aan de volgende syntax:

$$term ::= con \mid var \mid (term\ term) \mid (\lambda\ var\ .\ term)$$

Een term is constante (*con*), een variabele (*var*), een applicatie van twee termen t_1 en t_2 $(t_1\ t_2)$ of een λ-abstractie: $\lambda x_i.t$, waarin de variabele x_i al dan niet voorkomt in t. Een variabele die geïntroduceerd wordt door een λ-abstractie noemen we ook wel *gebonden*. De voorkomens van deze variabele in t zijn *niet vrij*. Variabelen die niet door een λ-abstractie geïntroduceerd worden heten ook wel de *vrije* variabelen. Voorbeelden van λ-termen zijn:

$$
\begin{array}{lcl}
T_0 & \equiv & 42 \\
T_1 & \equiv & x_0 \\
T_2 & \equiv & (\lambda x_0.x_0) \\
T_3 & \equiv & ((\lambda x_0.x_0)\ 42) \\
T_4 & \equiv & ((\lambda x_0.(x_0 x_0))(\lambda x_1.(x_1 x_1))) \\
T_5 & \equiv & (\lambda x_0.(\lambda x_1.x_0)) \\
T_6 & \equiv & ((\lambda x_0.(\lambda x_1.x_0))\ 42)\ ((\lambda x_0.(x_0 x_0))(\lambda x_1.(x_1 x_1)))
\end{array}
$$

λ-Termen kun je in Clean representeren met behulp van een algebraïsch type:

```
:: Term    = C Value                      // constante v (C v)
           | X Index                      // variabele x_i (X i)
           | (@.) infixl 7 Term  Term     // applicatie t_1 t_2 (t_1 @. t_2)
           |  \.           Index Term     // abstractie λx_i.t (\.i t)
:: Value :== Int                          // willekeurige integer waarde
:: Index :== Int                          // index i (gebruikelijk i ≥ 0)
```

We beperken ons tot integer constanten v (`C` v). Variabele x_i geven we aan met de index i als (`X` i). De applicatie van twee termen t_1 en t_2 is (t_1 `@.` t_2). De λ-abstractie $\lambda x_i.t$ wordt weergegeven door (\.i t). De bovenstaande termen T_i worden door de volgende Clean termen t_i gerepresenteerd:

```
t0 = (C 42)
t1 = (X 0)
t2 = (\.0 (X 0))
t3 = (\.0 (X 0)) @. (C 42)
t4 = (\.0 ((X 0) @. (X 0))) @. (\.1 ((X 1) @. (X 1)))
t5 = (\.0 (\.1 (X 0)))
t6 = (\.0 (\.1 (X 0))) @. (C 42) @. t4
```

Let er bij de Clean representaties op dat je haakjes zet. Als je bijvoorbeeld `t5` definiëert als: `(\.0 \.1 (X 0))` of `(\.0 \.1 X 0)`, dan krijg je de volgende foutmelding van de Clean compiler:

Error [Lambda.icl, ..., ...]: `\.` *used with too many arguments*

8.14.1 Printen

Schrijf een *instance* van de `toString` *type class* voor het type `Term`. Deze zet dus `Term` waarden om in een `String`. Vermijd overbodige haakjes rond constanten en variabelen. Druk (`X` i) af als "`x_`i". De Clean termen t_i zou je als volgt kunnen omzetten naar `String`s (je mag hier enigszins van afwijken):

```
toString t0 = "42"
toString t1 = "x_0"
toString t2 = "(\x_0␣.␣x_0)"
toString t3 = "(\x_0␣.␣x_0)␣42"
toString t4 = "(\x_0␣.␣(x_0␣x_0))␣((\x_1␣.␣(x_1␣x_1)))"
toString t5 = "(\x_0␣.␣(\x_1␣.␣x_0))"
toString t6 = "(\x_0␣.␣(\x_1␣.␣x_0))␣42␣((\x_0␣.␣(x_0␣x_0))␣((\x_1␣.␣(x_1␣x_1))))"
```

Net als in Clean is iedere (deel)expressie van een λ-term van de vorm $((\lambda x.t_1)\ t_2)$ een *redex* (*red*uceerbare *ex*pressie). Een λ-term *zonder* redex is in *normaalvorm*.

8.14.2 Normaalvorm

Schrijf het predicaat `nf :: Term -> Bool` dat van een Clean representatie van een λ-term bepaalt of deze in normaalvorm is. Van de termen T_i zijn de termen T_0, T_1, T_2 en T_5 in normaalvorm.

Het *herschrijven* van een redex $((\lambda x.t_1)\ t_2)$ gebeurt, net als in Clean, door ieder *vrij* voorkomen van de variabele x in t_1 te vervangen door t_2. Dit heet *uniforme substitutie*. We noteren dit met $t_1 <: (x, t_2)$. Een variabele x komt vrij voor in t_1 als het niet gebonden wordt door een λ-abstractie binnen t_1. Bijvoorbeeld, in de term $((\lambda x_0.x_0)x_0)$ is alleen het laatste voorkomen van x_0 vrij, omdat de eerste gebonden wordt door de λ-abstractie. Het resultaat van $((\lambda x_0.x_0)x_0) <: (x_0, 42)$ is dus $((\lambda x_0.x_0)42)$.

8.14.3 Variabelen

Schrijf de functie `vars :: Term -> [Index]` die van de Clean representatie van een λ-term alle vrije en gebonden variabelen oplevert *zonder duplicaten*.

8.14.4 Verse variabele

Schrijf de functie `fresh :: [Term] -> Index` die, toegepast op een reeks Clean representaties van λ-termen $T_0 \ldots T_n$ de index i van een variabele x_i oplevert die *niet* voorkomt in $T_0 \ldots T_n$.

Uniforme substitutie $t_1 <: (x, t_2)$ vervangt ieder *vrij* voorkomen van x in t_1 door t_2. Deze functie is als volgt gedefinieerd:

$$
\begin{array}{llll}
con & <: (x,t) = & con & \\
x & <: (x,t) = & t & \\
y & <: (x,t) = & y & \\
(t_1\ t_2) & <: (x,t) = & ((t_1 <: (x,t))\ (t_2 <: (x,t))) & \\
(\lambda x.t_1) & <: (x,t) = & (\lambda x.t_1) & \\
(\lambda y.t_1) & <: (x,t) = & (\lambda y.(t_1 <: (x,t))) & \textbf{als}\ \ y \notin vars(t) \\
(\lambda y.t_1) & <: (x,t) = & (\lambda z.((t_1 <: (y,z)) <: (x,t))) & \textbf{met}\ z = fresh\{t, t_1\}
\end{array}
$$

8.14.5 Substitutie

Schrijf een operator `(<:) infixl 6 :: Term (Index,Term) -> Term` die de uniforme substitutie zoals hierboven beschreven implementeert.

Een redex $((\lambda x.t_1)\ t_2)$ wordt *gereduceerd* door uniforme substitutie van de vrije voorkomens van x in t_1 door t_2. Dit wordt ook wel β-reductie ($\longrightarrow_\beta$) genoemd:

$$(\lambda x.t_1)t_2 \longrightarrow_\beta t_1 <: (x, t_2)$$

8.14.6 Reductie

Schrijf de functie `beta_reduce :: Term Term -> Term` die als argumenten de Clean representaties van de λ-termen ($\lambda x.t_1$) en t_2 krijgt, en als resultaat de Clean representatie van term t oplevert waarvoor geldt: $(\lambda x.t_1)t_2 \longrightarrow_\beta t$. Als het eerste argument geen λ-abstractie is, dan dient `beta_reduce` een foutmelding te genereren.

