A Functional Programmer's Workout

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

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

Version October 16 2018

Preface

This booklet is a collection of exercises that cover the material taught in the *functional programming* courses at the Radboud University. You can use them to test, train, and improve your skills in functional programming and prepare yourself for the examination.

We use the Clean **2.4** distribution. You can download it for free from the site: `http://wiki.clean.cs.ru.nl/`

The exercises in each chapter focus on a certain theme. First, you get yourself acquainted with the Clean IDE to create and compile a project, and use its search facilities (Chapter 1). This is followed by writing functions on the standard set of basic types (Chapter 2), using the overloading mechanism of Clean and creating non-recursive types yourself (Chapter 3), working with lists and list comprehensions (Chapter 4), higher-order functions (Chapter 5), manipulating the console and files (Chapter 6), create and manipulate recursive types yourself (Chapter 7), create correctness proofs via equational reasoning (Chapter 8), and work with dynamic types (Chapter 9). The exercises of the remaining chapters are somewhat different. In Chapter 10 you find a number of exercises in which you create the brain of a footballer and assemble your team to participate in the *SoccerFun* competition.

Within a chapter, the difficulty level of the exercises gradually increases. It is not necessary to complete an entire chapter before proceeding to the next one. Some exercises depend on other ones, also across chapters. In that case you find which other exercises are follow-up exercises of a particular assignment.

All assignments have a name. This name coincides with the name of the 'main module' of the corresponding Clean implementation. At each exercise you find a reference to the name of that main module and the Clean environment that is required to create and compile the project. All main modules have been made available in an *Exercises folder* that you can download from the course site as a *.zip* file. Extract this file into your Clean directory in the *Examples* folder. Besides the main modules, the *Exercises folder* also contains *test modules*. A test module with name `MainTest.icl` can be used to automatically test the exported functions from module `Main.icl`. Follow the instructions that can be found in `MainTest.icl`.

The Clean environment *StdEnv (64)* directs the compiler to the standard modules of Clean, the so-called *standard environment*. These are necessary to perform basic operations on numbers, booleans, characters, strings, lists, and arrays. The document *A Concise Guide to Clean StdEnv* [Achten(2011)] describes the standard environment.

The syntax of Clean is very similar to the syntax of Haskell. In *Clean for Haskell98 Programmers - A Quick Reference Guide* - [Achten(2007)] you can find a comparison of the language features displayed on one page for easy reference.

This workout of functional programming exercises is regularly edited to match the

material of the functional programming courses, and to improve and add exercises and test files. We invite you to submit remarks, questions, tips, spotted errors, and so on, in order for us to improve the assignments for future versions.

Acknowledgements The assignments, test files, and main modules were originally written in Dutch. It started in 2006 as a small collection of exercises and former exam questions of the functional programming courses and its many incarnations that I have had the pleasure to teach with *Rinus Plasmeijer*. It has been expanded on during the subsequent years. During this period my student assistants have been very helpful in giving feedback and suggesting corrections and improvements. *Marc Schoolderman* in particular has suggested and added many improvements and test files and cases. In 2016 it has been decided to translate the collection to English. For this effort I have been fortunate to rely on the assistance of *Thomas Churchman*, *Mart Lubbers*, *Camil Staps*, and *Thom Wiggers*. They have translated all exercises, main and test files of Chapters 1 upto 10.

Last but not least, I wish you a lot of fun while doing your functional workout.

Peter Achten
P.Achten@cs.ru.nl

Bibliography

[Achten(2007)] P. Achten. Clean for Haskell98 programmers - a quick reference guide -, July 13 2007. http://www.mbsd.cs.ru.nl/publications/papers/2007/achp2007-CleanHaskellQuickGuide.pdf.

[Achten(2011)] P. Achten. A concise guide to Clean StdEnv, August 26 2011. http://www.mbsd.cs.ru.nl/publications/papers/2010/CleanStdEnvAPI.pdf.

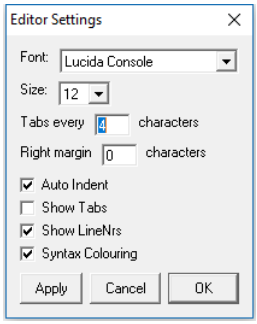
Contents

Chapter 1

Start

The exercises in this chapter serve to assist you in using the Clean IDE. They familiarize you with the various *windows* (such as project, editor, types, errors) that are used in the programming environment. Furthermore, it introduces useful navigation functionality, such as finding modules, functions, and types.

Tips:



- You can find the user manual for the Clean IDE with the command "Help:Help:UserManual.pdf".
- Line numbers are not shown with the default settings of the Clean IDE. It is useful to enable them on account of error messages. You can enable line numbers with the command: "Defaults:Window Settings:Editor Settings...". Tick the box "Show LineNrs".

1.1 Clean IDE Usage

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

Open the module `Start.icl`. Create a Clean project for it. You can do this with the command "File:New Project...". Confirm the project file as proposed by the Clean IDE (`Start.prj`) in the same directory. The content of `Start.icl` is:

```
module Start

import StdEnv

Start = expr0

expr0 = "Hello World!"
```

1. Choose "Project:Bring Up To Date (`Ctrl`+U)". What happens?
2. Choose "Project:Run". What happens?

3. Choose "Module:Module Options...". The dialog "Module Options" appears. Select the box near "Inferred Types". Press the "OK" button. Choose "File:Save Start.icl". Choose "Project:Update and Run (Ctrl+R)". What happens?

Add the following `expr`*i* `= ...` definitions from below step by step, and replace `Start =` `expr`*i*. Afterwards run "Project: Update and Run (Ctrl+R)". Explain what happens in every step.

```
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!']
```

- When one of the expressions gives a type error you can put it into comments by prefixing it with `//` (as in C).
- You can quickly jump to the location of the error with the command "Search:Goto Next Error... (Ctrl+E)", or by double-clicking the error message.

1.2 Searching in the Clean IDE

The Clean IDE provides many different ways to search for elements in your program: (definition and implementation) modules, types, functions, classes, etcetera. Start the Clean IDE and open the project `scrabble.prj` found in:

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

Bring the application *up to date* (Ctrl+U). After launching (Ctrl+R) you should see the *window* from Figure 1.1.

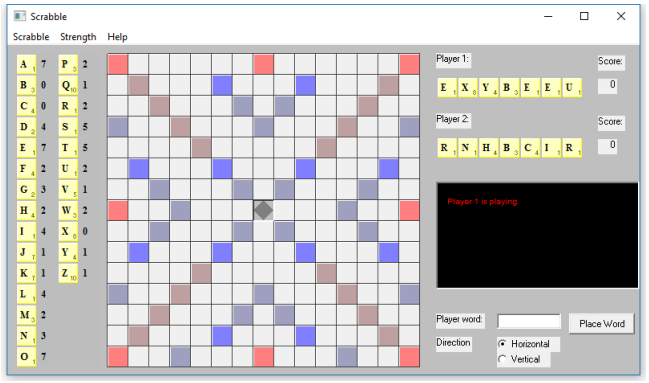


Figure 1.1: The initial window of scrabble.

Modules: the Project window

After an application has been brought *up to date*, the *project window* displays the modules of which the project consists. The main module is *always* displayed on top.

Modules reside in *directories*. You can show and hide modules from a directory by double clicking the directory name. By double clicking a module name the corresponding *definition* module opens, and with a *shift* double click on a module name you open the *implementation module*. With this method you can quickly find all definition and implementation modules.

If you prefer opening the implementation module without holding *shift*, you can configure this using the project window options (command "Defaults:Window Settings:-Project Window...").

You can switch between implementation (`icl`) and definition modules (`dcl`) swiftly by pressing `Ctrl`+/.

Exercise: Activate the scrabble project window and open the following modules:

1. Open the main module of scrabble.
2. Open the module `language.icl` of scrabble.
3. Open the module `StdOverloaded.dcl` from `StdEnv`.

Definitions in a project

Every Clean module consists of type definitions and function definitions. Modules can use definitions from other modules by *importing* the definition module that *exports* those definitions. In a module you can only use definitions created there yourself, or definitions that you imported. There are several ways to find definitions.

Within a module you can quickly find a definition by pressing Crtl+`. This opens a *pop up menu* displaying all (top-level) definitions in alphabetical order. The Clean IDE will place the cursor at the start of the line of the selected definition.

The command Crtl+= lets you search for a specific name (*identifier*) in all definition modules (Find Definition), implementation modules (Find Implementation) or identifiers (Find Identifier). You can limit your search to only the imported modules (Search in Imported Files), the project (Search in Project), or project paths (Search in Paths).

If you have selected a complete name (e.g. with a double click) you can find the definition module exporting that name by pressing Crtl+L, and the implementation module by pressing Crtl+`Alt`+L.

Exercise: Perform the following search queries:

1. Find the definition module exporting `isEven`. Which module did you find?
2. Activate the main scrabble module. Find all occurrences of the identifier `isEven` within the imported modules. How many did you find?
3. Activate the scrabble module `language.icl`. Again find all the occurrences of `isEven` within the imported modules. How many did you find?

Changing project paths

The language of the scrabble game is set to English in the distribution. However, there also exists a Dutch module, which resides in a different directory. Commonly occurring combinations of project paths are predefined in *environments* (an overview of all the environments is found in the "Environment" menu). You can switch between project

paths quickly by choosing a different environment. You can change project paths in the *Project Options* dialog (Project:Project Options...).
Exercise: Open the *Project Options* dialog and select the Project Paths *radio button*. Remove {Project}\Engels and add {Project}\Nederlands. Close the dialog. Recompile the entire application by pressing `Ctrl`+`Shift`+U. If all went well, the scrabble you have just compiled is in Dutch.

Chapter 2

Simple Functions

The exercises in this chapter are practices for reading and working with functional expressions (rewrite). A developed skill in working with expressions is essential for reasoning about functions and making correctness proofs (this will be thoroughly treated in Chapter 8). Additionally, this chapter contains exercises with a few classical standard functions (function composition, currying, flip), with which existing functions can be combined to form new functions. The remaining exercises are meant to illustrate how you can express recursive functions in a functional style.

2.1 Notation

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

Describe what each of the following Clean functions mean. Check your answer by adapting the `Start` line to call the corresponding function, with valid arguments if necessary. The function call a `+++` b glues string b after string a; a `rem` b computes the remainder after integer division of a with b; `fst` $(a,b) = a$; `snd` $(a,b) = b$.

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

f2          :: Int Int -> Int
f2 m n
| m < n     = m
| otherwise = n

f3          :: String Int -> String
f3 s n
| n <= 0    = ""
| otherwise = s +++ f3 s (n-1)
```

```
f4          :: Int Int -> Int
f4 x 0      = x
f4 x y      = f4 y (x rem y)

f5          :: (Int,Int) -> Int
f5 x        = fst x + snd x

f6          :: (a,b) -> (b,a)
f6 (a,b)    = (b,a)

f7          :: (a,a) -> (a,a)
f7 x        = f6 (f6 x)
```

2.2 Find the redex

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

Determine the result of the expressions below by rewriting. First add parentheses `()` in the expressions illustrating the priorities of the operations correctly. Then apply rewrites as shown in the lecture: start from `Start = e`$_i$, place every rewrite step on a new line, and underline the part of the expression that you rewrite (the *redex*, *reducible expression*).

```
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 gcd

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

The module `StdInt` implements the *greatest common divisor* function in the following way (the definition in `StdInt` is a bit different):

```
instance gcd Int where
gcd :: Int Int -> Int
gcd x y = gcdnat (abs x) (abs y)

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

Rewrite the following `Start`-functions:

```
Start1 = gcd -42 0
Start2 = gcd 0 -42
Start3 = gcd 18 42
Start4 = gcd 123456789 987654321
```

2.3.1 Evaluation strategies

Rewrite the above `Start`-functions again. First use the *normal order* evaluation strategy and afterwards use the *applicative order* evaluation strategy. Clearly denote which strategy you use.

2.4 Type inference and tuples

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

Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions. You can assume the following types from `StdEnv`: `fst :: (a,b) -> a` and `snd :: (a,b) -> b`. Please note that you can use the Clean compiler to check your answers.

```
f1 x        = x

f2 x
| x < 0     = -1
| x > 0     =  1
| otherwise =  0

f3 x        = (snd x, fst x)

f4 (x,y)    = (y,x)

f5 x y      = (x,y)

f6 x y      = (x,(y,y),x)

f7 (x,(y,z)) = ((x,y),z)
```

2.5 Type inference and strings

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

Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions. You can assume the following types from `StdEnv`: `size :: String -> Int`, `isUpper :: Char -> Bool`, `isLower :: Char -> Bool`. Furthermore, in an expression x`.[`y`]`, x has type `String`, y has type `Int`, and the result has type `Char` (the character at index position y in x). Finally, in an expression x `:=` (y,z), x has type `String`, y has type `Int`, z has type `Char`, and the result has type `String` (the same string as x, except that the character at index position y has been replaced by z). Please note that you can use the Clean compiler to check your answer.