Een λ-term t wordt gereduceerd door het herhaald toepassen van de $\longrightarrow_\beta$ regel tot de resulterende term een normaalvorm is. Op het college zijn twee reductie-strategieën behandeld: *normal order* en *applicative order*. Beide strategieën kiezen de *left-most, outermost* redex in t (de redex die zich in de `Term` representatie het meest links en hoogst in de datastructuur bevindt). *Normal order* reductie herschrijft deze redex, terwijl *applicative order* reductie eerst het argument reduceert tot deze in normaalvorm is, en dan pas de gevonden redex herschrijft.

8.14.7 Strategie

Schrijf de functies `normal_order :: Term -> Term` en `applicative_order :: Term -> Term` die van een term t de left-most, outermost redex opzoeken en herschrijven volgens de normal order en applicative order evaluatie-strategie. Het herschrijven wordt uiteraard door middel van de `beta_reduce` functie gedaan.

Voorbeeld:

```
                  t = (\.0 (\.1 (X 0))) @. ((\.0 (X 0)) @. (C 42)) @. (C 50)
normal_order      t = (\.1 ((\.0 (X 0)) @. (C 42))) @. (C 50)
applicative_order t = (\.0 (\.1 (X 0))) @. (C 42) @. (C 50)
```

8.14.8 Herschrijven tot normaalvorm

Schrijf de hogere-orde functie `herschrijf :: (Term -> Term) Term -> Term` die als argument een strategie-functie f krijgt zoals gemaakt in 8.14.7 en een term t. De term t wordt net zo lang herschreven door middel van f tot de normaalvorm bereikt is.

Voorbeeld:

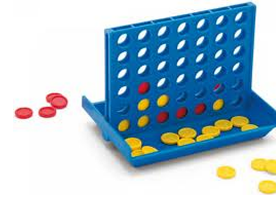
```
                       u1 = (\.0 (\.1 (X 0))) @. ((\.0 (X 0)) @. (C 42)) @. (C 50)
u2 = normal_order      u1 = (\.1 ((\.0 (X 0)) @. (C 42))) @. (C 50)
u3 = normal_order      u2 = (\.0 (X 0)) @. (C 42)
u4 = normal_order      u3 = C 42
```

Voorbeeld:

```
                       u1 = (\.0 (\.1 (X 0))) @. ((\.0 (X 0)) @. (C 42)) @. (C 50)
u2 = applicative_order u1 = (\.0 (\.1 (X 0))) @. (C 42) @. (C 50)
u3 = applicative_order u2 = (\.1 (C 42)) @. (C 50)
u4 = applicative_order u3 = C 42
```

8.15 Vier op een rij

Main module:	*VierOpEenRij.icl*
Environment:	*StdEnv*

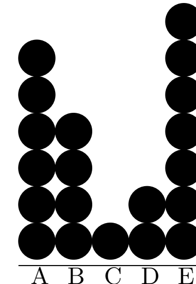


In het kinderspel *vier op een rij* spelen twee spelers tegen elkaar. Ze gebruiken een rechtopstaand speelbord dat uit een rooster bestaat van m kolommen ($m \geq 4$, vaak 7) van n rijen ($n \geq 4$, vaak $6 - 8$). In deze kolommen moeten ze om de beurt een steen laten vallen van hun eigen kleur. De speler die er als eerste in slaagt om 4 stenen van haar eigen kleur ononderbroken *naast elkaar*, *boven elkaar* of *diagonaal* te plaatsen wint.

Schrijf een programma dat *vier op een rij* implementeert. De gebruiker kan de dimensies van het speelbord kiezen en bepalen of zij de beginspeler is; het programma neemt de tegenstander voor zijn rekening. Implementeer de logica van de tegenstander met behulp van een *game tree*: ontwikkel dus een `GameState` data type en de `moves` en `worth` functies.

8.16 Nim

Main module:	*Nim.icl*
Environment:	*StdEnv*



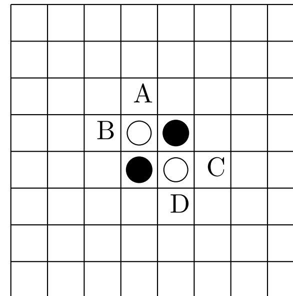
In het spel *nim* spelen twee spelers tegen elkaar. Ze beginnen met een reeks stapels die ieder een willekeurig aantal stenen heeft (bijv. 6-4-1-2-7 in de figuur hiernaast).Om de beurt neemt iedere speler van precies één stapel *tenminste* één steen (en dus maximaal het aantal stenen van die stapel). De speler die het laatst een steen weghaalt is de winnaar.

Schrijf een programma dat *nim* implementeert. De gebruiker kan kiezen of hij de beginspeler is; het programma neemt de tegenstander voor zijn rekening. Implementeer de logica van de tegenstander met behulp van een *game tree*: ontwikkel dus een `GameState` data type en de `moves` en `worth` functies.

8.17 Othello

Main module:	*Othello.icl*
Environment:	*StdEnv*

Het twee-speler spel *Othello* (ook wel bekend als *Reversi*) wordt op een bord gespeeld dat uit 8×8 vakjes bestaat. In een vakje kan een schijfje gelegd worden dat aan de ene zijde *zwart* is, en aan de andere zijde *wit*. Een speler speelt een partij met dezelfde kleur. De beginopstelling vind je hiernaast.



Bij aanvang van het spel worden de kleuren toegewezen aan de spelers. Om de beurt probeert iedere speler één steen van zijn eigen kleur op het bord te leggen. Hij moet er hierbij voor zorgen dat hij tenminste één rij (horizontaal, verticaal of diagonaal) van zijn tegenstander *insluit*: d.w.z.: aan weerszijden van de rij stenen van de tegenstander ligt nu tenminste één steen van de speler die aan zet is. In de beginopstelling kan de speler met de zwarte stenen witte stenen insluiten op de velden gemarkeerd met A, B, C en D.

Het gevolg is dat alle zojuist ingesloten stenen van de tegenstander van kleur wisselen, en nu bij de speler horen die aan zet was. Het leggen van een steen kan er voor zorgen dat meerdere rijen (horizontaal, verticaal, diagonaal) ingesloten zijn geraakt, en al deze stenen worden ook veroverd. Als een speler geen steen kan leggen waarmee hij tenminste één steen kan veroveren van de tegenstander, dan is zijn beurt over, en mag de tegenstander spelen. Als die vervolgens ook niet kan spelen, dan is het spel afgelopen. Het spel is eveneens afgelopen zodra het bord alleen maar stenen bevat van dezelfde kleur of helemaal vol is. Als het spel is afgelopen, dan is degene met de meeste stenen op het bord in zijn kleur de winnaar.

Schrijf een programma dat *Othello* implementeert. De gebruiker kan kiezen of hij de beginspeler is; het programma neemt de tegenstander voor zijn rekening. De gebruiker speelt altijd met zwarte stenen. Implementeer de logica van de tegenstander met behulp van een *game tree*: ontwikkel dus een `GameState` data type en de `moves` en `worth` functies.

8.18 Blokus

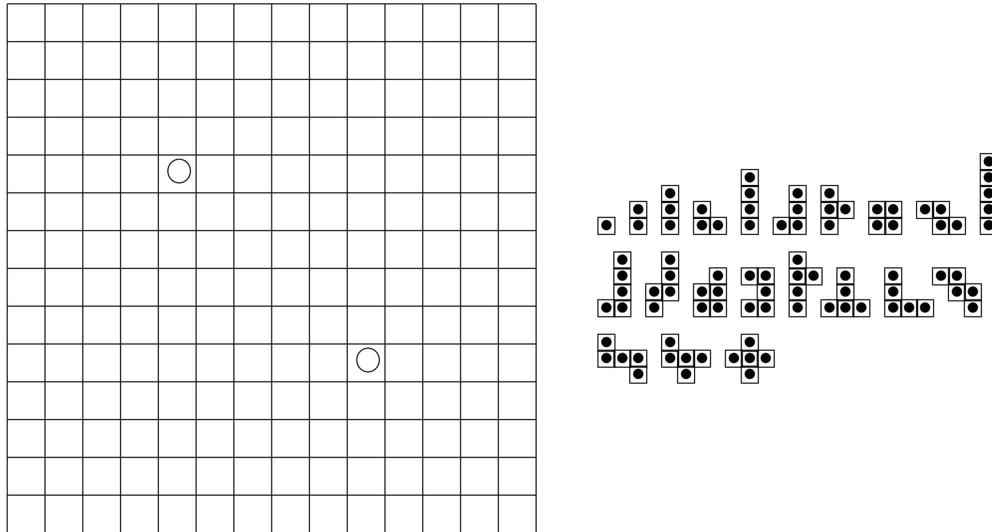
Main module:	*Blokus.icl*
Environment:	*StdEnv*

Blokus is een strategisch spel dat door twee en vier spelers gespeeld kan worden. In deze opdracht gaan we uit van twee spelers. Zie ook `www.blokus.com` voor meer informatie over blokus.

Het blokus spel bestaat uit een speelbord van 14×14 vakjes waarvan er twee gemarkeerd zijn en voor iedere speler een identieke verzameling van 21 stenen in een eigen speelkleur (zie afbeelding 8.1).

De spelers plaatsen om de beurt een van de speelstenen op het bord. De eerste steen van iedere speler moet op het gemarkeerde punt op het bord liggen. Iedere volgende steen van een bepaalde kleur moet een al geplaatste steen op het bord raken. Dit mag alleen aan de *hoekpunten* en *niet* aan de zijden. De te leggen steen mag wel de zijden raken van stenen van de andere kleur. Als een speler geen steen kan leggen, dan is zijn beurt voorbij. Het spel eindigt als geen van beide spelers nog een steen kan leggen, of als alle stenen op zijn.

De puntentelling wordt bepaald door de stenen die niet op het bord geplaatst konden



Figuur 8.1: Het blokus speelbord en de 21 blokus speelstenen

worden. Iedere eenheid telt voor -1. Als een speler al zijn stenen op het bord heeft weten te plaatsen, verdient hij $+15$ punten. Als een speler bovendien de kleinste steen als laatste gelegd heeft, verdient hij $+5$ punten. De speler met de hoogste score wint.

Schrijf een programma dat blokus implementeert. De gebruiker kan kiezen of hij de beginspeler is; het programma neemt de tegenstander voor zijn rekening. Implementeer de logica van de tegenstander met behulp van een *game tree*: ontwikkel dus een `GameState` data type en de `moves` en `worth` functies.

8.19 Abalone

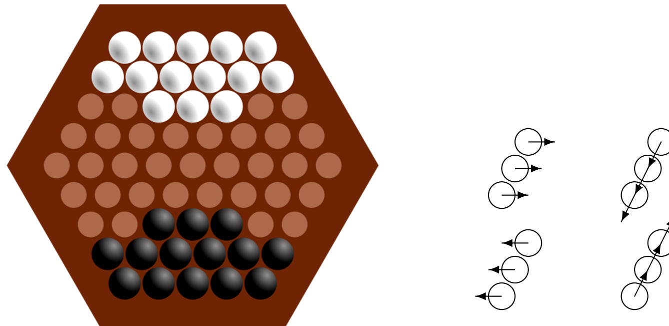
Main module:	*Abalone.icl*
Environment:	*StdEnv*

Abalone (zie ook `http://nl.wikipedia.org/wiki/Abalone`) is een strategisch bordspel voor twee spelers. Het zes-zijdige bord bestaat uit 61 gaten waarin witte en zwarte kogels geplaatst kunnen worden. Figuur 8.2**a.**[1] toont het bord en de standaard beginopstelling.

De speler met de zwarte kogels begint. De spelers doen om de beurt een zet. Een zet bestaat uit het verplaatsen van 1, 2 of 3 in één richting aaneengesloten kogels van dezelfde kleur met één stap in dezelfde richting. Dat kan dus vooruit en achteruit (in de lengte-richting) of zijwaarts. Zie figuur 8.2**b**. Kogels van de ene kleur mogen hierbij kogels van de andere kleur wegduwen, maar niet van de eigen kleur (anders verplaats je te veel eigen kogels). Hierbij geldt dat de duwende kogels in de meerderheid moeten zijn: om 1 kogel van je tegenstander weg te duwen heb je 2 of 3 eigen kogels nodig, en 2 kogels van je tegenstander kun je alleen met 3 eigen kogels wegduwen. In een zijwaartse beweging kunnen dus nooit kogels weggeduwd worden.

Het doel van het spel is om als eerste 6 kogels van de tegenstander van het bord geduwd te hebben.

[1] Bron: `http://nl.wikipedia.org/wiki/Bestand:Abalone_standard.svg`



Figuur 8.2: **a.** De beginopstelling. **b.** De bewegingsrichtingen.

Schrijf een programma dat abalone implementeert. De gebruiker kan kiezen of hij de beginspeler is; het programma neemt de tegenstander voor zijn rekening. Implementeer de logica van de tegenstander met behulp van een *game tree*: ontwikkel dus een `GameState` data type en de `moves` en `worth` functies.

8.20 `map` en type constructor classes

Main module:	*Mappen.icl*
Environment:	*StdEnv*

Bestudeer de volgende data structuren en functies:

```
map                     :: (a -> b) [a] -> [b]
map f []                = []
map f [x : xs]          = [f x : map f xs]

:: Maybe a              = Nothing | Just a

mapMaybe                :: (a -> b) (Maybe a) -> Maybe b
mapMaybe f Nothing      = Nothing
mapMaybe f (Just x)     = Just (f x)

:: Tree a               = Leaf | Node a (Tree a) (Tree a)

mapTree                 :: (a -> b) (Tree a) -> Tree b
mapTree f Leaf          = Leaf
mapTree f (Node x l r)  = Node (f x) (mapTree f l) (mapTree f r)
```

Ontwikkel in de voorgegeven module `Map.dcl` en `Map.icl` een *type constructor class* met de naam `Map` (`map` is al in gebruik) zodanig dat de functies `map`, `mapTree` en `mapMaybe` de implementaties zijn van instances van deze type constructor class `Map` van de respectievelijke type constructoren `[]`, `Tree` en `Maybe`. De main module `Mappen.icl` bevat een aanroep van ieder van deze instances.

8.21 RefactorXX, monadisch

Main module:	*RefactorXX.icl*
Environment:	*StdEnv*

In opdracht 8.13.4 heb je een evaluator geschreven voor *let-expressies* die middels de volgende recursieve datastructuren zijn gerepresenteerd:

```
:: Expr     =   NR  Int
            |   VAR Name
            |   OP  Expr Operator Expr
            |   LET Name Expr Expr
:: Name     :== String
:: Operator =   PLUS | MIN | MUL | DIV
```

8.21.1 Monadische evaluator

In deze opdracht transformeer je de door jou gemaakt evaluator naar een *monadische* variant, de `MonadFail` type constructor class om precies te zijn:

```
eval :: Expr -> c Int | MonadFail c
eval ...
```

Voor de volledigheid zijn de definities van de `MonadFail` type constructor class in de module `RefactorXX.icl` bijgevoegd:

```
class fail         c :: c a
class return       c :: a -> c a
class (>>=) infix 0 c :: (c a) (a -> c b) -> c b
class Monad        c | return, >>= c
class MonadFail    c | Monad, fail c
```

Test je monadische evaluator voor de *lijst* instances van de `MonadFail` type constructor class.