```
f1 x y z
| z < 0 || z >= size x = -1
| x.[z] == y           = z
| otherwise            = f1 x y (z+1)

f2 x y                 = f1 x y 0

f3 x y
| y < 0 || y >= size x = x
| isUpper x.[y]        = f3 (x := (y,toLower x.[y])) (y+1)
| isLower x.[y]        = f3 (x := (y,toUpper x.[y])) (y+1)
| otherwise            = f3 x (y+1)

f4 x                   = f3 x 0

f5 x y z
| y > z                = x
| y < 0                = x
| z >= size x          = x
| otherwise            = f5 (x := (z,x.[y]) := (y,x.[z])) (y+1) (z-1)
```

```
f6 x                    = f5 x 0 (size x - 1)
```

2.6 Matching strings *(used in 6.4)*

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

The standard environment of Clean provides various functions for working with texts of type `String`. In this exercise you are only allowed to use the following functions as `String` operations:

- `size`: returns the number of characters of the given `String`.
  Example: `size ""` returns 0.
  Example: `size "0123456789"` returns 10.

- `.[]`: returns the `Char` value on the given `Int` index in a string. The indices of a non-empty string s go from 0 up to and including `(size s)-1`. The empty string `""` does not have any valid indices.
  Example: `"0123456789".[4]` returns `'4'`.
  Example: `"0123456789".[-1]` returns a *Run Time Error: index out of range.*
  Example: `"0123456789".[10]` returns the same error as above.

- `%`: takes a *slice* of a `String`.
  Example: `"0123456789" % (0,2)` returns `"012"`.
  Example: `"Madam, I'm Adam" % (7,12)` returns `"I'm Ad"`.

Using only the above functions, write the following ones:

1. Write the two functions `head :: String -> Char` and `tail :: String -> String` that given a non-empty `String` return the first and the remaining elements of the `String` respectively. If the argument is the empty `String`, it should show an error.
   Example: `head "Madam, I'm Adam"` = `'M'`.
   Example: `tail "Madam, I'm Adam"` = `"adam, I'm Adam"`.

2. Write a function `is_equal` that determines the equality of two `String` arguments. This means that for every pair s_1, s_2 of type `String` the following should hold: `is_equal` s_1 s_2 = s_1 `==` s_2.

3. Write a function `is_substring` that returns a `Bool` when given two `String` arguments. The result should be `True` if and only if the first argument is a *substring* of the second argument.
   Example: `is_substring "there" "Is there anybody out there?"` [1] returns `True`, after all: `"Is there anybody out there?"`.
   Example: `is_substring "there" "Just for the record"` [2] returns `False` because there is a space between `the` and `re`.

[1] Pink Floyd – The Wall (1979)
[2] Marillion – Clutching at straws (1987)

4. Write a function `is_sub` that returns a `Bool` when given two `String` arguments. The results should be true `True` if and only if the characters of the first argument appear in the same order in the second argument. Characters of the second argument may be skipped.
   Example: `is_sub "there" "Just␣for␣the␣record"` returns `True` because the space will be skipped.
   Example: `is_sub "she␣and␣her" "Is␣there␣anybody␣in␣there?"` returns `True`, after all: `"Is there anybody in there?"`.
   Example: `is_sub "There␣there"` [3] `"Is␣there␣anybody␣in␣there?"` returns `False` because `T` is no part of the second `String`.

5. Write a recursive function `is_match` that returns a `Bool` when given two `String` arguments (*pattern* and *source*). The function determines whether the *pattern* can be applied on the source (the pattern *matches* the source). The *pattern* may contain *wildcard* characters:

   - `.`: this matches the next, arbitrary single character in the *source*.
   - `*`: this matches zero or more consecutive characters in the *source*.

   All other characters in the *pattern* have to match exactly with the corresponding characters in the *source*.
   Example: `is_match "here" "here"` returns `True`.
   Example: `is_match "here" "␣here"` returns `False` because the first characters of the two strings do not match.
   Example: `is_match "here" "here␣"` returns `False` because the second text has one extra character.
   Example: `is_match ".." "AB"` returns `True` because the second text consists of exactly two characters.
   Example: `is_match "*?" "Is␣there␣anybody␣in␣there?"` returns `True` because the second text ends with a `'?'` character.
   Example: `is_match "*?*!*?*!" "Answers?␣Questions!␣Questions?␣Answers!"` [4] returns `True`.
   Example: `is_match "*.here*.here*." "Is␣there␣anybody␣in␣there?"` returns `True`.
   Example: `is_match ".here.here." "Is␣there␣anybody␣in␣there?"` returns `False`.

2.7 Determining prime numbers *(used in 2.8)*

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

Write the *recursive* function `isPrime :: Int -> Bool` that determines whether the argument is a prime number. A prime number is a positive integer divisible by exactly two *different* positive integers; namely itself and 1 (hence the reason why 1 is not a prime itself). Test the function with `Start = [x \\ x <- [1 .. 1000] | isPrime x]`. This should return all primes in the range of one up to and including a thousand. The result should be:

> [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,

[3] Radiohead – Hail to the thief (2003)
[4] Focus – Focus III (1973)

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]

Note: Use the function `rem` for integer division with remainder. The function `mod` exists, but it is not defined for integers.

2.8 Prime factors

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

Every integer $x > 1$ can be decomposed in prime factors in a canonical way. This means that you can find a range of prime numbers $p_0 \ldots p_k$ where $k \geq 0$ for which $p_i < p_j$ for all $i < j$ and as many positive numbers $n_0 \ldots n_k$ in such a way that:

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

Examples:

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

Write the function `primeFactors :: Int -> String` that decomposes an integer into its prime factors. The prime factors found should be glued together in the resulting `String`.

Examples:

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

Use the function `isPrime` from Exercise 2.7.

The following operations on `String`s and `Int`s are useful: s_1 `+++` s_2 glues `String` s_2 to `String` s_1; `toString` n creates a `String` value from an `Int`.

2.9 Adding numbers

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

Write the function `numbersum :: Int -> Int` that sums the digits of a positive integer.
Example: `numbersum 9876543` = 9+8+7+6+5+4+3 = 42.
Example: `numbersum 1000` = 1+0+0+0 = 1.

2.10 O Tannenbaum (used in 6.2)

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

1. Write the function `triangle` that receives an `Int` argument n. This function will *draw* a triangle as shown on the right with $n = 5$. The *drawing* is actually returning a `String` of which every *line* ends with a *newline*.
   Hence, the string corresponding with the image to the right is: `"␣␣␣␣*\n␣␣␣***\n␣␣*****\n␣*******\n*********\n"`

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

2. Write the function `christmastree` that receives an `Int` argument n. This function *draws* a Christmas tree as a `String` (as shown on the right with $n = 4$). For every sub triangle the function should use a generalized version of the above defined `triangle` function.

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

The following operations on `String`s and `Char`s are useful: s_1 `+++` s_2 glues `String` s_2 to `String` s_1; `toString` c creates a `String` value from a `Char` c. `'\n'` is the *newline* character. To get program output without the `String` quotation marks you can select the option "Basic Values Only" in the Clean IDE dialog "Project:Project Options...".

2.11 Zeros and Ones

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

`StdEnv` introduces instances of the type classes `zero` and `one` for the types `Int` and `Real`. In module `ZeroOne` instances are defined for tuples:

```
instance zero (a,b) | zero a & zero b
   where zero = (zero,zero)
instance one (a,b) | one a & one b
   where one = (one, one)
```

Additionally, you find a number of functions (`f1` ...`f4`, `g1` ...`g4`) that use these instances. For each function, rewrite the program that starts with that function (so, `Start = f1` ...`Start = g4`).

2.12 Overloading and (,)

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

Equality is defined for *tuples* `(,)` and *triplets* `(,,)` under the condition that equality is defined for the components. Define in an analogue way instances for *tuples* `(,)` and *triples* `(,,)` for the overloaded functions `+`, `-`, `*`, `/`, `~`, `zero`, and `one`. The implementations should conform to the following properties:

```
∀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
```

You can easily test this by adding a `Start` line that calls the function `test` for several values of type `(,)` and `(,,)`:

```
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
         )
```

2.13 Trajectory

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

An object, such as a ball, which is shot away from a flat surface with an angle of $\theta_0 (0 < \theta_0 \leq \frac{\pi}{2})$ [5] and initial velocity $v_0 (0 < v_0)$, follows an arclike trajectory influenced by gravity [6]. If we ignore effects such as air resistance and wind, the trajectory of the ball is a curve defined in a vertical flat surface in which the x-axis shows the distance to the starting position on the ground and the y-axis shows the height of the ball.

Because we ignore air resistance, the velocity of the ball in the x-direction ($v_x(t)$) is constant. The velocity of the ball in the y-direction ($v_y(t)$) depends on the time t and the gravity $g = 9.81^{m}/_{s^2}$:

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

You can calculate the distance $x(t)$ and the height $y(t)$ of the ball on time t as follows:

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

Height h, expressed in distance x is:

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

The ball reaches the highest point at time $t_{\max y}$:

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

[5] Angle in radians.

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

Thus you can find the maximal height reached at that time by plugging t_{maxy} into $y(t)$. The ball takes just as much time to reach the highest point as it takes to get back on the ground. Therefore the ball is airborne for $t_2 = 2 \cdot t_{\max y}$ seconds. You can find the distance the ball travels by plugging t_2 into $x(t)$. The velocity of the ball when touching the ground can be found by plugging t_2 into $v_x(t)$ and $v_y(t)$. Then, the combined value is:

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

Functions Implement the aforementioned functions v_x, v_y, x, y and h in Clean with the respective names `v_x`, `v_y`, `x_at`, `y_at` and `h`. The equations given above are parameterized with the initial velocity v_0 and angle θ_0 and need to be passed as parameters to the Clean functions.

Best angle Write a *recursive* function `best_angle` that, given an initial velocity v_0, calculates the angle $\theta \in \{\frac{1}{100}\pi, \frac{2}{100}\pi \ldots \frac{50}{100}\pi\}$ in such a way that when the ball is shot away with velocity v_0 and angle θ it travels the *greatest* distance.

Experiment with different values for v_0. Does this result in the same return value?

2.14 This old man

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

Implement the function `this_old_man :: String` that produces the following poem.

This old man, he played one,
He played knick-knack on my thumb;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played two,
He played knick-knack on my shoe;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played three,
He played knick-knack on my knee;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played four,
He played knick-knack on my door;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played five,
He played knick-knack on my hive;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played six,
He played knick-knack on my sticks;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played seven,
He played knick-knack up in heaven;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played eight,
He played knick-knack on my gate;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played nine,
He played knick-knack on my spine;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played ten,
He played knick-knack once again;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

Try to make the program as short as possible by abstracting away from patterns in the text and writing suitable functions for those.

The following operations on `String`s and `Char`s are useful: s_1 `+++` s_2 glues `String` s_2 to `String` s_1; `toString` c creates a `String` representation of a `Char` c; `toString` t creates a `String` representation of an `Int` n; `'\n'` is the *newline* character (type `Char`), `""` is the *empty string* (type `String`). To suppress the `String` quotation marks in the output you can select in the Clean IDE dialog "Project:Project Options..." the option "Basic Values Only".

2.15 Stringnum (used in 2.16,3.16)

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

In the module `Stringnum` you develop functions to work with `String`s representing arbitrarily large non-negative integers. The functions are:

- `isAStringNum` tests whether the given `String` represents a correct positive integer (including zero).
- The functions `smaller`, `bigger` and `equal` implement the operations `<`, `>` and `==`.
- The functions `plus`, `decrement`, `times` and `divide` implement the operations `+`, `-`, `*` and /, where `decrement` a b = `"0"` if `bigger` b a.

The following functions on `String`s and `Char`s are useful: `isDigit` c tests whether a `Char` c is a digit; `toInt` c returns the ASCII-code of `Char` c; `size` s returns the length of the `String` s; `+++` glues two `String`s together; s`.[`i`]` returns the character from s at index i; s`%(`i`,`j`)` returns the slice from s from index i up to but not including j; s`:=(`i`,`c`)` replaces the character in s at index i with c.

2.16 Stringnum2 (used in 3.17)

Main module:	*Stringnum2.icl*
Environment:	*StdEnv*

If you can calculate with arbitrarily large non-negative integers, then, using these operations, you can also construct the implementations for all arbitrarily large integers (negative, positive, and 0). In this implementation, use the exported functions from `Stringnum` to create a new module `Stringnum2` implementing the same operations for all integers. Add the following operations besides the existing ones:

- `isNegative` tests whether the given number is negative (< 0).
- `absolute` calculates the absolute value of the argument.
- `changeSign` flips the sign of the number.

2.17 Time (used in 3.18)

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

The length of movies and music is often displayed in the $m^+ : ss$ format where m^+ is the number of minutes and ss the number of seconds. For example, the length of the movie *"The Lord of the Rings: The Fellowship of the Ring"* is 178:00, i.e. two hours and 58 minutes. The length of the song *"Tea"* of Sam Brown on the album *"Stop!"* is 0:41, i.e. 41 seconds. Moreover, time in between songs (silence) is often indicated via negative time. For instance, three seconds of silence is -0:03. Implement the following two functions:

- `showtime` n: with n the number of seconds (positive or negative), this function shows the time as explained in the above format;

- `parsetime` x: with x a string, this function attempts to extract the format explained above and return the corresponding (positive or negative) number of seconds. If x is ill formatted, then the result should be zero.

Chapter 3

Non-recursive Types

Algebraic types and *records* are the eminent tool for modeling new data structures in functional languages. New data structures play an essential role with *overloading*, as it is type-driven. Algebraic types enable you to introduce new constants in your programs, generally increasing the readability and correctness. Record types allow you to manipulate collections of values without knowing exactly where the values reside in the collection, nor where they are defined.

3.1 ADT Notations *(used in 3.2)*

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

A number of algebraic types are defined below. If possible, give three *different* expressions (not the trivial expressions `abort` and `undef`) for each of these types. Examples of expressions of type `Int` are `1`, `2` and `1 + 1`.

```
:: Day      = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
:: Nat      = Nat Int
:: PosNat   = Pos Int | NotAPosNat
:: Number   = Whole Nat | Decimal Real
:: Void     = Void
```

3.2 Type inference and ADTs

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

Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions, using the definitions of exercise 3.1. Please note that you can use the Clean compiler to check your answer.

```
f1 Saturday  = True
f1 Sunday    = True
f1 _         = False

f2 Monday    = 1
```

```
f2 Tuesday   = 2
f2 Wednesday = 3
f2 Thursday  = 4
f2 Friday    = 5
f2 Saturday  = 6
f2 Sunday    = 7

f3 x y       = f2 x == f2 y

f4 x         = Nat x

f5 (Nat x)   = x

f6 x
| x > 0      = Pos x
| otherwise  = NotAPosNat

f7 (Pos x) (Pos y) = Pos (x+y)
f7 _ _             = NotAPosNat

f8 (Pos x) (Pos y)
| x > y            = Pos (x-y)
| otherwise        = NotAPosNat
```

3.3 Record Notations *(used in 3.4)*

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

A number of record and algebraic data types are defined below. Give one expression (not the trivial expressions `abort` and `undef`) for each of the *record* types.

```
:: Gender = Male | Female
:: Month  = JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC
:: Date   = {day :: Int, month :: Month, year :: Int}
:: Name   = {given :: String, suffix :: String, family :: String, postfix :: String}
:: Person = {name :: Name, birth :: Date, gender :: Gender, father :: Name, mother :: Name}
```

3.4 Type inference and records

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

Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions, using the definitions of exercise 3.3. Please note that you can use the Clean compiler to check your answer.

```
f1 {given,suffix,family,postfix}
    = given +++ "␣" +++ suffix +++ "␣" +++ family +++ "␣" +++ postfix

f2 = {given="", suffix="",family="",postfix=""}
```

```
f3 a b = {f2 & given = a, family = b}

f4 p=:{gender=Male}
| p.name.given == p.father.given = True
f4 p=:{gender=Female}
| p.name.given == p.mother.given = True
f4 _ = False

f5 p
| f4 p      = {p & name = {p.name & postfix = "junior"}}
| otherwise = p
```

3.5 Day (*used in 3.10*)

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

Module `Day` defines the following algebraic data type:

```
:: Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Write the following functions and test them with an appropriate `Start`-function:

1. `friday_on_my_mind :: Day -> Bool` which yields `True` only if the argument is `Friday`.
2. `is_weekend :: Day -> Bool` which yields `True` only if the argument is `Saturday` or `Sunday`.
3. `(on_my_mind) :: Day Day -> Bool` which yields `True` only if the two arguments are equal. For instance, `Friday on_my_mind` d is `True` only if $d =$ `Friday`.
4. `yesterday :: Day -> Day` and `tomorrow :: Day -> Day` that return the previous day and next day of their argument respectively.

3.6 Date (*used in 3.11*)

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

Module `Date` defines the following types:

```
:: Date  = { year :: Int, month :: Month, day :: Int }
:: Month = January | February  | March   | April    | May | June | July
         | August  | September | October | November | December
```

Implement the following functions on these types:

1. `is_leap_year` y determines whether year y is a leap year. In a leap year, month february has 29 days instead of the usual 28. A year is a leap year if it is dividable by 4, except if it is a multiple of 100 that is not dividable by 400. For instance, 1600 is a leap year, but 1700 is not a leap year.
2. `no_of_days` y m determines the number of days of month m in year y (taking leap years into account).
3. `yesterday` and `tomorrow` that return the date of the previous day and next day respectively (taking leap years into account).