8.21.2 Monadische values

Maak van het `Val` type ook een instance van elk van de `MonadFail` type constructor classes. Maak dus de volgende definities af:

```
:: Val a = Result a | Undef

instance fail   Val where ...
instance return Val where ...
instance >>=    Val where ...
```

Test je monadische evaluator voor de `Val` instances van de `MonadFail` type constructor class.

Hoofdstuk 9

Correctheidsbewijzen

In deze opgaven geef je van een aantal stellingen het correctheidsbewijs. Deze stellingen maken gebruik van functie definities. Elk alternatief van een functie definitie is genummerd. Geef in het bewijs aan m.b.v. dit nummer welk alternatief je hebt toegepast en onderstreep dat stuk in de tekst *waarop* je deze toepassing gedaan hebt. Als je de gelijkheid van rechts naar links toegepast hebt, geef dat dan met een ⇐ aan. Sla geen stappen over.

Werk het bewijs eerst uit op papier. Op het tentamen moet je zeker een of twee bewijzen geven van stellingen, dus oefening op papier is een goede voorbereiding daarvoor.

Als je het bewijs af hebt, voer het dan in als ASCII tekst. Stel dat je moet bewijzen dat voor alle eindige lijsten `xs` geldt dat: `xs ++ [] = xs` met gegeven definitie van `++`:

```
(++) :: [a] [a] -> [a]
(++) []     ys = ys                 (1)
(++) [x:xs] ys = [x : xs ++ ys]     (2)
```

Dan **moet** je bewijs er als volgt uitzien (anders wordt het niet nagekeken):

```
Te bewijzen:
    voor alle xs::[a] : xs ++ [] = xs
Bewijs:
    met inductie naar de lengte van xs.

    Basis:
    aanname: xs = [].

        xs ++ []             // basisaanname
        **
      = [] ++ []             // (1)
        ********
      = []                   // basisaanname
        **
      = xs.

    Inductiestap:
    aanname:     stelling geldt voor zekere xs, ofwel:
                 xs ++ [] = xs  (IH)
```

```
Te bewijzen: stelling geldt ook voor [x:xs], ofwel:
             [x:xs] ++ [] = [x:xs]
Bewijs:
    [x:xs] ++ []         // (2)
    ************
  = [x : xs ++ []]       // (IH)
         ********
  = [x : xs].

Dus: basis + inductiestap => stelling bewezen.
```

9.1 map en o

Main module:	*BewijsMapO.icl*
Environment:	—

Maak gebruik van de volgende functie definities:

```
map :: (a -> b) [a] -> [b]
map f []     = []                          (1)
map f [x:xs] = [f x : map f xs]            (2)

(f o g) x    = f (g x)                     (3)
```

Bewijs de volgende stelling voor alle *eindige* lijsten xs en functies f en g:

```
map (f o g) xs = map f (map g xs)
```

9.2 init en take

Main module:	*BewijsInitTake.icl*
Environment:	—

Maak gebruik van de volgende functie definities:

```
init :: [a] -> [a]
init [x]      = []                         (1)
init [x : xs] = [x:init xs]                (2)

take :: Int [a] -> [a]
take 0 xs     = []                         (3)
take n []     = []                         (4)
take n [x:xs] = [x : take (n-1) xs]        (5)

length :: [a] -> Int
length []     = 0                          (6)
length [x:xs] = 1 + length xs              (7)

(f o g) x     = f (g x)                    (8)
```

Bewijs de volgende stelling voor alle *eindige, niet-lege* lijsten xs:

```
init xs = take (length xs - 1) xs
```

Ga er voor het gemak van uit dat het *Int*eger bereik onbeperkt is.

9.3 Peano aritmiek

Main module:	*BewijsPeano.icl*
Environment:	—

We kunnen natuurlijke getallen als volgt representeren (we noemen dat ook wel *Peano* aritmetiek):

```
:: Nat = Zero | Suc Nat
```

Oftewel: 0 is een natuurlijk getal (`Zero`), en als x een natuurlijk getal is, dan is de opvolger van x (`Suc` x) ook een natuurlijk getal. De functie die de relatie tussen *Nat* en *Int* aangeeft is de volgende:

```
( ## ) :: Nat -> Int
( ## ) Zero    = 0             (1)
( ## ) (Suc n) = 1 + ##n       (2)
```

Ga er voor het gemak van uit dat het *Int*eger bereik onbeperkt is.

Optellen Zij gegeven de functie `add`:

```
add :: Nat Nat -> Nat
add Zero    n = n                (3)
add (Suc m) n = Suc (add m n)    (4)
```

Bewijs de volgende stelling voor alle m en n:

```
##(add m n) = ##m + ##n
```

Vermenigvuldigen Zij gegeven de functie `mul`:

```
mul :: Nat Nat -> Nat
mul m Zero    = Zero               (5)
mul m (Suc n) = add (mul m n) m    (6)
```

Bewijs de volgende stelling voor alle m en n:

```
##(mul m n) = ##m * ##n
```

9.4 `map`, `flatten` en `++`

Main module:	*BewijsMapFlatten.icl*
Environment:	—

Beschouw de volgende twee functie definities:

```
(++) :: [a] [a] -> [a]
(++) [ ]    xs = xs                          (1)
(++) [y:ys] xs = [y : ys ++ xs]              (2)

map :: (a -> b) [a] -> [b]
map f [ ]           = [ ]                    (3)
map f [x:xs]        = [f x : map f xs]       (4)
```

9.4.1 `map` en `++`$^{(gebruikt\ in\ 9.5)}$

Bewijs de volgende stelling voor alle *eindige* lijsten *as* en *bs* en functie *f* (denk goed na over welke lijst je de inductie laat lopen!).

```
map f (as ++ bs) = (map f as) ++ (map f bs)      (9.4.1)
```

9.4.2 `map` en `flatten`

Beschouw de volgende functie definitie:

```
flatten :: [[a]] -> [a]
flatten [ ]    = [ ]                          (5)
flatten [x:xs] = x ++ (flatten xs)            (6)
```

Bewijs de volgende stelling met behulp van volledige inductie over de lengte van lijst *xs*. Ga er van uit dat *xs* een eindige lijst is:

```
flatten (map (map f) xs) = map f (flatten xs)
```

Maak hierbij gebruik van eigenschap 9.4.1. **Let op:** als je 9.4.1 niet hebt bewezen mag je in dit onderdeel er van uit gaan dat de eigenschap geldig is.

9.5 Lijsten en bomen

Main module:	*BewijsMeppenEnTippen.icl*
Environment:	—

In deze opdracht maken we gebruik van de volgende type definities en functies:

```
::  BTree a = Tip a | Bin (BTree a) (BTree a)

map :: (a -> b) [a] -> [b]
map f []     = []                                          (1.)
map f [x:xs] = [f x : map f xs]                            (2.)

mapbtree :: (a -> b) (BTree a) -> BTree b
mapbtree f (Tip a)     = Tip (f a)                         (3.)
mapbtree f (Bin t1 t2) = Bin (mapbtree f t1) (mapbtree f t2)  (4.)

foldbtree :: (a a -> a) (BTree a) -> a
```

```
foldbtree f (Tip x)     = x                                     (5.)
foldbtree f (Bin t1 t2) = f (foldbtree f t1) (foldbtree f t2)   (6.)

tips :: (BTree a) -> [a]
tips t = foldbtree (++) (mapbtree unit t)                       (7.)

unit :: a -> [a]
unit x = [x]                                                    (8.)
```

Leg uit wat de functies `foldbtree` en `tips` doen. Bewijs de volgende stelling voor elke functie f, en elke eindige binaire boom t:

```
map f (tips t) = tips (mapbtree f t)
```

Maak gebruik van hulpstelling 9.4.1.

9.6 subs en map

Main module:	*BewijsSubsEnMap.icl*
Environment:	—

Zij gegeven de volgende functie-definities:

```
subs            :: [a] -> [[a]]
subs []         = [[]]                                          (1.)
subs [x:xs]     = subs xs ++ map (cons x) (subs xs)             (2.)

map             :: (a -> b) [a] -> [b]
map f []        = []                                            (3.)
map f [x:xs]    = [f x : map f xs]                              (4.)

(++)            :: [a] [a] -> [a]
(++) []     ys  = ys                                            (5.)
(++) [x:xs] ys  = [x : xs ++ ys]                                (6.)

cons            :: a [a] -> [a]
cons x xs       = [x : xs]                                      (7.)
```

Leg uit wat de functie `subs` uitrekent en geef de uitkomst van de expressie (`subs ['abcd']`).
Bewijs de volgende stelling voor alle functies f en eindige lijsten xs:
`subs` (`map` f xs) $=$ `map` (`map` f) (`subs` xs).
Je kunt gebruik maken van de volgende hulpstellingen (lemma's) die gelden voor alle functies f, g en eindige lijsten xs en ys:

`map` f (xs `++` ys)	$=$	`map` f xs `++` `map` f ys	**(8.)**
`map` g (`map` f xs)	$=$	`map` (g `o` f) xs	**(9.)**
(`cons` (f a)) `o` (`map` f)	$=$	(`map` f) `o` (`cons` a)	**(10.)**
(g `o` f) x	$=$	g (f x)	**(11.)**

Hoofdstuk 10

Dynamics

Dynamics stellen je in staat een berekening in te pakken en door te geven. Bij het inpakken wordt niet alleen de berekening op de een of andere manier opgeslagen, maar ook een representatie van diens *type*. Een ingepakte berekening kan op de gebruikelijke manier doorgegeven worden naar andere functies, maar ze kan ook weggeschreven worden naar een file om op een later moment door dezelfde of een andere applicatie weer ingelezen te worden. De applicatie die een dergelijke dynamic uitpakt moet echter wel aangeven welk type verwacht wordt. Alleen als dat type kloppend gemaakt kan worden met het daadwerkelijke type kan de applicatie de concrete inhoud gebruiken. Deze controles vinden plaats gedurende executie van de applicatie.

10.1 Notaties

Main module:	*NotatieDynamics.icl*
Environment:	*Experimental*

Hieronder staan een aantal functies en expressies. Leid van iedere functie het meest algemene type af. Leg van elke functie en expressie uit wat deze betekent.

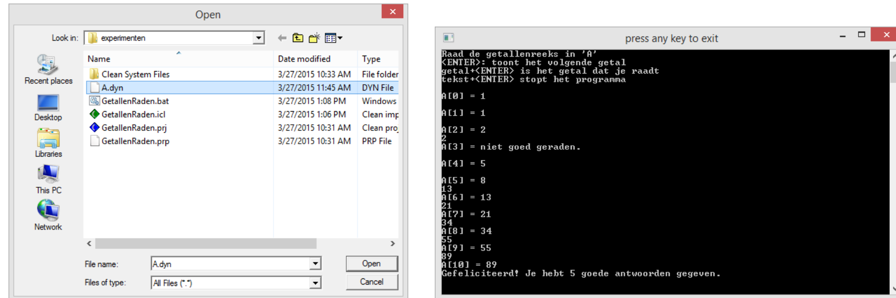
```
f1 (x :: Int) y                    = x + y
f2 (b :: Bool) (e1 :: a) (e2 :: a) = dynamic if b e1 e2 :: a
f3                                 = dynamic map fib [1 ..]
f4 (xs :: [Int])                   = take 10 xs
f5                                 = f4 f3
```

10.2 Getallen raden

Main module:	*GetallenRaden.icl*
Environment:	*Experimental*

In deze opdracht ontwikkel je een console-I/O programma dat een dynamic file inleest (met een non-descripte filenaam zoals A, B, C, etc., te selecteren door de gebruiker). Stel dat de gebruiker de file met de naam *filenaam* heeft gekozen (in figuur 10.1, linker afbeelding, zie je dat de gebruiker de dynamic file `"A.dyn"` selecteert). Het doel is dat de gebruiker in zo min mogelijk stappen raadt welke getallenreeks in de dynamic is opgeslagen. Figuur 10.1, rechter afbeelding, laat een sessie zien. Als de gebruiker *enter*

invoert, dan reageert het programma door het eerstvolgende getal in de reeks te laten zien. Als de gebruiker een getal invoert en dit komt niet overeen met het volgende getal, dan krijgt zij niet te zien welk getal het had moeten zijn (in de getoonde sessie bij het vierde getal). Als het getal klopt, dan wordt dit bevestigd door het te tonen. Als de gebruiker een bepaald aantal keer het juiste getal heeft voorspeld (in de getoonde sessie vijf keer), dan heeft ze de reeks ontdekt.



Figuur 10.1: Het getal-raad programma in actie.

10.3 Heterogene verzamelingen

Main module:	*StdDynSet.icl*
Environment:	*Experimental*

In opdracht 6.3 wordt er gevraagd om een module te ontwikkelen voor verzamelingen van elementen van *hetzelfde* type (*homogene verzamelingen*). In deze opdracht gaan we een stap verder, en gebruiken we *dynamics* om verzamelingen van elementen van (mogelijk) *verschillend* type te maken (*heterogene verzamelingen*). De elementen die in deze verzamelingen kunnen zitten moeten vergeleken kunnen worden, en dus de overloaded `==` ondersteunen. Om verzamelingen af te kunnen drukken is ook de overloaded `toString` instantie vereist (niet noodzakelijk, wel handig om te testen). Dit korten we af middels:

```
class Set a | TC, ==, toString a
```

Realiseer de implementatie module die hoort bij `StdDynSet.dcl`. De volgende eigenschappen dienen te gelden voor de operaties op heterogene verzamelingen:

- De `zero` instance levert de *lege verzameling* op.
- De `toString` instance toont alle elementen van de verzameling tussen `"{"` en `"}"`, en gescheiden door `,`.
- De `==` instance levert alleen `True` als de twee verzamelingen exact dezelfde elementen hebben.
- `nrOfElts` levert het aantal elementen van de verzameling op; `isEmptySet` levert alleen `True` op voor de *lege verzameling* (de lege verzameling heeft nul elementen).

- `memberOfSet` bepaalt of het eerste argument een element is van de verzameling; `isSubset` bepaalt of het eerste argument een deelverzameling is van het tweede argument (ieder element van de eerste verzameling komt voor in de tweede verzameling); `isStrictSubset` bepaalt of het eerste argument een *strikte* deelverzameling is van het tweede argument (ieder element van de eerste verzameling komt voor in de tweede verzameling, maar de tweede verzameling heeft daarnaast elementen die niet in de eerste verzameling voorkomen).
- `union` berekent de vereniging van de twee verzamelingen; `intersection` berekent de doorsnede van de twee verzamelingen; `without` verwijdert uit de eerste verzameling alle elementen uit de tweede verzameling.

`Set` is een instance van `==` en `toString`, en dus ook van de class `Set`. Met deze module kun je dus ook verzamelingen van verzamelingen maken.

10.4 Een IKS interpreter

Main module:	*IKS.icl*
Environment:	*Experimental*

In deze opdracht ga je met behulp van dynamics een *interpreter* maken voor een extreem eenvoudig programmeertaaltje: X. X is een zogenaamde *combinator* taal. Een X programma is een expressie, opgebouwd uit:

1. Het symbool `I`, `K` of `S`.
2. De gehele getallen.
3. Als e een X expressie is, dan is `(`e`)` ook een X expressie.
4. Als e_1 en e_2 X expressies zijn, dan is e_1 e_2 dat ook.