3.7 Fractions(*used in 3.12*)

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

Choose a suitable representation for the type `Q` in module `Q.icl` to implement rational numbers $\mathbb{Q}$, also known as fractions. The following operations on `Q` values should be supported:

- The functions `equalQ` and `smallerQ` test whether two fractions are equal or smaller respectively.
- The operations `plusQ`, `decrementQ`, `timesQ`, and `divideQ` implement the mathematical operations $+$, $-$, $\cdot$, and $\div$.
- Just like integers and reals, fractions are signed numbers (can be negative, positive, or zero). `absoluteQ` takes the absolute value, `signOfQ` returns the sign of the argument and `negateQ` flips the sign.

  Warning: be aware that the absolute value of *min-int* is equal to *min-int*!
- `isIntQ` tests whether the argument corresponds to a whole number, and yields `True` only in that case. `IQ` creates a fractional value of its `Int` argument. `QR` creates the floating point value that approximates the fractional `Q` argument.

In your implementation you can use the `Int` instance of the function `gcd`, which calculates the *greatest common divisor* of two positive integers. This function is defined in the `StdInt` module, which you get automatically when you import `StdEnv`.

3.8 Numerals(*used in 3.13*)

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

Clean distinguishes between whole numbers (`Int`) and floating point numbers (`Real`) in the type system. Complete the implementation module corresponding with `StdNum.dcl`. This module exports the operations for working with numbers that are either an `Int` or a `Real`.

> **Optional:** If you did Exercise 3.7, you can also add rational numbers to this implementation module.

In the implementation you need to choose the most precise representation, thus you should not just convert values to `Real`s. The following operations on `Num` values should be supported:

- The functions `equalNum` and `smallerNum` test whether two fractions are equal or smaller respectively.
- The operations `plusNum`, `decrementNum`, `timesNum`, and `divideNum` implement the mathematical operations $+$, $-$, $\cdot$, and $\div$.
- Just like integers and reals, numerals are signed numbers (can be negative, positive, or zero). `absoluteNum` takes the absolute value, `signOfNum` returns the sign of the argument and `negateNum` flips the sign.

3.9 Card (used in 3.14)

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

A standard card deck consists of 52 cards. Every card has a *suit* and a *value*. The suits are *heart* (♡), *diamond* (♢), *spade* (♠), and *club* (♣). The possible values are the numbers 2 up to and including 10, and *jack*, *queen*, *king*, and *ace*. Implement the following components and use algebraic data types and records where possible.

1. **Representation** Design suitable data structures to represent a deck of cards. The types representing a card, suit, and value should be named `Card`, `Suit`, and `Value` respectively. You *must* chose an algebraic type *or* a record type to create `Card`. You are free to choose your representation for `Suit` and `Value`.

2. **Equality of cards** Write the function `equalCard` that returns `True` only if the two arguments are the same card.

3. **Printing and parsing** Write the functions `showCard` and `parseCard`. The `parseCard` function should be the inverse of the `showCard` instance.

   $\forall c$ `::Card: parseCard (showCard` c`)` $= c$.

   Can you also make sure that the following holds for every `String` s?

   let `c ::Card; c = parseCard` s **in** `showCard c` $= s$?

3.10 StdDay

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

In exercise 3.5 the type `Day` and a number of operations have been defined. Use this implementation in the module `StdDay` to make the operations available as instances of the overloaded operators `==` (`on_my_mind`), and the overloaded functions `previous` (`yesterday`) and `next` (`tomorrow`) that are defined in module `PrevNext`.

3.11 StdDate

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

In exercise 3.6 the types `Date` and `Month` and a number of operations have been defined. Use this implementation in the module `StdDate` to make the operations available as instances of the overloaded operators `==`, `<` and the overloaded functions `previous` and `next` that are defined in module `PrevNext`.

3.12 StdQ

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

In exercise 3.7 you have defined a type `Q` to manipulate fractions. Use this implementation

in the module StdQ to make the operations available as instances of the overloaded operators == (equalQ), < (smallerQ), + (plusQ), - (decrementQ), * (timesQ), / (divideQ), abs (absoluteQ), sign (signOfQ), ~ (negateQ).

Furthermore, module StdQ adds the following new instances: zero, one, toInt, toReal, toQ, and toString. All together, the following properties should hold:

- The operation == and < test equality and less-than.
- The operations +, -, * and / are the usual arithmetical operations on the rational numbers. The values zero and one are the neutral elements of addition and multiplication respectively.
- abs takes the absolute value, sign returns the sign of the argument and ~ flips the sign.

  Warning: be aware that the absolute value of *min-int* is equal to *min-int*!
- isInt tests whether the argument corresponds to a whole number. If this is true, the Q instance for toInt results in exactly the same number (no rounding). toReal converts the rational number to a Real (by approximation).

  The instances of the class toQ have the reverse functionality: Int and Real convert the argument of their type (by approximation) to a rational number. For the (Int, Int) instance, toQ (t, n) should create the rational number $\frac{t}{n}$. It throws an error message in case $n = 0$. For the (Int, Int, Int) instance, toQ (d, t, n) should create the rational number $d\frac{t}{n}$ (assuming $d \neq 0$ and $0 < t < n$). It throws an error message in case $d = 0$ or $t \leq 0$ or $n \leq 0$ or $n \leq t$.

In your implementation you can use the Int instance of the function gcd, which calculates the *greatest common divisor* of two positive integers. This function is defined in the StdInt module, which you get automatically when you import StdEnv.

3.13 StdNum

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

In exercise 3.8 you have defined a type Num to manipulate various kinds of numbers: whole numbers, floating point numbers, and, optionally, fractions. Use this implementation in the module StdNum to make the operations available as instances of the overloaded operators == (equalNum), < (smallerNum), + (plusNum), - (decrementNum), * (timesNum), / (divideNum), abs (absoluteNum), sign (signOfNum), ~ (negateNum).

Furthermore, module StdNum adds the following new instances: zero, one, toInt, toReal, fromNum, toNum, and toString. If you have included fractions in exercise 3.8, then StdNum also implements toQ.

3.14 StdCard *(used in 4.30)*

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

In exercise 3.9 you have defined new types (Card, Suit, Value) to represent cards. Use them in module StdCard.icl to make the following overloaded operations available: == (equalCard), toString (showCard), and fromString (parseCard).

3.15 Overloading and records

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

Define the following record type for two-dimensional vectors:

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

Implement the classes `==`, `zero`, `one`, `+`, `-`, `*`, /, and `~` for the new type in a similar manner as in Exercise 2.12.

3.16 StdStringnum

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

In Exercise 2.15 you created a type and operations for arbitrary large non-negative integers. Use this implementation in the module `StdStringnum` to make the operations available as instances of the overloaded operators `<` (`smaller`), `==` (`equal`), `+` (`plus`), `-` (`decrement`), `*` (`times`), and / (`divide`). To this end, define a new type `Stringnum` in `StdStringnum.icl`:

```
:: Stringnum = Stringnum String
```

Also define instances for the overloaded functions `zero` and `one`, and the conversion functions `fromInt`, `toString`, and `fromString`.

3.17 StdStringnum2

Main module:	*StdStringnum2.icl*
Environment:	*StdEnv*

In Exercise 2.16 you created a type and operations for arbitrarily large integers (negative, positive, and zero). Use this implementation in the module `StdStringnum2` to make the operations available as instances of the overloaded operators `abs` (`absolute`), `~` (`changeSign`), `sign`, `<` (`smaller2`), `==` (`equal2`), `+` (`plus2`), `-` (`decrement2`), `*` (`times2`), and / (`divide2`). To this end, define a new type `Stringnum2` in `StdStringnum.icl`.

```
:: Stringnum2 = Stringnum2 String
```

Also define instances for the overloaded functions `zero` and `one`, and the conversion functions `fromInt`, `toString`, and `fromString`.

3.18 StdTime (*used in 5.17,6.4*)

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

In exercise 2.17 you have implemented two conversion function, `showtime` and `parsetime` that convert (positive and negative) number of seconds to strings and vice versa. Use these functions in module `StdTime.icl` to create the abstract type `Time` and the following functions:

- Comparison operations: `==` and `<`.

- Arithmetical operations: `zero`, `abs`, `~`, `+`, `-`.
- Conversion to seconds: `toInt`, `fromInt`.
- Conversion to text: `toString` (`showtime`), `fromString` (`parsetime`). If a text `s` does not adhere to the m^+ss format, then `fromString s` = `zero`.

Chapter 4

Lists

Lists are one of the most widely used recursive data types in functional languages. The exercises in this chapter teach you how to create and manipulate lists, and how to work with so-called *list comprehensions*. Lists make extensive use of the *lazy* nature of Clean. This enables the programmer to work with potentially infinite lists and to postpone certain calculations that they might need in the future.

4.1 List notation

Main module:	*NotationLists.icl*
Environment:	—

In this exercise you practice reading list notation. Several lists are written down below. For every list do the following:

- explain in your own words which list is written down;
- give the number of elements in the list;
- give the shortest notation that results in an equal list (this could be the same expression); and
- give the result of the standard functions `hd`, `tl`, `init`, and `last` on the list. You can find these functions in the module `StdList.icl`.

These are said lists:

```
11    = []
12    = [1000:[]]
13    = [[]]
14    = [[]:[]]
15    = ['a':['b':['c':['d':['e':[]]]]]]
16    = [[1],[],[2,3]:[[4,5,6]]]
17    = [[[1,2],[3,4]],[],[[],[],[5,6]]]
18    = [1:2:3:4:5:[]]
```

4.2 Types and values

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

In this exercise you are given a number of list types. Give one expression for each type that corresponds to that type. The expression may not be trivial (e.g. `abort`, `undef`, `[]`).

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

e2 :: [Bool]
e2 = ...

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

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

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

4.3 Type inference and lists

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

Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions. Please note that you can use the Clean compiler to check your answer.

```
f1         = []

f2         = [[]]

f3 x       = [x]

f4 (x,y)   = [x,y]

f5 (x,y)   = [(x,y)]

f6 x       = [[x]]

f7 [x,y:z] = (x,y)

f8 [x:y]   = (x,y)

f9 [x]     = x
f9 [x,y]   = x
f9 [x:xs]  = f9 (init xs)
```

4.4 The first will be the last

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

The purpose of this assignment is that you practice writing recursive functions on lists yourself, using *guards* and *pattern-matching*. Therefore, do not use any of the pre-defined functions that are available in `StdList`.

First two and last two Write the function `first2` and `last2` that both get a *list* as argument and return a *list*. The type of both functions thus is: `[a] -> [a]`. The function `first2` returns the first two elements of the argument and `last2` returns the last two elements. Generate an error using the function `abort` when the argument lists are too short.

Examples:

```
first2 [42]          =  "List is too short."
first2 [1,2,3,4,5]   =  [1,2]
last2 [42]           =  "List is too short."
last2 [1,2,3,4,5]    =  [4,5]
```

Calculate In this exercise you calculate the results of a number of functions using rewriting with pen and paper. The functions used are `first2` and `last2`, or are from `StdEnv`. Write the rewrite steps in the same way as presented in the lecture: start from `Start`, start every step on a new line, and underline the part you are going to rewrite. Examples can be found in the lecture slides.

1. `Start = hd (hd [[[1,2,3],[4]],[[5],[6]]])`
2. `Start = hd (tl [1,2,3,4,5])`
3. `Start = first2 [[1],[],[2,3],[4,5,6]]`
4. `Start = last2 [[1],[],[2,3],[4,5,6]]`

First *n* and last *n* Write the recursive functions `firstn` and `lastn` that both get a number and a *list* as argument and return a *list*. The type of both functions thus is: `Int [a] -> [a]`. The function `first n lst` returns the first `n` elements of list `lst` and `lastn n lst` returns the last `n` elements of list `lst`. Generate an error using the function `abort` when the argument lists are too short.

Examples:

```
firstn 4 [42]          =  "List is too short."
firstn 4 [1,2,3,4,5]   =  [1,2,3,4]
lastn 4 [42]           =  "List is too short."
lastn 4 [1,2,3,4,5]    =  [2,3,4,5]
```

Properties Complete the following statements:

$$
\begin{array}{lllllllll}
\forall 0 \leq n, & xs :: [\mathtt{a}] : \mathtt{firstn} & n & (\mathtt{firstn} & n & xs) & = & \ldots \\
\forall 0 \leq n, & xs :: [\mathtt{a}] : \mathtt{firstn} & n & (\mathtt{lastn} & n & xs) & = & \ldots \\
\forall 0 \leq n, & xs :: [\mathtt{a}] : \mathtt{lastn} & n & (\mathtt{firstn} & n & xs) & = & \ldots \\
\forall 0 \leq n, & xs :: [\mathtt{a}] : \mathtt{lastn} & n & (\mathtt{lastn} & n & xs) & = & \ldots \\
\forall 0 \leq m \leq n, & xs :: [\mathtt{a}] : \mathtt{firstn} & m & (\mathtt{firstn} & n & xs) & = & \ldots \\
\forall 0 \leq m \leq n, & xs :: [\mathtt{a}] : \mathtt{length} & & (\mathtt{firstn} & m & xs) & ? & \mathtt{length}\ (\mathtt{firstn}\ \mathtt{n}\ \mathtt{xs})
\end{array}
$$

4.5 Glueing lists with ++

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

List concatenation (`++`) is defined as follows (see module `StdList.icl`):

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

Calculate the results of the expressions below by rewriting the `++` operator. The values of x_i are irrelevant. Part **6** is challenging!

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

Use the results of part **5** and **6** to argue why the `++` operator is *right associative*.

4.6 Glueing lists with `flatten`

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

The `flatten` operation on lists is defined as follows (see module `StdList.icl`):

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

Calculate the results of the expressions below by rewriting the `flatten` and `++` functions. The values of x_i are irrelevant. Use the results found in **2** in part **3**.

1. `flatten [[`x_0`,`x_1`,`x_2`], [`x_3`,`x_4`], [`x_5`], []]`
2. `flatten [[[`x_0`,`x_1`,`x_2`], [`x_3`,`x_4`]], [], [[`x_5`], []]]`
3. `flatten (flatten [[[`x_0`,`x_1`,`x_2`], [`x_3`,`x_4`]], [], [[`x_5`], []]])`

4.7 List generators

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

The purpose of this assignment is that you practice writing recursive functions that generate lists yourself, using *guards* and *pattern-matching*. Therefore, do not use any of the pre-defined functions that are available in `StdList` or the language support to create lists (list comprehensions and ..-expressions). Assignment 4.9 is used for that purpose.

Write the following functions that generate lists:

- `all_elements` x calculates the infinite list $[x, x, x \ldots]$.
  Example: `all_elements 42` = `[42, 42, 42, 42, ...]`.

- `step` x calculates the infinite list $[x$, x+`one`, x+`one`+`one`, $\ldots]$.
  Example: `step -10` = `[-10, -9, -8, -7 ...]`.
  Example: `step 'a'` = `['a','b','c','d' ...]`.

- `from_step` x z calculates the infinite list $[x, x+z, x+z+z, \ldots]$.
  Example: `from_step 0 -2` = `[0, -2, -4, -6 ...]`.