De volgende expressies zijn dus allemaal geldige X expressies: `I`, `K`, `I(42)`, `K(42)(100)`, `(I)`, `(K)`, `(S)`, `SI`, `(SI)`, `SKK(100)`, enz. Om deze opdracht niet nodeloos ingewikkeld te maken mag je aannemen dat expressies geen *white-space* tekens bevatten. Dit wordt bij bovenstaande voorbeelden aangegeven door ieder afzonderlijk getal argument van een X expressie tussen haakjes te plaatsen. De betekenis van X expressies wordt gegeven door de volgende interpretatie $[\![\cdot]\!]$:

- $[\![I]\!] \stackrel{def}{=} \lambda x.x$; $[\![K]\!] \stackrel{def}{=} \lambda xy.x$; $[\![S]\!] \stackrel{def}{=} \lambda xyz.xz(yz)$.
- $[\![n]\!] \stackrel{def}{=} n$, als n een geheel getal is.
- $[\![(e)]\!] \stackrel{def}{=} [\![e]\!]$.
- $[\![e_1\ e_2]\!] \stackrel{def}{=} ([\![e_1]\!]\ [\![e_2]\!])$.

Hoewel dit taaltje er extreem eenvoudig uitziet, is het even krachtig als de zuivere λ-calculus.

Maak nu m.b.v. dynamics een interpreter voor X expressies. Om met dynamics te kunnen werken, moet je nadat je een Clean project aangemaakt hebt de *environment* op *Experimental* zetten. Tenslotte moet je in *Project Options* het vlaggetje *Enable dynamics* aanzetten.

Dynamics

Schrijf een programma dat drie file-dynamics oplevert in dezelfde directory als de applicatie:

1. Een bestand met de naam “I” en inhoud **dynamic** $[\![I]\!]$.
2. Een bestand met de naam “K” en inhoud **dynamic** $[\![K]\!]$.
3. Een bestand met de naam “S” en inhoud **dynamic** $[\![S]\!]$.

Gebruik hiervoor de functie `writeDynamic`. Let er op dat je het correcte type geeft bij elk van de dynamics, inclusief ∀ quantificatie. Bijvoorbeeld:

```
Start :: *World -> *World
Start world
    # (ok,world) = writeDynamic "I" (dynamic i :: A. a: a -> a) world
    ...
    = world

i :: a -> a
i x = x
...
```

Na executie zijn de bestanden “I.dyn”, “K.dyn” en “S.dyn” in de directory toegevoegd.

Parseren

De volgende stap is het *parseren* van een regel invoer. Introduceer hiervoor het volgende algebraïsche data type `IKS`:

```
:: IKS = I | K | S | N Int | App IKS IKS
```

Dit type representeert de *syntax boom* van de ingevoerde expressie. De symbolen `'I'`, `'K'` en `'S'` komen overeen met de alternatieven `I`, `K` en `S`. De gehele getallen corresponderen met het alternatief `N`. Let er bij het parseren op dat X expressies, net als λ expressies, links-associatief zijn. Dat betekent dat bijvoorbeeld de expressies `SKI`, `(SK)I` en `((S)K)I` dezelfde betekenis hebben.

Enkele tips voor dit onderdeel:

- Zet elke te parseren `String` om naar een `[Char]`. Gebruik hiervoor de overloaded functie `fromString`. Vergeet niet het afsluitende *newline* character te elimineren!
- Schrijf de functie `pIKS :: [Char] -> IKS` die één regel invoer parseert. Als je rekening wilt houden met verkeerde invoer, dan kun je deze ook als `pIKS :: [Char] -> Maybe IKS` definiëren. Dit laatste is optioneel: als je wilt mag je er van uit gaan dat alleen geldige invoer gegeven wordt.
- Vanwege het links-associatieve karakter van X expressies is het het voordeligst om invoer van achter naar voor te parseren, in plaats van van voor naar achter. Ontleed de invoer dus niet m.b.v. `hd` en `tl` (of `[x:xs]` pattern-matches), maar gebruik de functies `init` en `last`.
- Omdat X expressies haakjes bevatten, is het handig om een hulpfunctie te definiëren die een groepje gebalanceerde haakjes uit een expressie haalt:

```
split_bracket :: [Char] -> ([Char],[Char])
```

Bijvoorbeeld:

`split_bracket ['(SK)I']` ⇒ `(['(SK)'],['I'])`.
`split_bracket ['((S)K)I']` ⇒ `(['((S)K)'],['I'])`.

- Een lijst van digits kun je eenvoudig omzetten naar een integer:

```
getal :: [Char] -> Int
getal chars = toInt (toString chars)
```

Interpreteren

Een `IKS` syntax-boom kan geïnterpreteerd worden volgens de $[\![\cdot]\!]$ functie die in de inleiding van deze opdracht beschreven is. Schrijf een interpreter functie `interp` die als argumenten de dynamics meekrijgt die corresponderen met de combinatoren `I`, `K` en `S` en de `IKS` syntax boom van de geparseerde expressie. Het resultaat van `interp` is een dynamic die correspondeert met de geïnterpreteerde functie. Het type is dus:

```
interp :: (Dynamic,Dynamic,Dynamic) IKS -> Dynamic
```

Gebruik voor *applicatie* van symbolen ($[\![e_1\ e_2]\!] \stackrel{def}{=} ([\![e_1]\!]\ [\![e_2]\!])$) de functie `dynApply`:

```
dynApply :: Dynamic Dynamic -> Dynamic
dynApply (f :: a -> b) (x :: a) = dynamic f x :: b
dynApply     _              _    = dynamic "dynamic type error"
```

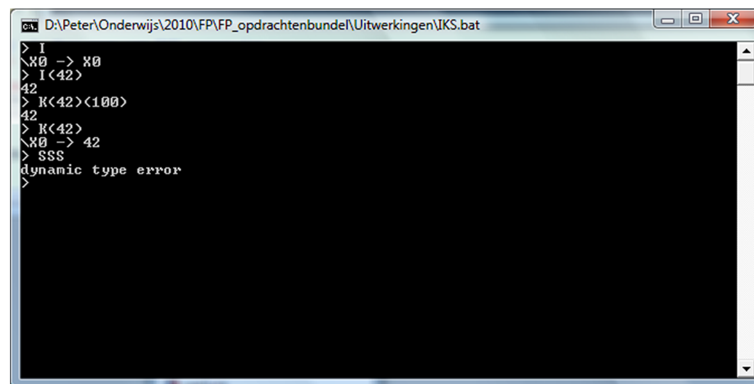
Console

In dit onderdeel brei je de bovenstaande onderdelen aan elkaar in een console I/O programma. In de console kan de gebruiker `X` expressies invoeren. Het programma "prompt" met de interpretatie van de ingevoerde `X` expressie. Hiertoe leest het programma eerst de drie dynamics in die in onderdeel 10.4 op disk geschreven zijn. Gebruik hiervoor de functie `readDynamic`. Bijv:

```
    # (ok,dyn_I,world) = readDynamic "I" world
```

zou succesvol de dynamic in moeten lezen die in onderdeel 10.4 weggeschreven is.

De prompt moet onderscheid maken tussen een functie-resultaat, een integer-resultaat, en een string-resultaat. Een voorbeeld van een sessie wordt gegeven in figuur 10.2.



Figuur 10.2: De IKS interpreter in actie.

Hoofdstuk 11

SoccerFun

Voor de opdrachten in dit hoofdstuk heb je de SoccerFun omgeving nodig.