- `from_to` x f (with $x \leq y$) calculates the list $[x, x$+`one`$, x$+`one`+`one`$, \ldots y']$ in such a way that $y' \leq y$ and y'+`one` $> y$.
  `from_to x y` (with $x > y$) returns the empty list.
  Example: `from_to 'a' 'z'` = `['abcdefghijklmnopqrstuvwxyz']`.
  Example: `from_to 0.5 7.0` = `[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5]`.
  Example: `from_to 3 3` = `[3]`.
  Example: `from_to 3 0` = `[]`.

- `from_to_step` x y z calculates the list $[x, x+z, x+z+z, \ldots, y']$ in such a way that $x \leq y \wedge z >$`zero`: $y' \leq y \wedge y'+z > y$ and if $x \geq y \wedge z <$`zero`: $y' \geq y \wedge y'+z < y$.
  For all other cases the result is the empty list.
  example: `from_to_step 0 -10 -2` = `[0, -2, -4, -6, -8, -10]`.
  example: `from_to_step 0 -10 -3` = `[0, -3, -6, -9]`.
  example: `from_to_step 0 -10 -42` = `[0]`.
  example: `from_to_step 0 -10 2` = `[]`.

Make sure that the function types are as general as possible.

4.8 ZF-notations

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

For all the functions below explain what they calculate, and additionally give the most general type of each of the functions.

```
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]
```

4.9 List generators, part II

Main module:	*ListGenerator2.icl*
Environment:	*StdEnv*

Write the same functions as in Exercise 4.7, but now only use *list comprehensions*. It is possible that the type of the functions has to be adapted. Verify and compare the results of the new implementation with the examples from Exercise 4.7.

4.10 !! and ??

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

The operator `(!!)` **infixl** `9 :: ![a] !Int -> a` from `StdList` selects the ith element from a list xs after calling xs `!!` i (counted from zero). If xs does not contain enough elements an error message is generated.

Implement using list comprehensions the operator

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

that searches for the index of x in a list xs, if x is a member of xs. If there is no such value, then xs `??` x should return `-1`. If xs is a list containing element x, then the following must hold: xs `!!` (*xs* `??` x) $= x$. If x is not an element of xs, then xs `!!` (*xs* `??` x) results in an error message.

Examples:

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

4.11 List comprehensions and `removeAt`

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

Consider the function `removeAt` from the module `StdList`. Write, using list comprehensions, the function `removeAt2` that, given the same arguments, returns the same result as `removeAt`. In other words:

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

4.12 List comprehensions and `updateAt`

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

Consider the function `updateAt` from the module `StdList`. Write, using list comprehensions, the function `updateAt2` that, given the same arguments, returns the same result as `updateAt`. In other words:

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

4.13 Fibonacci

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

4.13.1 The Fibonacci function

The *Fibonacci* sequence $F = F_0, F_1, F_2, \cdots$ is defined as:

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

Write the recursive function `fibonacci` as above with the meaning: `fibonacci n` $= F_n$. For which values is this function defined?

Test your function with incrementing values:

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

Describe and explain the execution behaviour of the program.

4.13.2 Sumlists

Write the function `sumLists` with the following meaning:

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

You need to choose your own behaviour for lists that are of unequal length. Motivate that choice. Test your function with some representative sample inputs. Write what inputs you chose and give the result.

4.13.3 The Fibonacci sequence(used in 4.20)

Another way of thinking about the fibonacci sequence F 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 ...]  sumLists F (tl F)
```

This allows you to compute the first N fibonacci numbers as follows:

```
Start    = take N fibs
where
   fibs = [1,1:sumLists fibs (tl fibs)]
```

Describe and explain the execution behaviour of the program. Compare it to the behaviour of the program from 4.13.1.

4.14 Prefixes(used in 6.3)

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

The function `firsts` computes all prefixes of a list `xs`:

$$\begin{array}{lcl} \texttt{firsts}\ [\,] & = & [[\,]] \\ \texttt{firsts}\ [x_0 \ldots x_n] & = & [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2] \ldots [x_0 \ldots x_n]] \quad (\text{for } n \geq 0) \end{array}$$

1. Give the most general type for `firsts`.

2. If *xs* is a list of *n* elements, how many elements does the list (`firsts` *xs*) have?

3. Implement `firsts`. The result of this function should have the *same elements* as above. The *order of the elements* may be different.

4.15 Postfixes(*used in 6.3*)

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

The function `lasts` computes all postfixes of a list `xs`:

$$\begin{array}{lcl} \texttt{lasts}\ [\,] & = & [[\,]] \\ \texttt{lasts}\ [x_0 \ldots x_n] & = & [[x_0 \ldots x_n] \ldots [x_{n-2}, x_{n-1}, x_n], [x_{n-1}, x_n], [x_n], [\,]] \quad (\text{for } n \geq 0) \end{array}$$

1. Give the most general type for `lasts`.

2. If *xs* is a list of *n* elements, how many elements does the list (`lasts` *xs*) have?

3. Implement `lasts`. The result of this function should have the *same elements* as above. The *order of the elements* may be different.

4.16 Fragments(*used in 6.3*)

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

The function `frags` computes of a list *xs* all *fragments*. A fragment of *xs* is defined as a part of *xs* such that it only contains consecutive elements of *xs*. The number and the position of the first element is arbitrary (but of course limited by *xs*). Note that both `[]` and *xs* itself are fragments of *xs*. For example:

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

1. Give the most general type of `frags`.

2. If *xs* is a list of *n* elements, how many elements does the list (`frags` *xs*) have?

3. Implement `frags`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

4.17 Sublists(used in 6.3)

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

The function `subs` computes of a list *xs* all *sublists*. A sublist of *xs* is a list with the same elements in the same order, but in which an arbitrary number of elements is left out. Note that both `[]` and *xs* are sublists of *xs*. For example:

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

1. Give the most general type of `subs`.
2. If *xs* is a list of *n* elements, how many elements does the list (`subs` *xs*) have?
3. Implement `subs`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

4.18 Permutations(used in 4.20,6.3)

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

The function `perms` computes all *permutations* of a list *xs*. A permutation of *xs* is a list that contains the same elements, but possibly in a different order. Note that *xs* is a permutation of *xs*. For example:

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

1. Give the most general type of `perms`.
2. If *xs* is a list of *n* elements, how many elements does the list (`perms` *xs*) have?
3. Implement `perms`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

4.19 Partitions

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

The function `partitions` computes all *partitions* of a list *xs*. A partition of *xs* is a list $[A_1, \ldots, A_n]$ $(n \geq 0)$, such that A_1 `++` $\ldots$ `++` $A_n = xs$ and every A_i is non-empty. The empty list only has itself as possible partition. For example:

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

1. Give the most general type of `partitions`.
2. If xs is a list of n elements, how many elements does (`partitions` xs) have?
3. Implement `partitions`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

4.20 Lists and sorting

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

A list $L = [l_0 \ldots l_n]$ is sorted if for every $0 \leq i < j \leq n : \ l_i \leq l_j$.

Testing for sorted Use the idea from exercise 4.13.3 to construct a function `is_sorted` that takes a list and tests if the list is sorted. The function `is_sorted` may only inspect the list using list comprehension.

Sorting lists The list L' is the sorted list of L if L' is a *permutation* of L *and* L' is sorted. Write the function `zfsort` that takes a list and sorts it according to this definition.

Use the function `perms` from exercise 4.18, your function `is_sorted` and only a list comprehension.

Complexity What is the order of complexity of this way of sorting in terms of the length of the list? Is it a good way to sort lists?

4.21 Element frequency *(used in 5.9)*

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

Write the function `frequencylist :: [a] -> [(a, Int)] | == a` that takes a list and computes its *frequency-table*. A frequency table is a list of *tuples*. In each tuple, the first element is from the original list and the second element is the number of occurrences of that element in the original list.

Example:

```
frequencylist ['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)]
```

With the `showFrequencylist` function from `FrequencylistGUI` we can visualise the result.

Example:

```
import FrequencylistGUI

Start world = showFrequencylist (sort (frequencylist text)) world
where
   text     = ['Hello world! Here I am!']
```

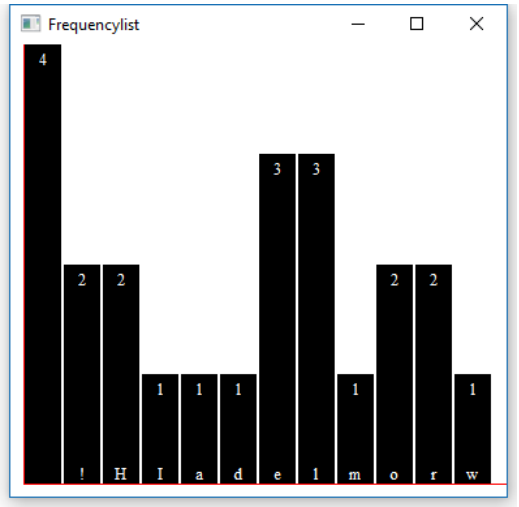


Figure 4.1: The frequency list of 'Hello world! Here I am!'

4.22 The last will be the first

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

The function `last` that returns the last element of a list can be written in several ways. A first method is by writing a recursive function:

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

Another way is by noting that the last element of the list is the same as the first element of the reversed list (you may find the function `reverse` in module `StdList`):

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

Compute Rewrite the following expressions by writing each step on a new line, and by underlining the redexes that you rewrite.

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

Function equality versus execution behaviour The functions `last1` and `last2` are *equal*, that is: given the same arguments they give the same result. That does not mean they are the same. Explain the difference using the results from the previous question.

4.23 Number sequence

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

Study the function `intChars :: (Int -> [Char])` from the lecture notes, paragraph 3.2.4 (*"Displaying a number as a list of characters"*). In the same style, write the inverse function, `charsInt :: ([Char] -> Int)`, that takes a `Char` list of all digits and computes the number represented by the list.
Example: `charsInt ['3210']` = 3210.

4.24 Overloading and []

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

In module `StdList` equality is defined for *lists* iff there is equality for the components. Implement instances for `zero`, `one`, `+`, `-`, `*`, / and ~ analogously. The implementations need to have the following properties:

```
∀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 this in a similar fashion as in exercise 2.12.

4.25 Uniform sets

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

Complete the implementation module that belongs to `StdSet.icl` for working with finite sets of elements of the same type. Start by choosing a representation for your set and completing the definition `:: Set a = ...`. The following properties need to be valid for operations on sets:

- The operation `toSet` converts a list of elements to a set. Sets do not contain duplicate elements, so the list that is obtained from `fromSet` may not contain duplicates.
- The predicates `isEmptySet`, `isDisjoint` and `memberOfSet` respectively test if a given set is empty (contains no elements), two sets are disjoint (have no common elements), and if a given value is a member of a given set (is contained by the set).

- The `Set` instance of `zero` gives the empty set. The empty set has no elements. The `Set` instance of `==` tests if two sets are identical (contain the same elements). Note that in sets, the order of the elements is not defined! This means that $\{1, 2\} = \{2, 1\}$. The function `numberOfElements` returns the number of elements of a set.
- There are two predicates that test a subset-relationship. The predicate `isStrictSubset` determines if the first argument is a *strict* subset of the second argument (i.e. the second argument contains elements that do not occur in the second argument, $A \subset B$). The predicate `isSubset` determines if the first argument is a subset of the second argument (all elements of the first argument are also an element of the second argument, $A \subseteq B$). Every set is a subset of itself, but it is not a *strict* subset of itself.
- The operations `union` and `intersection` respectively compute the union ($A \cup B$) and the intersection ($A \cap B$) of two sets. The operation `without` removes all elements of the second argument from the first argument ($A \setminus B$). The function `product` computes the Cartesian product of two sets ($A \times B$).
- The function `powerSet` computes the power set (set of all subsets, $\wp(A)$) of a given set A.

Use *ZF*-expressions, `..`-expressions or recursive functions as you see fit. Try to determine with which core functions you can construct the other functions in this module (for instance: two sets are equal if they are each other's subset).

4.26 Stack *(used in 4.27,5.16)*

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

Construct the implementation module that belongs to `StdStack.dcl` for working with stacks. The usual properties need to hold for the operations:

- `newStack` constructs an empty stack.
- (`push` x s) places x on top of stack s, and (`pushes` $[x_0, \ldots, x_n]$ s) places x_0, x_1, ..., x_n consecutively on top of the stack. Afterwards this means that x_n is at the top of the stack.
- (`pop` s) removes the top element of the stack s if possible. (`popn` n s) removes the top n elements of stack s if possible.
- (`top` s) gives the top element of stack s if possible, and a runtime error if s is empty. (`topn` n s) gives the top n elements of the stack s if possible, and a runtime error if there are not enough elements.
- (`elements` s) returns all elements of stack s, from the top to the bottom. (`count` s) returns the number of elements on the stack s.

Try to develop a small core of functions you can use to construct the other functions in this module (for example: define `pushes` using `push`).