11.1 Training: rondjes lopen

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implementeer een voetballer die rondjes rond het veld loopt met de wijzers van de klok mee. Hij mag niet verder van de rand van het voetbalveld komen dan 5 meter. Als hij in het begin niet bij de rand van het voetbalveld staat, dan moet hij eerst in de richting van de loodlijn van de meest dichtbijzijnde rand van het voetbalveld lopen (dus als hij 'meest' noord staat, dan moet hij recht naar het noorden lopen, als hij 'meest' west staat, dan moet hij recht naar west lopen, enz.).

11.2 Training: slalommen

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implementeer een voetballer die om een aantal tegenstanders slalomt. Als de voetballer op de `West` helft begint, dan rent hij naar `East`. Als de voetballer op de `East` helft begint, dan rent hij naar `West`. Aan de andere zijde van het voetbalveld ligt de bal te wachten om in de dichtsbijzijnde goal te worden geschopt. Gebruik de *RefereeCoach_Slalom* en *Opp_Slalom* om je *Student Slalom* te testen. Deze laatste wordt geïmplementeerd in module `Team_Student_Slalom_Assignment`.

11.3 Training: overspelen

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implementeer een voetbalbrein voor spelers die de bal naar elkaar overspelen van `West` naar `East` en de bal in de goal schoppen (of omgekeerd vanuit `East` naar `West`). Gebruik de *RefereeCoach Passing* en *Opp_Passing* om je *Student Passing* te testen. Deze laatste wordt geïmplementeerd in module `Team_Student_Passing_Assignment`.

11.4 Training: vrij overspelen

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implementeer voetbalbreinen voor twee spelers die aan weerszijden van een groep tegenstanders staan. De eerste speler moet de bal over de grond schoppen naar de andere speler zonder dat de tegenstanders de bal in bezit kunnen nemen. Een kwestie van timing dus. Gebruik de *RefereeCoach DeepPass* en *Opp_Deep_Pass* om je *Student_Deep_Pass* te testen. Deze laatste wordt geïmplementeerd in module `Team_Student_DeepPass_Assignment`.

11.5 Training: keeper

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implementeer een brein voor een keeper die zijn goal verdedigt. De keeper is omringd door tegenstanders die de bal naar elkaar overspelen. De keeper moet er voor zorgen dat deze de bal niet naar het midden van de goal kunnen spelen zonder dat de keeper in de weg staat. De keeper moet vóór de doellijn staan. Gebruik de *RefereeCoach Keeper* en *Opp_Keeper* om je *Student Keeper* te testen. Deze laatste wordt geïmplementeerd in module `Team_Student_Keeper_Assignment`.

11.6 Eindopdracht

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

In deze eindopdracht ontwikkel je een compleet voetbalteam dat bestaat uit één doelman en tien veldspeler. Maak een Clean module met je eigen naam (zoals PeterAchten.icl) en implementeer en exporteer hierin de team functie:

```
implementation module PeterAchten

import Footballer

TeamPeterAchten :: !Home !FootballField -> Team
TeamPeterAchten home field = ...
```

De bijbehorende *definitie module* dient er dus als volgt uit te zien:

```
definition module PeterAchten

import Footballer

TeamPeterAchten :: !Home !FootballField -> Team
```

zoals dat al in het raamwerk gedaan is voor `TeamMiniEffie`. Je mag zelf de begin-opstelling bepalen. Het argument van type `Home` geeft aan welke speelhelft je toegewezen krijgt (`West` of `East`). Je maakt je team beschikbaar in het raamwerk door in de module `Team.icl` de volgende regels toe te voegen (hier met module `PeterAchten.icl`):

```
implementation module Team
...
import PeterAchten                          // vergeet niet je module te importeren
...
allAvailableTeams = [ Team_MiniEffies
                    , TeamPeterAchten       // hiermee maak je je team bekend
                    ]
```

Dat is alles. Nu is je team selecteerbaar in het raamwerk.

Je mag naar eigen inzicht kiezen of je voor iedere voetballer een aparte breinfunctie schrijft, of dat je categorieën voetballers creëert zoals keeper, verdedigers en aanvallers, of gaat voor een universeel brein. De voetballers *moeten* aan de volgende eisen voldoen:

Doelman: Deze mag het *strafschopgebied* niet verlaten. Als de bal redelijkerwijs binnen bereik is, moet de keeper deze onderscheppen. “Redelijkerwijs binnen bereik” houdt in dat de doelman rekening houdt met zijn afstand tot de bal en diens snelheid, en de afstand en snelheid van de overige spelers tot de bal. Als hij in staat is om eerder bij de bal te zijn dan een tegenstander, dan is hij verplicht de bal te spelen. Dit is afhankelijk van de strategie van zijn medespelers: de doelman is niet verplicht medespelers in de weg te lopen.

Veldspeler: De veldspelers mogen niet met zijn allen achter de bal aanlopen. In plaats daarvan moeten ze een redelijke veldverdeling aanhouden. “Redelijke veldverdeling” houdt in dat veldspelers er naar streven om een bepaald deel van het voetbalveld te bespelen, en dat de gezamenlijke overlap tussen deze delen nihil tot erg klein is. Afhankelijk van de spelsituatie (bijvoorbeeld aanvallen en verdedigen) dienen ze deze posities in te nemen. Veldspelers mogen niet continu in balbezit zijn; d.w.z. ze zijn verplicht de bal af te spelen naar andere spelers als dit redelijkerwijs zinvol is. ”Redelijkerwijs zinvol” houdt in dat als een medespeler er voldoende beter voor staat dan de voetballer zelf, en aanspeelbaar is, dat dan de bal afgespeeld wordt.

Spelregels: Voetballers dienen de beslissingen van de scheidsrechter te respecteren. Dat wil zeggen dat je team *niet* de bal mag spelen of in bezit nemen als de tegenpartij daar recht op heeft (bijv. bij een inworp, doeltrap, corner, enz.). Dat wil ook zeggen dat je team *verplicht is* de bal te spelen of in bezit te nemen als de scheidsrechter dat aangeeft (soortgelijke situaties).

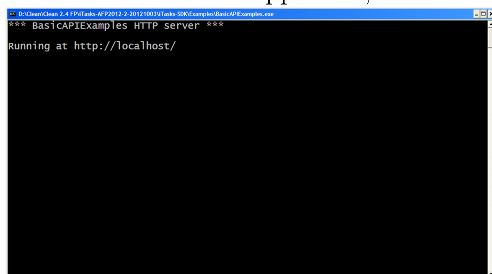
Efficiency: Voor alle voetballers geldt dat de breinfunctie voldoende efficient is, d.w.z.: als het team dat je gemaakt hebt in tweevoud opgesteld zou worden, dan mag de berekening van de voetbalacties van alle 22 spelers gezamenlijk niet meer tijd kosten dan één-twintigste seconde. Dit kun je controleren aan de hand van de *framerate indicator*: dat is het getal dat achter de tekst `Rounds/sec:` staat. Dit getal mag bij normale snelheid niet onder de 20 komen (behalve als er een scheidsrechterdialoog tussenbeide komt).

Hoofdstuk 12

TOP

Voor de opdrachten in dit hoofdstuk heb je de iTask distributie nodig. Je kunt deze vinden op: `http://wiki.clean.cs.ru.nl/ITasks`. Vanwege het experimentele karakter heb je een aangepaste versie van de Clean compiler en programmeeromgeving nodig. Je vindt een compleet systeem bij *'Snapshot for AFP2012 Course'*. Download deze distributie en pak hem uit. Je vindt daarin een *CleanIDE*. Voer de volgende taken uit:

- Start de bijbehorende *CleanIDE*;
- Open het project: {*Application*}*/iTasks-SDK/Examples/BasicAPIExamples.prj*;
- Build het project en launch de gegenereerde applicatie (*Update and Run (Ctrl+R)*): dit is een web-server applicatie;



- Start je favoriete web browser (zie de bovenstaande URL om bekende problemen met web browsers op te lossen) en navigeer naar `http://localhost/`.