4.27 Stack, part II

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

In exercise 4.26 you have implemented the abstract data type (`Stack a`). In this exercise we will again create an abstract data type (`Stack2 a`) using existential quantors (`E.`). The signature for this new type stack, `Stack2`, is:

```
definition module StdStack2

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

A (`Stack2 elem`) is a stack in which elements of type `elem` are stored. The type of the implementation (`impl`) is hidden by the existential quantor (`E.`).

`Stack2` instances

Extend the module `TestStdStack2` with two functions that create new stack objects:

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

The implementation of `listStack` has to work with lists of type `[a]`, and the implementation of `charStack` has to work with `String`. So complete the following implementations:

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

`Stack2` access functions, part I

The existential quantifier functions as a black box. Enter the following expression into module `StdStack2.icl` and recompile your module.

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

This will lead to a type error. Explain why.

`Stack2` access functions, part II

Pattern matching is the only allowed way to 'break in' on the encapsulation of an existential quantifier. To *push* an element `new :: elem` on a stack `stack2 :: Stack2 elem`, use an expression of the following form:

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

or:

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

This is quite a hassle. Add the following access functions to `StdStack2.icl`, and implement and export them.

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

Compare `StdStack` and `StdStack2`. Are they the same? Explain your opinion.

4.28 Sorted List

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

Complete the implementation module that belongs to `StdSortList.dcl`, which offers operations for working with lists of which the elements are sorted. The following properties need to hold for the operations (pay attention to the complexity restrictions below!):

- (`minimum` l) and (`maximum` l) return the minimum and maximum value of l, and generate a runtime error if l is empty. The order of runtime complexity of both functions needs to be $\mathcal{O}(1)$ (constant time).
- `newSortList` constructs a new empty, sorted list.
- (`memberSort` x l) tests if value x occurs in l. The predicate (`memberSort` x `newSortList`) is thus false for all x.
- (`insertSort` x l) inserts value x in l in the right position. The predicate (`memberSort` x (`insertSort` x l)) is thus always true for all x and l.
- (`removeFirst` x l) removes the first occurrence of x from l (there may exist duplicate elements in l). (`removeAll` x l) removes all occurrences of x from l. This means the predicate (`memberSort` x (`removeAll` x l)) is always false for every x and l.
- (`elements` l) returns all elements of l. This list is sorted and may contain duplicate elements. (`count` l) returns the number of elements of l. So: `length (elements` l`) = count` l.
- (`mergeSortList` l_1 l_2) merges l_1 and l_2 into a new sorted list. The following holds: `count` l_1 + `count` l_2 = `count (mergeSortList` l_1 l_2`)`; if (`memberSort` x l_1), then also (`memberSort` x `(mergeSortList` l_1 l_2`)`) and (`memberSort` x `(mergeSortList` l_2 l_1`)`).

4.29 Association list

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

Complete the implementation module that belongs to `StdAssocList.dcl`. In an associative list of type `AssocList k a`, elements of type `a` are stored using a *key* of type `k`. For each *key*, one element is stored. The following properties need to hold:

- `newAssocList` creates an empty association list.
- (`countValues` l) returns the number of stored elements.
- (`lookupKey` k l) gives the singleton list with value v if *key-value pair* (k, v) was present in l, and an empty list otherwise.
- (`updateKey` k v l) inserts the *key-value pair* (k, v) in l if k was not already present in l, and replaces the previous value associated with key k by v otherwise.
- (`removeKey` k l) removes the *key-value pair* associated with k if present, and leaves l unmodified otherwise.

4.30 Cards, part II

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

In exercise 3.9 you created the type `Card`. In this exercise you use them to develop some functions that generate lists of cards.

1. **Card deck** Write the function `carddeck :: [Card]` that generates a complete list of all cards in a deck. Try to write as short a definition as possible.
2. **Sorting by value** Write function `sort_by_value` that sorts a deck in ascending order. First sort by value, then by suit in the order *hearts* (♡), *diamonds* (♢), *spades* (♠) and *clubs* (♣). Try to write as short a definition as possible.
3. **Sorting by suit** Write the function `sort_by_suit` that sorts a deck of cards in increasing order. First sort by suit in the same order as above, then by value. Try to write as short a definition as possible.

4.31 Roman numerals *(used in 5.13)*

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

The classical Romans wrote their numbers using the following symbols: M, D, C, L, X, V en I with the respective values of 1000, 500, 100, 50, 10, 5 en 1. A positive integer is noted by writing these symbols consecutively, where the highest value symbols are on the left, and the lowest value symbols are on the right. The value of the number is the sum of the values of the individual symbols. Negative values and zero can't be represented.

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

For often occurring patterns, *abbreviations* are used:

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

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

Let the algebraic type `RD` represent Roman 'digits', and the type `Roman` a number notated by Roman numerals.

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

Roman → Int Write an instance for the class `toInt` for `Roman` that satisfies the properties given above.

Int → Roman Write an instance for the class `fromInt` for `Roman` that converts a *positive* `Int` value to the *shortest* Roman representation.

4.32 Boids

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

Boids were invented in 1986 by Craig Reynolds as an example of *artificial life* to describe the movement behaviour of animals in a group, like a flight of birds or a school of fish. For more background information, see http://www.red3d.com/cwr/boids/.

A simulation consists of a set of individuals named *boids*. Each boid has a *position* and a *speed*. You can run the simulation both in a space and on a surface. In this exercise, we use a surface. This means a position is a point (rx, ry), where $0 \leq rx \leq 1$ and $0 \leq ry \leq 1$. A speed is a vector (vx, vy). We assume that we have functions *pos* and *vel* that return the position and speed of a boid. Every boid respects three rules that each determine a share of its next speed, and thus position. A different explanation of these rules can be found at http://www.vergenet.net/~conrad/boids/pseudocode.html. The rules are:

1. *Boids move to the centre of nearby boids.* Let us compute the new partial speed v_1 for boid B. Let $B_1, \ldots, B_{n_1}$ be the boids that are within the short distance d_1 of B (experiment with values of d_1). Then the new partial speed v_1 for B is:

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

2. *Boids keep (a small) distance from objects, including other boids.* Let us compute the new partial speed v_2 for boid B. Let $B_1, \ldots, B_{n_2}$ be the boids that are within short distance d_2 of B (experiment with your own values for d_2). The new partial speed v_2 for B is:

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

3. *Boids adjust their speed to the speed of other boids nearby.* Let us compute the new partial speed v_3 for B. Let $B_1, \ldots, B_{n_3}$ be the boids within short distance d_3 of B (experiment with your own values of d_3). The new partial speed v_3 of B is:

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

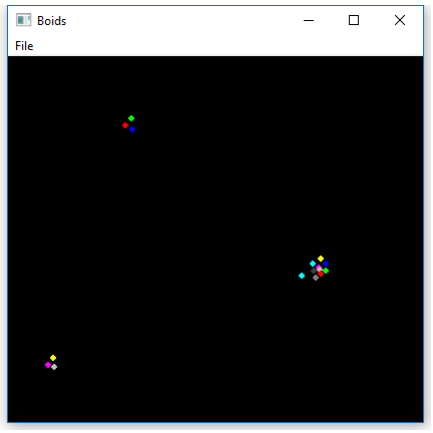


Figure 4.2: The Boids simulation program

The final new speed of boid B is $v = vel(B) + \sum_{i=1} 3v_i$, and the new position is $pos(B) + v$. By applying this computation to each boid B in a set of boids, it is possible to keep recomputing the new speeds and positions.

Write a function `simulate` that takes a list of boids, and returns a new list where the speeds and positions are updated as described above. In the given part of this exercise a window is opened that allows you to place new boids in the window using the mouse. These boids have an initial speed of $(0, 0)$. The framework draws boids in a simple fashion as coloured circles (see figure 4.2). In your implementation, make as much use of records and/or algebraic data types (to represent boids, speeds and positions), overloading (adding and subtracting speeds and distances) and list comprehensions as possible.

4.32.1 Optional: Boids 3D

Complete the same exercise, but have the boids move in a space. A position is then a point (rx, ry, rz) $(0 \leq rz \leq 1)$ and a speed is a vector (vx, vy, vz). Adapt the drawing of boids such that higher boids are drawn bigger and later than lower boids. You may also consider to change the colours by the height of the boid: lower boids get a colour shift towards black.

4.33 Aligning text

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

In this exercise you extend the main module `Aligning.icl`. This module opens a *window* in which text is shown (see figure 4.3). The idea is that the text is *aligned*. This can be done in four different modes, that can be represented by the type `AlignMode`:

```
:: AlignMode = TextAlignLeft | TextAlignCentred | TextAlignRight | TextAlignJustified
```

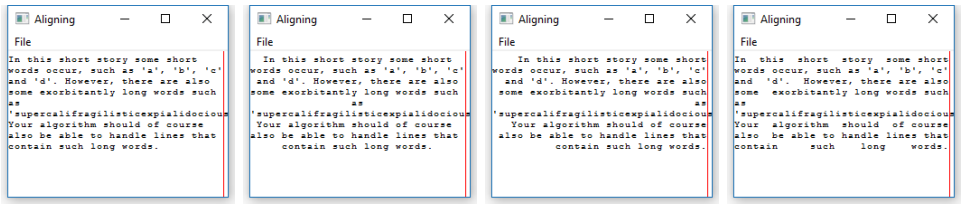


Figure 4.3: The four modes of alignment in action: `Left`, `Centred`, `Right` and `Justified`.

The program offers a *menu* where you can choose the active align mode so that you may conveniently test your program. In addition the program allows you to choose the font size.

The program is almost finished: the implementation of the alignment function is still missing. This is the function you need to complete. The type is:

```
:: Text        :== String
:: Line        :== String
:: TextWidth   :== Int
:: LetterWidth :== Int

align :: AlignMode LetterWidth TextWidth Text -> [Line]
```

The first argument is the requested alignment mode. The second argument gives the width of each letter (we are using a so-called *non-proportional* font, i.e. all symbols have the same width). The third argument is the width of the *window* in which the text is drawn. The fourth argument is the text to be aligned.

Note that this program uses the GUI libraries of Clean: Object I/O. Make sure you have set the environment to *Object IO* in the Clean IDE after you have made the project.

4.34 Composing text *(used in 7.2,7.7,7.9)*

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

Complete the implementation module belonging to `TextCompose.dcl`. This module offers an abstract type `Text` and the corresponding operations, that allow you to easily compose ASCII text blocks. A text block consists of a number of columns (the 'width') and rows (the 'height'). These `Text` values can be placed next to each other (while you are controlling the vertical *alignment*) and below one another (while you are controlling the horizontal *alignment*). This allows to relatively easily display the structure of recursive data structures. This can be very convenient to test your functions.

The pre-given type definitions are:

```
:: Text                                    // The abstract type of a text block
:: 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
```

Implement the following operations:

- The `Text` instance of the overloaded `zero` function generates a text block that is 0 characters wide and 0 rows high.
- The operation `toText (ah, reqw) (av, reqh) a` transforms the value `a` first to an 'ordinary' `String` using the default class `toString`. Then a text block is created that has *at least* `reqw` columns and *at least* `reqh` rows. If the text needs more columns or rows, these are increased as required. The value `ah` decides the horizontal alignment, the value `av` the vertical alignment.
- The `Text` instance of the default `toString` overloaded function transforms a text block to a `String` in which the line breaks are implemented using *newline* characters.
- The overloaded function `sizeOf` returns the number of columns and rows of its argument. The `String` instance treats its argument as a text block of one row and the `Text` instance returns the instance's dimensions.
- The operation `(horz av ts)` places all text blocks in `ts` *next to each other*, and makes sure their vertical alignment corresponds with `av`. The width of the resulting text block is the sum of the widths of the elements in `ts`. The height is the maximum height of the elements in `ts`. The operation `(vert hv ts)` analogously places all text blocks *vertically*, using the horizontal alignment value `hv`. The width is then the maximum width of the elements in `ts` and the height the sum of the heights of the elements.
- The operation `(repeath n c)` creates a text block of one row that consists of `n c` characters. The operation `(repeatv n c)` creates a text block of one column that consists of `n c` characters.
- The operation `(frame t)` creates an ASCII-art text frame around the block `t`.

Example:
`toString (frame (toText (CenterH,10) (BottomV,4) 42))`
produces:

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

Example:

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

produces:

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

4.35 Word Sleuth

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

'Word sleuth' or 'word search' puzzles consist of letters in an $m \times n$ matrix. A word list is supplied with words that need to be found and crossed out in the grid. Punctuation is ignored and each word occurs exactly once. A word can occur horizontally (both from left to right and from right to left), vertically (from top to bottom or reversed) and diagonally (both from left to right and right to left, both from the top to bottom and reversed). After crossing out the words it is often possible to construct a sentence of the remaining letters. The example below reveals the name of a familiar programming language after crossing out the words:

```
E  C  L  L        LAZY
P  E  I  A        LIST
Y  S  F  Z        TYPE
T  A  N  Y        ZF
```

Implement an algorithm that given an $m \times n$ matrix of letters and a word list, crosses out all words as above. The algorithm needs to return the remaining letters from left to right, top to bottom.

4.36 Boggle words (*used in 7.12*)

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

Boggle (http://en.wikipedia.org/wiki/Boggle) is a word game that consists of sixteen dice that have letters instead of pips. The dice are rolled and then placed in a 4×4 matrix. Players try to extract as many words as possible. Words are formed by starting on an arbitrary letter and then moving to a neighbouring dice that is above, below, left, right or diagonally in any direction. Words need to have at least three letters and each die may only be used once in a word. This means the longest possible word is sixteen letters. Words may occur in both singular and plural form, verbs may not be conjugated. Perfect past tence is allowed. Names, foreign words, abbreviations and composed words are not allowed.

Implement the algorithm `boggle_words` that, given a 4×4 matrix of letters and a word list generates all valid words of at least 3 letters from the given matrix.

4.37 Change

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

In this exercise you write the program for a change machine. The change machine has an infinite amount of each kind of coin and bank note. It may however not dispense more than a specified amount of k coins / bank notes at the same time. Write the function

change that, given the value to be exchanged, the available currency, and the maximum amount of coins / bank notes to be dispensed, computes all possible solutions.

Use the following type synonyms and the type of change:

```
:: Amount     :== Int          // a positive number
:: Currency   :== Int          // a positive number
:: Currencies :== [Currency]   // a non-empty list
:: Coin       :== Int          // a positive number
:: K          :== Int          // a positive number
:: Change     :== [Coin]

change :: Amount Currencies K -> [Change]
```

Example: (The solutions may be returned in a different order)

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

4.38 Dominoes

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

The classical game of dominoes (the so-called *double-six* variant) consists of 28 rectangular tiles, called dominoes. Each domino has two halves that each have a number of *pips*. The 28 dominoes contain all unique combinations of pips, with the number of pips between *zero* and *six* (see also figure 4.4).

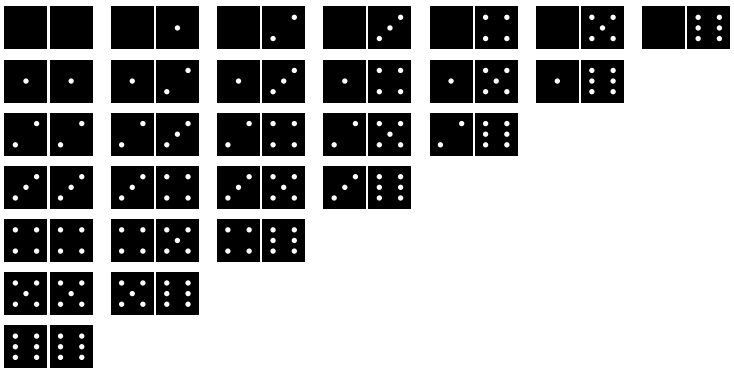


Figure 4.4: The dubble-six set of dominoes.

There exist variants with other combinations, like double-*nine*, double-*twelve* and even double-*eightteen*. We call such a collection of dominoes a double-N game (with N the maximum possible number of pips on one half of a tile). A double-N game has $MAX = \frac{(N+2)(N+1)}{2}$ dominoes. With the tiles of a double N game it is possible to make

a snake, i.e. a sequence of consecutive dominoes such that the short sides are connected, and the pips on connecting halves have the same number of pips (so 0 with 0, 1 with 1, etc.). A snake does not contain 'loops', which can be played during a real game of dominoes.

Write the function `all_snakes` that prints all possible snakes with length MAX of a double-N game.
Example: for a double-1 game all possibilities of length 3 are:

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

Example: for a double-2 game all possibilities of length 6 are:

```
[(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)]
```

4.39 Bag packer (used in 4.40)

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

In many so-called *role playing games* the player controls an avatar that through adventures (*quests*) can collect objects. A typical assortment consists of gems (light, small, worth a lot), potions (small, somewhat heavier, not very expensive), weaponry (swords, axes, bows, etc.; big, heavy, and varying in price from cheap to very expensive), armour (helmets, chainmail, gloves, boots, shoulder pads, etc.; all varying in size, weight and value), books (small, somewhat heavier, sometimes expensive) and what you find in these kinds of worlds (enchanted equipment such as rings, weapons, armour). Some objects are *stackable*. These objects have certain dimensions and weight, but multiple objects can take up the same space at the same time. This does lead to an increase of total and weight, but not in the taken up space. Typical *stackable* objects are gold, gems, potions, rings, etc.

The whole of items that a player collects is called their *inventory*. This inventory always has a certain *limited* capacity. This capacity is sometimes only decided by numbers, weight or dimensions, and often by a combination of these factors.

In the game *Skyrim* objects have a weight and a value. All objects are *stackable*. The inventory is only limited by the common weight: it may not be heavier than a predetermined value.

The lead character wants to fill his inventory as optimally as possible with the items he obtained in-game (whether legitimately or illegitimately). The player is not attached to any object and plans to sell them to replenish his bag of money. This means the

player wants to fill his inventory in such a way that it has the maximum possible value; there should be no other combination of items of which the sum of values is greater (although it may be the same). For our protagonist, write a function that takes the following arguments:

1. A capacity of the maximal weight.
2. A list of objects that each have a value and weight.

The function returns a collection of items that has the maximum summed value. Design appropriate data structures for this assignment.

4.40 Bag packer, part II

Main module:	*BagPacker2.icl*
Environment:	*StdEnv*

In exercise 4.39 you have designed a function that filles a bag pack with items to get the maximum possible value. The criterion to stop stuffing was the maximum weight that the bag pack could carry. In this exercise we introduce the *dimensions* criterium, via the computer game *Neverwinter Nights*. In this game objects have not just a weight and a value, but also *dimensions* (width and height). The inventory is carried in a certain backpack, which also has a certain width and height. In the game the inventory is limited by the dimensions: objects may not overlap. Some items in *Neverwinter Nights* are *stackable*, some are not. Typical examples of *stackable* objects are potions, arrows, rings, etc. Typical non-*stackable* objects are swords, armour, helmets, and so on.

A special element of *Neverwinter Nights* is the so-called *bag of holding*: this is a bag pack you can place in your inventory, and in which you can then place further items. Every *bag of holding* has its own dimensions, value and capacity (which is usually much bigger than its dimensions — e.g. a *bag of holding* with dimensions 2×2 units may have an internal capacity of 30×25 units). Obviously, these items are treasured by adventurers because these significantly increase the capacity of a player's inventory.

Again, write a function for the adventurer to get an optimal contents for their bag pack, such that the sale of the items gains the maximal amount of profit. Write a function that takes the following arguments:

1. A bag pack with a certain width and height;
2. A list of objects that each have a width and height, a value, and a weight. Each object belongs to a kind, and the kind may or may not be *stackable*.

The function needs to return the contents of the bag pack for which the sum of the values of all objects is maximal, i.e. there is no other combination of items that fit in the bag pack for which the sum of the values is *greater* (although it may be equal).

4.41 Farmer wants a wife

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

In the "stable marriage" problem, a set of N suitors needs to be coupled with N partners such that these result in 'stable' relationships. Each person has supplied an ordered

list of preferences for all candidates on the other side, through a numbering of 1 to N, where 1 indicates the most preferred candidate and N the least preferred candidate. A coupling of all suitors and partners is stable when there is no suitor and partner that would rather be with each other than with their current 'better half'.

If N is an even number, then there is an algorithm that computes such a stable coupling. This algorithm was proven correct in 1962 by David Gale and Lloyd Shapely[1]. It is an iterative algorithm that keeps computing the following: find a not yet coupled suitor and find the most favoured partner to whom the suitor has not proposed yet. The suitor is rejected if the proposed partner is already coupled with a suitor that is higher on the proposed partner's list of favourites. In every other case they accept. This means that if the proposed partner was already matched up with a suitor (which should then be lower on her list of favourites), then that suitor is no longer matched. This repeats until all suitors are coupled.

Implement this algorithm. Do this by implementing the following function:

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

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

The expression (`farmer_wants_a_wife (preferences_suitors, preferences_partners)`) computes a 'stable marriage' solution between the population of suitors and partners using the Gale / Shapely algorithm, if the input corresponds to the following properties:

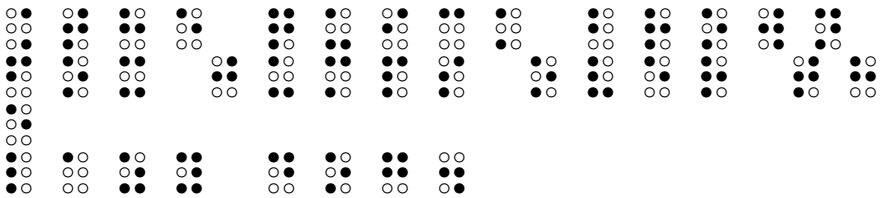
1. The length `N` of `preferences_suitors` is identical to the length of `preferences_partners` and is even.
2. The preferences of each suitor and partner are a permutation of `[1 .. N]`.

The solution is a list of couples (suitor, partner) that is stable.

4.42 Braille

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

In figure 4.5 the characters from the Dutch *Braille* writing system are shown (source MUZIEUM Nijmegen www.muzieum.nl). It was invented by Louis Braille (1809–1852) to allow those who are blind or visually impaired to read and write texts. Braille is a tactile script that's embossed on paper by creating raised dots that can be felt by touch. The characters are small rectangular blocks called *cells* that consist of two columns of three dots. The black dots (•) form the 'dots' in the paper and the empty dots (○) are not embossed into the paper. The digits 1 to 9 and 0 are the same as the letters a through j. To prevent ambiguity, numbers are prefixed by the number sequence symbol. The sentence *"The quick brown fox jumps over the lazy dog."* looks as follows in Braille:



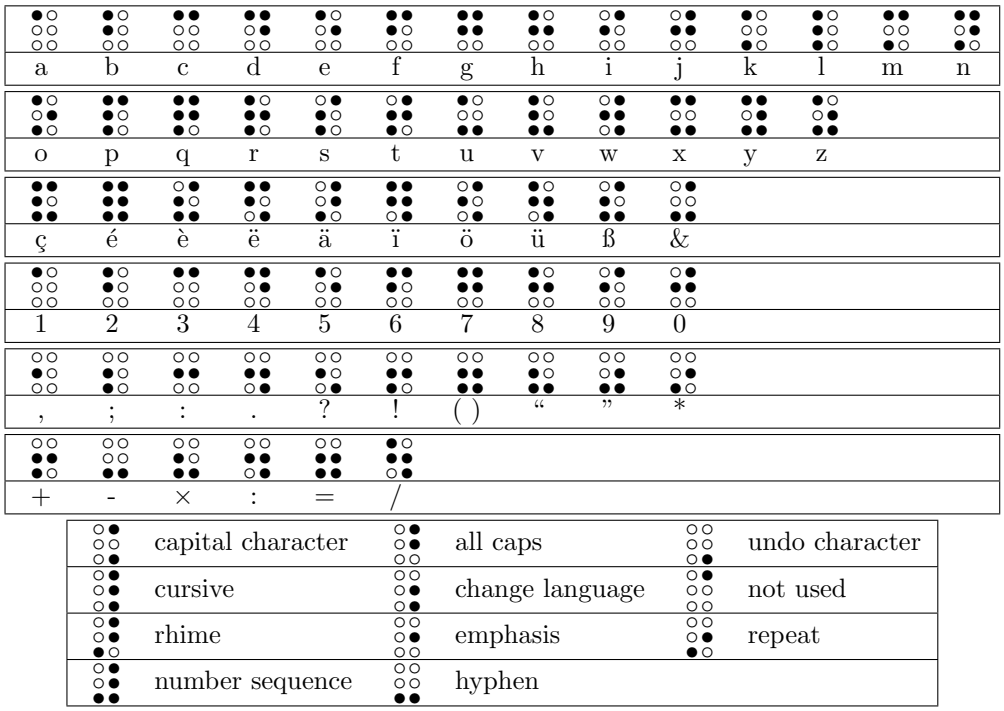


Figure 4.5: The Dutch Braille writing system.

Representation: write the function `showBrailleCharacter :: Char -> [(String, String, String)]` that converts a character from the alphabet in figure 4.5 to a Braille character. An empty dot is denoted as `'.'`, and an embossed dot is denoted as `'*'`.

Example: `showBrailleCharacter 'T' = [(".*","..",".*"),(".*","**","*.")]`. This consists of two elements: the capital character symbol, followed by the representation of t.

Example: `showBrailleCharacter 'a' = [("*.","..","..")]`

Example: `showBrailleCharacter '0' = [(".*",".*","**"),(".*","**","..")]`. This consists of two elements: first the number sequence symbol, followed by the 0 representation.

Example: `showBrailleCharacter '␣' = [("␣␣","␣␣","␣␣")]` (three rows of two spaces). Spaces are represented as an empty letter.

Example: `showBrailleCharacter '@' = []`. This one is empty, because `@` is not a part of the alphabet.

Writing: write the function `showBrailleText :: [Char] -> [(String, String, String)]` that converts a sequence of letters from the alphabet in figure 4.5 to Braille characters.

Example:

```
showBrailleText ['The␣quick␣brown'] = [(".*.**.*.␣␣***..****.␣␣*.*.*..***"
```

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

```
                                        ,"..****.*␣␣**..*.....␣␣*.**.***.*"
                                        ,".**.....␣␣*.**....*.␣␣..*.*..**."
                                       )]
```

Reading: write the function `readBrailleText :: [(String, String, String)] -> [Char]` that converts Braille texts like the one above to the latin alphabet.

Example:

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

4.43 Modern English spelling

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

The following satirical proposal to improve the spelling of the English language is often attributed to American author Mark Twain (1835–1910):

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)

Write a program that converts an English text (e.g. `ImproveEnglishSpelling.txt` in the *Exercises folder*) by the rules described above. The text leaves some room for interpretation of the new spelling rules. Instead, implement the following approximation of Twain's proposal for improvement of the English Spelling. These rules must be applied *consecutively* (make sure to account for capital letters):

1. 'ch' remains unchanged; 'c' followed by 'e' or 'i' becomes 's'; 'c' becomes 'k' in every other case.

2. ‘x’ becomes ‘ks’
3. ‘wh’ becomes ‘w’
4. ‘y’ becomes ‘i’
5. ‘g’ followed by ‘e’, ‘i’, ‘y’ becomes ‘j’ except for *gear*, *get*, *give*, *gir*, *gift*.
6. All double consonants are reduced to single consonants.
7. ‘ph’ becomes ‘f’
8. ‘r’ is removed if it is at the end of a word and the next word starts with a consonant.
9. ‘ch’ is replaced by ‘c’; ‘sh’ is replaced by ‘y’; ‘th’ is replaced by ‘x’.

Chapter 5

Higher Order Functions

In earlier chapters we have already used functions that created new functions from existing functions (like `o` and `flip`). *Currying* allows you to apply a function to less arguments than its arity (e.g. `((+) 1)`). These are all examples of *higher order* functions. Higher order functions are essential tools for the functional programming method, because you can use it to define *customizable programming patterns*.

5.1 Notations

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

Below a number of functions are given. Deduce from each the most general type and explain what the function does.

```
f1 x y       = x y
f2 x y z     = x z (y z)
f3 x y       = x (x y)
f4 x y z     = [v \\ v <- [y .. z] | x v]
f5 x y (z,w) = (x z,y w)
f6           = f5
f7 "-"       = -
f7 "+"       = +
f7 "*"       = *
f7 "/"       = /
```

5.2 With or without curry

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

Examine the `curry` and `uncurry` functions in `StdTuple`.

1. What do these functions do?
2. Deduce the type of the following expressions: `curry fst` and `curry snd`. What are their semantics?

3. Deduce the type of the following expressions: `uncurry (+)`, `uncurry (-)`, `uncurry (*)` and `uncurry (/)`. What are their semantics?

5.3 Function composition

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

The function composition operator `o` can be used to create a new function from two existing functions. It can be defined as: `o f g x = f (g x)` (the definition in `StdFunc` is slightly different). Explain what the following compositions do:

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

5.4 Flipping arguments

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

The function `flip` can be used to flip arguments of a function. It can be defined as: `flip f a b = f b a` (the definition in `StdFunc` is slightly different). Compare the functions below with their "flipped" variant and explain the difference, if any:

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*

Examine the function `twice` in `StdFunc`. What does this function do? Argue the result of the following `Start`-rule.

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

If you want to check your answer in the Clean IDE, you have to set the *Stack Size* to *1M*.

5.6 Ellipse perimeter

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

The *perimeter* of an ellipse with radii r_1, r_2 ($r_1 \geq r_2 > 0$) can be approximated by the following series:

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

Write a program that calculates the perimeter of an ellipse with radii r_1 and r_2 ($r_1 \geq r_2 > 0$) up to a desired accuracy. Use `until` from `StdFunc`.

5.7 Grouping elements (*used in 5.8,5.17*)

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

Write a polymorphic function `group :: (a -> Bool) [a] -> [[a]]` that gets a predicate p and a list xs as its arguments. The function calculates an *almost-partition* of xs. An almost-partition is a partition (see 4.19) of which the first and last element may be empty. Let $[A_0, A_1, \ldots A_{2 \cdot n}, A_{2 \cdot n+1}]$ be the result of (`group` p xs). Then:

- For all elements in every A_i where i is even, the predicate p holds. A_0 is empty only if p does not hold for the first element of xs.
- For all elements in every A_i where i is odd, the predicate p does not hold. $A_{2 \cdot n+1}$ is empty only if p holds for the last element of xs.
- `flatten (group` p xs`)` $= xs$. The function `flatten` can be found in `StdList`. This means that `group` does not remove or add elements and leaves the order unchanged.

Examples:

```
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 ['p4ssw0rd'] = [[],['p'],['4'],['ssw'],['0'],['rd']]
```

5.8 Word list (*used in 5.9,6.3,6.8,6.9*)

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

Write a function `words` that receives a list of `Char` and selects all its words. A word is, for

the purpose of this exercise, defined as a consecutive sequence of alphanumeric characters. Use `group` from exercise 5.7 in your definition: call it using a suitable predicate (have a look at `StdChar`).

5.9 Word frequency *(used in 6.9)*

Main module:	*WordFrequency.icl*
Environment:	*StdEnv* or *Object IO*

In exercise 4.21, you wrote the function `frequencylist` that computes the frequency list of a list of elements. In exercise 5.8, you wrote the function `words` that selects all words from a text, represented as a list of `Char`. In Clean, texts are usually represented as `String` values (arrays (`{}`) of unboxed (`#`) characters, hence: `{#Char}`). You can use the overloaded function `toString` to create a `String` from a list of `Char`, and `fromString` for the inverse operation.

Combine these functions to create a new function `word_frequency`, that given a text, represented as `String`, computes the frequency list of its words. The words in the frequency list must be represented as `String`.

5.9.1 Optional: displaying word frequencies

In exercise 4.21, you have seen that frequency lists can be visualised using the function `showFrequencylist` in `FrequencylistGUI`. This visualisation is only suitable for relatively short lists and elements with a short text representation. In the same module, the function `showFrequencylist2` is defined. It displays the bars horizontally. Figure 5.1 shows the generated output.

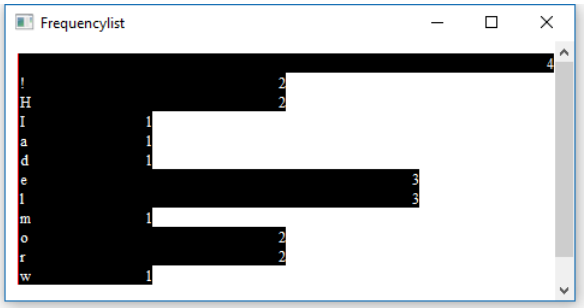


Figure 5.1: The frequency list of `['Hello␣world!␣Here␣I␣am!']`.

If you compare the two functions, you will realise that they are almost identical. Write a third, more general function, that can be used by parametrisation to realise the two existing functions in a much shorter way.

5.10 Origami

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

Rewrite the following functions from `StdEnv` using `foldl` or `foldr` and λ-abstractions: `sum`, `prod`, `flatten`, `length`, `reverse`, `takeWhile` and `maxList`. Rename your new functions `sum‘`, `prod‘`, `flatten‘`, `length‘`, `reverse‘`, `takeWhile‘` and `maxList‘`.

5.11 Any and all

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

Examine the functions `and`, `or`, `all` and `any` in `StdList`. Explain in your own words what they do. Write the function `and‘` which uses `all` and has the same meaning as `and`. Write the function `or‘` which uses `any` and has the same meaning as `or`.

`foldl` and `foldr`

The functions `and` and `or` can be expressed using `all` and `any`. Express the function `all` and `any` using both `foldl` and `foldr`. Call these functions `all_l`, `all_r`, `any_l` and `any_r`.

`foldl` or `foldr`?

Both the `&&` and the `||` operator are *conditional* tests: they inspect first the value of the first argument, and if the second argument is not needed to determine the result, it is not evaluated. This is reflected in their types:

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

The strictness annotation (`!`) for the first argument indicates that the function will always evaluate the argument, while the absence of that annotation for the second argument indicates that it is only evaluated when needed (lazy evaluation).

Predict what will happen when `all_l`, `all_r`, `any_l` and `any_r` are applied to an infinite list of booleans. Do this using the following examples:

```
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 `StdBool` improved

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

Lift

Deduce the most general type of the following functions and explain what they do:

```
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)
```

Boolean classes

In `StdBool`, the following boolean operations are defined:

```
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
```

These operations are, contrary to the arithmetic operations `+`, `-`, etc., not *overloaded*. It often happens that you don't only want to combine `Bool` values to a new `Bool`, but also predicates (`a -> Bool`) to a new predicate (`a -> Bool`). For example: to select all elements between 3 and 8 you have to write:

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

If `&&` would have been overloaded, you could have defined it on predicates, in order to write:

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

To realise this, the following definition module has been made. It expands `StdBool` with the desired overloaded operators:

```
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)
```

Write the corresponding implementation module. Use, when possible, the `lift`*i* functions.

5.13 Roman Numerals II

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

In exercise 4.31, you wrote an implementation of Roman numerals. Develop a new Clean module `StdRoman` in which Roman numerals and computations are defined. Create instances of the overloaded classes that are instantiated for `Int` in `StdInt`, with the following exceptions:

- Rename `toInt` and `fromInt` to `toRoman` and `fromRoman`, respectively, and define the corresponding class definitions.

- The `Int` operations `bit[x]or`, `bitand`, `bitnot`, `<<` and `>>` are not overloaded and cannot be used for `Roman`s.

Use higher order functions in your implementation whenever applicable. In fact, all your instances *should* be one-liners.

5.14 `scan` and `iterate`

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

Examine the functions `scan` and `iterate` in `StdList`.

1. Rewrite the `Start` rule:

   ```
   Start = scan (+) 0 [3 .. 5]
   ```

   first using the *applicative* evaluation strategy and then using the *normal order* evaluation strategy.

2. Use the above result to construct the rewriting process of the following `Start` rule:

   ```
   Start = scan (*) 1 [3 .. 5]
   ```

3. Argue what the result of the following two `Start` rules would be:

   ```
   Start = take 5  (iterate (flip (^) 2) 2)
   Start = take 10 (iterate (flip (/) 10) 123456)
   ```

5.15 Arithmetic sequences

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

Use only functions from `StdEnv`, λ-abstractions and list comprehensions in this exercise.

Plus-minus Write a function `plusminus` which, given a list of values $[x_0 \ldots x_n]$ $(n \geq 0)$, computes the value $x_0 - x_1 + x_2 - x_3 + \ldots$.

Taylor sequence for sine The *Taylor* sequence for the *sine* is defined as:

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

Implement the function `sine` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Taylor sequence for cosine The *Taylor* sequence for the *cosine* is defined as:

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

Implement the function `cosine` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Gregory-Leibniz sequence for π The *Gregory-Leibniz* sequence to approximate π is defined as:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots = \sum_{n=0}^{\infty} \frac{4}{(-1)^n \cdot (2n+1)}$$

Implement the function `pi1` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Nilakantha sequence for π The *Nilakantha* sequence to approximate π is defined as:

$$\pi - 3 = \frac{4}{2 \cdot 3 \cdot 4} - \frac{4}{4 \cdot 5 \cdot 6} + \frac{4}{6 \cdot 7 \cdot 8} - \frac{4}{8 \cdot 9 \cdot 10} + \ldots = \sum_{n=0}^{\infty} \frac{4}{(-1)^n \cdot \prod_{i=2(n+1)}^{2(n+2)} i}$$

Implement the function `pi2` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

5.16 `seq` and `seqList`

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

Examine the functions `seq` and `seqList` in `StdFunc`. Implement the following functions from `StdStack` (exercise 4.26) again:

- `pushes‘`, similar to `pushes`, using `seq` and `push`.
- `popn‘`, similar to `popn`, using `seq` and `pop`.
- `topn‘`, similar to `topn`, using `seqList` and `top` and `pop`.

5.17 Database queries *(used in 6.4)*

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

In this exercise, you write a number of database queries using list comprehensions and higher order functions. Use `FQL.icl` to read the database into the program. It imports the `StdTime` module, which you have written in exercise 3.18. The `Start` rule of the `FQL` module reads the file `FQL-songs.dbs`. This file holds the data of tracks from Peter's music CD collection. To store all this data, your application needs a bigger *heap* and *stack* than the default. Set in the project options *Maximum Heap Size* to *4M* and *Stack Size* to *1M*. After reading the file, all tracks are available as a member of a list of type `[Song]`:

```
::  Song = { group     :: String      // Name of the group
           , album     :: String      // Name of the album
           , year      :: Int         // Release date
           , track     :: Int         // Track nr. (1..)
           , title     :: String      // Title of the song
           , tags      :: [String]    // Descriptive tags of the album / song
```

```
        , time      :: Time        // Playing time of the song
        , countries :: [String]    // Countries of origin of the group
        }
```

The `FQL-songs.dbs` file has the following characteristics:

- The content is sorted (ascending) on the fields above (first `group`, then `album`, etc.). Fields are considered `String` and ordered using the `<` instance on `String` values.
- Every `album` of every `group` has a unique title. However, there are `album` titles that are used by different `group`s (like the album *Up*, which is used by both *Peter Gabriel* and *R.E.M.*).

Below, a number of queries is described. Write for every query a function that answers it. Use list comprehensions, higher order functions, and standard functions whenever applicable.

1. Write the function `all_groups :: [Song] -> [String]`, which enumerates all `group`s without duplicates and in ascending order according to the `<` instance on `String` values.
2. Write the function `all_albums_of :: String [Song] -> [(Int,String)]`, which enumerates all albums that were published by a given group. List both the `year` and the `album` title without duplicates, and sort in ascending order on the year of release.
3. Write the function `all_tracks :: String String [Song] -> [(Int,String,Time)]`. It should list all tracks of a given `album` and `group`. Show of each track the `track` number, the `title` and the `time`. The list has to be sorted in ascending order of the `track` number.
4. Write the function `time_albums :: [Song] -> [(Time,String,String)]`, which returns all albums of all groups. In every result (a, b, c), a is the total playing time, b the `group` and c the `album`. Sort the list in ascending order of the total playing time. Write the names of the shortest and the longest album down in comments.
5. Write the function `total_time :: [Song] -> Time`, which returns the total playing time of all songs. Write the total playing time in the $m^+ : ss$ notation of exercise 3.18 in comments.
6. Write the function `dutch_metal :: [Song] -> [String]`, which enumerates all `group`s without duplicates where one of the `tags` equals `"metal"`, and one of the countries matches `"Netherlands"`.
7. Write the function `all_periods :: [Song] -> [String]`, which enumerates all the `year`s in ascending order, using short notation. The short notation entails that for every period of more than one consecutive year only the first and last year are mentioned.

   Example: suppose that the albums were from the years 1967, 1968, 1969, 1972, 1973, 1975 and 1978. Then the result of this function is: `["1967-1969", "1972-1973", "1975", "1978"]`.

5.18 Pseudo random numbers(*used in 5.19,6.5,7.12,6.10,6.11*)

Main module:	*RandomNumbers.icl*
Environment:	*StdEnv* or *Object IO*

The `Random` module provides a function `random` to create pseudo random numbers:

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

This function gets as its argument a `RandomSeed` with which a pseudo random number of type `Int` and a new `RandomSeed` can be computed. With this new `RandomSeed`, the next pseudo random number can be computed, etc.

Since we are working in a pure functional language, `Random rs` will always return the same pseudo random number and new `RandomSeed`.

A more or less random `RandomSeed` can be retrieved using `getNewRandomSeed`:

```
getNewRandomSeed :: *World -> (RandomSeed, *World)
```

This function uses its `*World` argument to read the current time, and derives from that an initial `RandomSeed`. You can use the `*World`, that you can get in the `Start` rule, as the argument to `getNewRandomSeed`:

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

On Windows, the `Random` module uses the `StdTime` module from *Object IO*. Therefore, you have to use the `Object IO` environment. Linux users can edit `Random.icl` by following the comments and avoid the Object IO dependency.

n pseudo random numbers

Write the function `random_n :: Int RandomSeed -> ([Int], RandomSeed)`, which given a positive number n and an initial *RandomSeed*, returns a list of n consecutive pseudo random numbers. Use the function `seqList` in `StdFunc` and `repeatn` in `StdList`.

∞ pseudo random numbers

Write the function `random_inf :: RandomSeed -> [Int]`, which returns *all* consecutive pseudo random numbers. To this end, first write a general function `iterateSt :: (s -> (a,s)) s -> [a]` which looks like `iterate` in `StdList`. Use `iterateSt` in your definition of `random_inf`.

Shuffling

In many programs (simulations, games, ...) it is required to reorder a list of values randomly. Write the function `shuffle :: [a] RandomSeed -> [a]`, which given a finite list `xs` and a `RandomSeed` returns a random permutation of `xs` (see for an explanation of the term permutation exercise 4.18).

5.19 `return` and `bind`

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

Examine the functions `return` and `bind` in `StdFunc`.

(id,o) vs. (return,bind) Compare `id` with `return` and `o` with `bind`. What is the relationship between these functions? Rewrite `bind` using `o` and `uncurry`, that is, finish the following definition:

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

Applications Write the following functions using `return` and `bind1`:

- Use the `random` function as explained in exercise 5.18 and write the function `sum2 :: (RandomSeed -> (Int,RandomSeed))` (mind the parentheses!) which generates two random numbers using the `RandomSeed` and returns their sum together with a new `RandomSeed`.
- Give an other implementation of `seqList` from `StdFunc`. Finish the following definition (mind the type!):

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

Chapter 6

Console and File I/O

In this chapter, we discuss simple console and file I/O. By console I/O we mean text-based input from and output to the console or terminal window. Using file I/O we can deal with files in text and binary format. Both forms of I/O are based on the *uniqueness type system* of Clean, and use the same abstraction: `File` and `*File`.

6.1 Echo

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

Write a console I/O program that performs the following actions:

1. It opens the console file `stdio` from the `world`;
2. reads repeatedly a line of input from `stdio` and echoes each line through `stdio`;
3. and exits (cleanly: `stdio` should be closed in the `world`) if the input was empty.

6.2 Oh Tannenbaum II

Main module:	*OTannenbaum2.icl*
Environment:	*StdEnv*

Write a console I/O program that repeatedly reads a number as its input and prints the corresponding Christmas tree as described in exercise 2.10. If you have done exercise 2.10 or 4.34, you may of course use those functions. The program exits when a non-numeric value is inputted.

Note: the `toInt` function that converts a `String` to an `Int` fails if the string is not exactly an integer. In particular, this may happen when the console input ends with a newline. Use the `String` instance of the overloaded function `%` to remove the newline character from the input string.

6.3 Wordplays

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

In this exercise, you write a console I/O program that repeatedly reads a command (as described below) and prints the corresponding output. Use the function `words` from exercise 5.8. For each command, reuse the list function that is mentioned if you implemented it.

Prefixes The command `firsts`, followed by a sentence s, consisting of the words $w_0 \dots w_n$ (ended with a newline character) prints the sentences with the words w_0, w_0 and w_1, w_0 through w_2, ..., and w_0 through w_n. Each sentence is placed on a separate line. Use the function `firsts` from exercise 4.14.

Postfixes The command `lasts`, followed by a sentence s, consisting of the words $w_0 \dots w_n$ (ended with a newline character) prints the sentences with the words w_n, w_{n-1} and w_n, w_n through w_{n-2}, ..., and w_0 through w_n. Each sentence is placed on a separate line. Use the function `lasts` from exercise 4.15.

Fragments The command `frags`, followed by a sentence s, consisting of the words $w_0 \dots w_n$ (ended with a newline character) prints the sentences with all word fragments. Each sentence is placed on a separate line. Use the function `frags` from exercise 4.16.

Subwords The command `subs`, followed by a sentence s, consisting of the words $w_0 \dots w_n$ (ended with a newline character) prints the sentences with all subwords. Each sentence is placed on a separate line. Use the function `subs` from exercise 4.17.

Permutations The command `perms`, followed by a sentence s, consisting of the words $w_0 \dots w_n$ (ended with a newline character) prints the sentences with all word permutations. Each sentence is placed on a separate line. Use the function `perms` from exercise 4.18.

6.4 Interactive database queries

Main module:	*FQL2.icl*
Environment:	*StdEnv*

In exercise 5.17, you have written a number of functions to extract information from Peter's CD collection. In this exercise, you write an interactive variant of that program, using console I/O. If you have completed exercise 2.6, use the function `is_match` to match strings with *wildcards*. The queries a user can enter start with a *command*. If the command has parameters, these are separated by exactly one space. Implement the following queries:

1. *group␣text*: show all groups (every group exactly once) of which the `.group` name corresponds to *text*. End with printing the number of groups.

2. *album␣text*: show all albums (every album exactly once) of which the `.album` title corresponds to *text*. Show of every album the `.group`, `.album` and `.year`. End with printing the number of albums.

3. *year␣m␣n*: Show all albums a (every album exactly once) for which holds: $m \leq a.\texttt{year} \leq n$. Show of every album the `.group`, `.album` and `.year`. End with printing the number of albums.

4. *title␣text*: show all tracks of which the `.title` corresponds to *text*. Show of every track the `.group`, `.album`, `.title` and `.track` number. End with printing the number of tracks.

5. *albums␣text*: show the `.albums` of the `.groups` of which the group name corresponds to *text*. Show for every group the found albums (the `.album` and `.year`). End with printing the number of albums.