Als alles goed gaat, krijg je in je browser het volgende login-scherm te zien:

Login
iTasks Example Collection
Enter your credentials and login or press continue to remain anonymous
Username*:
Password*:
Login Continue

Je kunt altijd inloggen met de waarden *root* en *root*, of anoniem via *Continue*.

Alle opdrachten in dit hoofdstuk breiden het project `BasicAPIExamples` uit. Ga voor iedere *opdracht* in dit hoofdstuk als volgt te werk:

1. Breid de folder {*Application*}*/iTasks-SDK/Examples/* uit met:

```
definition module opdracht

import iTasks

opdracht :: Workflow
```

```
implementation module opdracht

import iTasks

opdracht :: Workflow
opdracht = workflow "opdracht" "omschrijving" wf

wf = viewInformation "" [] "Hello␣World!"
```

2. Breid de main module `BasicAPIExamples.icl` uit met:

```
module BasicAPIExamples

bestaande imports laten staan...

import opdracht

basicAPIExamples :: [Workflow]
basicAPIExamples =
    [ opdracht
    , bestaande voorbeelden laten staan...
    ]
```

3. "Project: Update and Run (Ctrl+R)" van de applicatie voegt jouw *opdracht* bovenaan de lijst van uitvoerbare taken toe.

12.1 Galgje

Main module:	*BasicAPIExamples.icl*
Environment:	*iTasks*

Maak de module `Galgje.icl` en `Galgje.dcl` zoals uitgelegd in het begin van dit hoofdstuk.

In deze opdracht implementeer je een task die het spel *galgje* met jou speelt waarin jij de woorden moet raden die gekozen en gecontroleerd worden door de task (zie opdracht 7.12 voor verdere uitleg van het spel).

Kopiëer een woordenlijst naar {*Application*}*/iTasks-SDK/Examples/*. Een geschikte kandidaat is het *"Nederlands_lexicon"* dat je in de Clean distributie kunt vinden in: {*Application*}*/Examples/ObjectIO Examples/scrabble/Nederlands*.

In deze opdracht kun je *niet* de pseudo-randomgenerator gebruiken uit opdracht 5.18 omdat de iTask distributie gebruik maakt van een andere `Random` module. Dat wordt je eerste klus.

Taak: maak random getal

Ontwikkel de taak:

```
import Random           // voor het genereren van random getallen

randomgetal :: Task Int
randomgetal = ...
```

De `Random` module biedt de functie `genRandInt` aan die een oneindige lijst van random getallen genereert, gegeven een initiële waarde. Om aan een willekeurige initiële waarde te komen zou je de huidige tijd kunnen uitlezen. Dit doe je met `(get currentTime) :: Task Time`. `Time` is een voorgedefinieerd type in iTask (en staat in module `SystemTypes`).

Test je task door in `Galgje.icl` de functie `wf` te vervangen in:

```
wf = randomgetal >>= \getal -> viewInformation "" [] getal
```

Als het goed is, zal iedere keer dat deze taak gestart wordt een ander getal gegenereerd en getoond worden.

Taak: lees bestand

Ontwikkel de taak:

```
import File         // om file regels in te lezen
import StdFile      // om een World instance voor FileSystem te hebben

leesbestand :: String -> Task [String]
leesbestand filenaam = accWorldError (readFileLines filenaam) id
```

De iTask module `File` biedt een functie aan die alle regels uit een bestand leest: `readFileLines`. Deze overloaded functie verwacht een `FileSystem` environment argument die aangeboden kan worden door deze functie als volgt aan te roepen: `(accWorldError (readFileLines filenaam) id))`.

Test je task als volgt (als je *"Nederlands_lexicon"* gebruikt):

```
wf = leesbestand "Nederlands_lexicon" >>= \woorden ->
     case woorden of
        [w:ws] -> viewInformation "" [] ("Eerste␣woord␣is␣" +++ w)
        leeg   -> viewInformation "" [] "Bestand␣niet␣ingelezen."
```

Taak: kies random woord

Ontwikkel de taak:

```
import Text     // om laatste newline teken weg te halen (rtrim)

randomwoord :: Task String
randomwoord = ...
```

`randomwoord` leest de woordenlijst in met de taak `leesbestand` en kiest daaruit een willekeurig woord, gebruik makend van `randomgetal`.

Houd er rekening mee dat de random module positieve en negatieve getallen genereert, en dat de ingelezen woorden eindigen met een newline-teken. Het newline-teken kan verwijderd worden met de `rtrim` functie uit de iTask module `Text`.

Test je task bijvoorbeeld als volgt:

```
wf = randomwoord >>= \w -> viewInformation "Willekeurig woord" [] w
```

Als het goed is, wordt iedere keer een ander woord getoond.

Taak: gebruiker aan het woord

Het programma zal aan de gebruiker om invoer moeten vragen. Deze invoer is ofwel een letter ofwel een poging om het woord te raden. Een manier om invoer van de gebruiker te krijgen is met behulp van de `enterInformation` taak-functie. Ontwikkel hiermee twee taken:

```
letter :: [Char] -> Task Char
letter al_gebruikt = enterInformation ...

gokje  :: Task String
gokje  = enterInformation ...
```

`(letter al_gebruikt)` vraagt één letter aan de gebruiker, en laat zien wat deze al gebruikt heeft. De taak `gokje` vraagt de gebruiker om een woord.

Je kunt deze taken tegelijkertijd aan de gebruiker aanbieden met behulp van de `-||-` taak-combinator. Deze gaat er van uit dat de twee deeltaken een resultaat van hetzelfde type hebben. Daarom moeten de resultaten van `letter` en `gokje` naar eenzelfde type gebracht worden. Een manier om dit te doen is als volgt:

```
:: Poging    = Gok Woord | Letter Char
derive class iTask Poging      // zorgt ervoor dat iTask met Poging-waarden kan werken

gebruiker :: [Char] -> Task Poging
gebruiker al_gebruikt
    = (letter al_gebruikt >>= \c -> return (Letter c))
          -||-     (gokje >>= \w -> return (Gok w))
```

Test je task bijvoorbeeld als volgt:

```
wf = gebruiker ['abc'] >>= \poging -> viewInformation "Poging:" [] poging
```

Met deze taak zou de gebruiker niet in staat moeten zijn om de enkele tekens `'a'`, `'b'` of `'c'` in te voeren. Dat kan uiteraard wel via `gokje`.

Taak: raad het woord

Ontwikkel de *recursieve* taak `raden`:

```
:: Toestand = ...
derive class iTask Toestand  // zorgt ervoor dat iTask met Toestand-waarden kan werken

raden :: Toestand -> Task Toestand
raden toestand = ...
```

Deze taak laat de gebruiker net zo vaak aan het woord (met behulp van de `gebruiker` taak) tot er geen beurten meer zijn of het woord geraden is. Ontwikkel een geschikte `Toestand` (gebruik een record vanwege uitbreidbaarheid) om de informatie bij te houden.

Voor ieder nieuw type dat je maakt moet je instanties genereren voor de generieke klasse `iTask`. Doe dat zoals dat in de voorbeelden hierboven gedaan is voor `Poging` en `Toestand`.

Test je task bijvoorbeeld als volgt:

```
wf = raden { ... "hottentottententententoonstelling" ... }
```

Taak: breng verslag uit

Ontwikkel de taak:

```
verslag :: Toestand -> Task Void
verslag eindtoestand = ...
```

die de gebruiker op de hoogte brengt van de geleverde prestatie.

Taak: galgje

Je hebt nu alle deeltaken gemaakt die je nodig hebt om de hoofdtaak te maken. Ontwikkel en test de hoofdtaak.