6. *tracks␣text*: show all tracks of every `.album` of which the title corresponds to *text*. Show for every album first the `.group`, `.album`, `.year` and total playing time, and then all tracks (`.track`, `.title` and `.time`) in ascending order of `.track`.

7. *tags*{␣*tag*}$^+$: show all tracks of which the `.tags` correspond to the *tag* parameters of the command (at least one of the parameters should be present in the `.tags` field). If for some album *all* tracks correspond to the request, only the information of the album has to be printed (`.group`, `.album`, `.year`). If not all tracks correspond to the request, the information of every track has to be printed separately: `.group`, `.album`, `.year`, `.track` and `.title`.

6.5 Mastermind

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

Mastermind is a code-breaking game for two players: the *codemaker* and the *codebreaker*.

- The *codemaker* thinks of a code that consists of *four* colours. He can choose from *eight* colours: *white*, *silver*, *green*, *red*, *orange*, *pink*, *yellow* and *blue*. There are no restrictions for the code: it may consist of four identical colours, but also four different colours. In total, there are $8^4 = 4096$ possible combinations.

- The *codebreaker* tries to guess the code in as few turns as possible. If he needs more than *twelve* guesses, he loses. A turn consists of giving a concrete colour code. The *codemaker* gives two answers:

  – The number of colours that is correct (the right colour on the right position).
  – The number of colours (apart from the number of colours that are correct) that occur in the code (but are not on the right position).

Write a console I/O program that implements the game *Mastermind*, where the user has the role of *codebreaker*. The program is the *codemaker*. For generating random codes you can use the pseudo random number generator from exercise 5.18.

Write the implementation in such a way that it abstracts from the constant values that are mentioned above: it has to work for any code length, for any number of colours and for any maximum number of tries.

6.6 A file I/O module(*used in* 6.8,6.9,6.10,6.11,7.13)

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

To work with files, the standard Clean module `StdFile` can be used. Frequently occuring

use-cases of working with files is to work with the file content entirely as if it were a single text (`String`), or a list of lines (`[String]`, or some content onto which you want to map a function of type (`a -> b`). In this assignment you create access functions for these purposes.

Base module

Develop the module `SimpleFileIO` with the following signature:

```
definition module SimpleFileIO

import StdFile, StdOverloaded, StdMaybe

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

The first argument of `readFile` and `writeFile` is a path to a file, in the same format as is needed for the `fopen` and `sfopen` functions in `StdFile`. The last argument of both functions is a suitable environment on which the file I/O operations are instantiated (like `*World`). The result value of type `env` is, naturally, the modified environment after the operation has been performed. The function `readFile` reads in the complete contents *txt* of the given file and returns (`Just` *txt*). The file needs to be opened, read and closed. If something goes wrong, `Nothing` can be returned. The function `writeFile` gets as its second argument the new contents of the given file. If opening the file, writing the new contents to it and closing the file succeeds, the `Bool` value should be `True`; if not, it should be `False`.

The module can be tested using the first `Start` rule in `TestSimpleFileIO.icl`. This rule copies the module itself to a file with the name `TestSimpleFileIO1.icl`.

Line by line

Add read and write functions to this module that consider the contents as a list of lines (ended with `'\n'` except for the last line):

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

Test these functions using the second `Start` rule in `TestSimpleFileIO.icl`. This rule stores the lines in `TestSimpleFileIO1.icl`, in reversed order, in `TestSimpleFileIO2.icl`.

An overloaded interface

Add an overloaded function `mapFile` to `SimpleFileIO`:

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

The first argument to `mapFile` is a path to a file of which the contents will be read. The second argument is a path to a file to which we will write. The third argument, a function of type `a -> b`, has to be applied to the contents of the first file, after conversion from string has been applied, and the result of which is written to the second file, after conversion to string has been applied.

You can test the new function with the third `Start` rule in `TestSimpleFileIO.icl`. It converts all characters from `TestSimpleFileIO2.icl` to upper-case and writes these to `TestSimpleFileIO3.icl`.

6.7 Monadic Console and File I/O

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

In the *Exercises folder*, you find the `StdMonad` module in which the *monadic operations* `return` and `>>=` are defined. The `fail` pattern is also defined there. In the same folder, you find the modules `StdMaybeMonad`, `StdListMonad` and `StdStateMonad`, which define instances of these operations for `Maybe`, `[]` and `St`.

In this exercise, you develop the `StdIOMonad` module which defines the type `IO` as an instance of monadic operations. The module makes monadic versions of a small number of console and file functions:

- Reading a line of input (including newline character) from the console (`read`), and writing a string to the console (`write`).
- Opening and closing a file (`open` and `close`). The file is identified by its name. We restrict ourselves to text files and therefore only support the file modi `Read` and `Write`. A file which is already open at the moment of evaluating the `open` function, cannot be opened. If the `open` function successfully opens a file, a *file handle* (`FileHandle`) to that file has to be returned. That `FileHandle` can be used for further operations on the file. Only a file that is open can successfully be closed by the `close` function. From that moment on, the *file handle* should be unusable. The file itself can be edited again by opening it again (resulting in a new file handle).
- Reading and writing of a file (`eof`, `readline` and `writeline`). The function `eof` checks if all text data from the file has been read. The function `readline` reads the next line of text, including newline character (if present), from the file. The function `writeline` writes the line of text to the file. All three functions only work on opened files.

Finally, the module defines the `doIO` function, which transforms a computation of the type `(IO a)` into a Clean function that manipulates the `*World`. If m is a monadic program, then `Start world = doIO` m `world` is the equivalent Clean program.

6.8 Word list II

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

Write a function `wc` (*word count*) of the following signature:

```
wc :: String *env -> (Int,*env) | FileSystem env
```

It gets as its first argument the name of a text file that needs to be opened, using the `FileSystem` environment `env`. If successful, it counts the number of words in that file. Use the simplistic function `words` from exercise 5.8 for counting the words. Use the `SimpleFileIO` module from exercise 6.6 for reading the file. If it fails, it returns zero, and the environment.

6.9 Word frequency II

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

Write a function `wf` (*word frequency*) of the following signature:

```
wf :: String *env -> ([(String,Int)],*env) | FileSystem env
```

It gets as its argument the name of a text file that needs to be openend, using the `FileSystem` environment `env`. If successful, it returns the frequency table of the words in that text file, as described in exercise 5.9. Use the `SimpleFileIO` module from exercise 6.6 for reading the file. If you have completed the optional exercise 5.9.1, you can use the `showFrequencylist2` function for the visualisation. If it fails, it returns the empty list, and the environment.

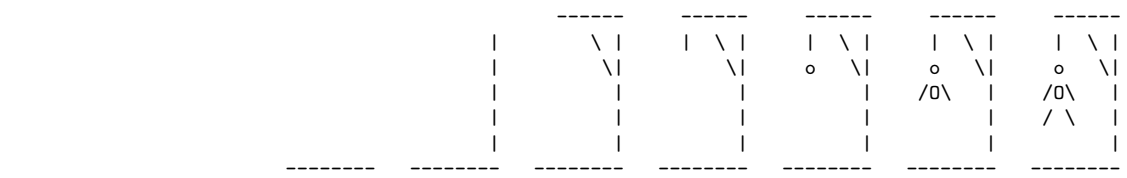
6.10 Hangman

Main module:	*Hangman.icl*
Environment:	*ObjectIO*

The game *Hangman* is a word guessing game for two players. One player thinks of a word consisting of at least two letters. The other player has to guess the word in as few turns as possible. The length of the word is known to the guesser. In every turn, the guesser may do either of the following:

- Say a letter. The challenger shows all positions of that letter in the word. If the letter does not occur in the word, the guesser gets a penalty point.
- Say a word. The challenger says if that is the word he had in mind. If this is correct, the guesser wins. If it is not correct, the guesser gets a penalty point.

After every turn in which the guesser has gotten a penalty point, a part of a gallows is drawn. This can be done using ASCII-*art*. For example:



The guesser loses when the gallows is drawn completely.

Write a console I/O program that implements *Hangman*. The user is the guesser; the program, the challenger.

You can use the dictionary in `EnglishWords.txt` in the *Exercises folder*. Use the module `SimpleFileIO` that you have written in exercise 6.6 to read it. You can use the pseudo random number generator from exercise 5.18 to randomly select a word.

6.11 One-time pad encryption

Main module:	*OTP.icl*
Environment:	*ObjectIO*

In this exercise, you develop a program that can encrypt and decrypt arbitrarily large

text files using the one-time pad method[1]. For this we need random streams, so you will have to work with random numbers (see exercise 5.18). But first some background information about the one-time pad method.

OTP is an encryption method which has, theoretically, perfect security and is thus unbreakable. The idea is that you can manipulate a message by XOR-ing it with a *truly* random number stream, such that it only contains ambiguous data. To decrypt the data, you need an exact copy of the random number stream used for encryption. In theory, this method is unbreakable. In real, there are several problems:

- Generating true randomness is not trivial. Most random number generators found in programming language are not truly random.
- Every random stream may be used only once, and has to be destroyed afterwards. Destroying data on a common computer is difficult.
- The transmitter and receiver must have an exact copy of the random stream. Therefore, the system is vulnerable to damage, theft and copying.

Despite these drawbacks, the OTP method has been used by for example the KGB (the main security agency of the former Soviet Union). They used truly random streams printed in a very small font, so that it could be hidden relatively easily (see figure 6.1).

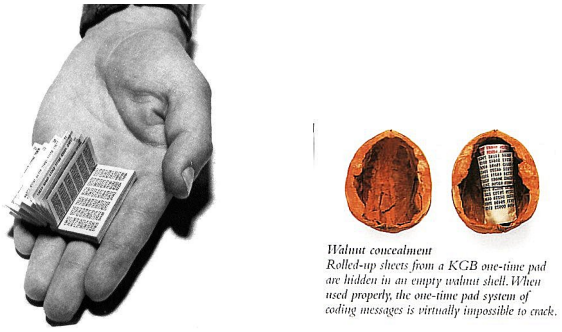


Figure 6.1: Truly random stream sources for OTP encryption, used in espionage. Source: `http://www.ranum.com/security/computer_security/papers/otp-faq/` (left image); `http://home.egge.net/~savory/chiffre9.htm` (right image).

In this exercise, you implement the OTP method for encrypting files. Since we don't have truly random streams, you work with the pseudo random number generator from exercise 5.18. Choose an integer $\mathcal{R}$ for your encryption method. Keep this value secret. Use $\mathcal{R}$ as the `RandomSeed` to generate a sequence of pseudo random numbers $r_0, r_1, \ldots$.

One-time pad encryption and decryption

Encryption and decryption works as follows. Let A be the input text file that should be encrypted / decrypted to the output text file B. Read the ASCII characters $a_0, a_1, \ldots, a_N$

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

from A, and write the characters $b_0, b_1, \ldots, b_N$ to B.

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

This case distinction ensures that non-printable ASCII characters ($0 \leq a_i < 32$) are unchanged, and that other ASCII characters ($32 \leq a_i < 128$) are encrypted. When *encrypting*, $\oplus$ is *addition* ($+$). When *decrypting*, $\oplus$ is *subtraction* ($-$). The addition of $(128 - 32)$ is superfluous when encrypting, but it guarantees that intermediate results are positive values in case of decrypting.

Write a console I/O program that accepts the input `otp` A B, where A and B are paths to text files. It encrypts the contents of A and writes the result to B. The input `pto` A B does the reverse: it decrypts A and writes the result to B.

Test your program on a representative input file A, that you encrypt and then decrypt. Verify that the decrypted file is identical to A.

You can use the module `SimpleFileIO` from exercise 6.6 to read and write the files.

Chapter 7

Recursive Types

Up to this point, disregarding lists, we have not yet made use of recursive data structures in the exercises. Recursive data structures can be created with algebraic types and records. Lists are a special case. Recursive data structures are also known as *tree structures*, because they have a starting point, known as the *root*, and a *branching structure*. The principles you employed to work with lists are also applicable to tree structures. This means that their size is unbounded, and that calculations can be stored for potential future computation.

7.1 Binary trees (*used in* 7.2, 7.3, 7.18)

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

Take the following type definition of a binary tree (lecture notes, section 3.6.1, p. 71):

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

Drawing Draw the trees `t0` ... `t7` that are created as follows:

```
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)
```

Tree metrics Implement the functions `nodes`, `leaves` and `depth`: `nodes` counts the number of `Node`s, `leaves` counts the number of `Leaf`s, and `depth` is the *maximum* number of `Node`s that you can encounter by traversing the tree from its *root* to one of its *leaves*. As such:

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

Number of nodes versus number of leaves versus depth The metrics of binary trees are related to each other. Show that the following metrics hold for all finite trees t by using structural induction on t.

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

7.2 Displaying trees *(used in 7.3)*

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

In exercise 7.1 you manually drew a number of trees. To be able to check whether you have indeed drawn the trees correctly, it is convenient to have a function for that purpose. In this exercise you develop two algorithms that fit that purpose:

1. a function `indentTree` printing the tree by systematically *indenting* depending on the depth of a node and its subtrees;
2. a function `tree2D` utilizing the module `TextCompose` from exercise 4.34 to produce a more traditional representation of the tree structure.

The `BinTreePrint` module exports this functionality as an instance of the standard `toString` class of Clean:

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

Choose the instance that you want to implement, test or use by commenting away the other instance. In this exercise you can use the functionality to assess whether you have drawn the trees in exercise 7.1 correctly.

7.2.1 Printing with indentation

The function `indentTree` receives a binary tree structure, such as `t7` from exercise 7.1, and prints it as follows:

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

This output shows you that the distances of nodes to the left column depend linearly on their depths in the tree. The *root* node is placed on the left column, its child nodes are indented one level, the children's child nodes are indented two levels, etc. The parenthesis structure further indicates which elements belong together.

Implement the function `indentTree` that prints a binary tree structure as described above. Test your function with trees `t0` up to and including `t7` from exercise 7.1.

7.2.2 Printing in 2D

The output generated by `indentTree` is not always clearly readable, because nodes at the same depth can be displayed far from each other. Sometimes it is clearer to display these nodes *side by side*. The function `tree2D` receives a binary tree structure, such as `t7` from exercise 7.1, and prints it as follows:

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

This output shows you that nodes of the same depth are displayed side by side. The two subtrees of a node are displayed side by side as two *text blocks*. This suggests that the `TextCompose` module developed in exercise 4.34 can be applied here.
A challenging aspect of displaying tree structures in this way is that the size of the left subtree is not in general equal to that of the right subtree. This has consequences for determining the position of the corresponding node.

Implement your function `tree2D` that prints a binary tree structure as described above. Test your function with trees `t0` up to and including `t7` from exercise 7.1.

7.3 Binary search trees

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

Take the type definition of a binary tree (lecture notes, section 3.6.2, p. 73).

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

insert Study the function `insertTree` (lecture notes, section 3.6.2, p. 73). Draw trees $z_0 \ldots z_7$:

```
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
```

If you performed exercise 7.2, check your answers using your implementation of the `toString` instance for binary trees.

delete Study the function `deleteTree` (lecture notes, section 3.6.4, p. 75). Draw tree z_8:

```
z8 = deleteTree 50 z7
```

If you performed exercise 7.2, check your answer using your implementation of the `toString` instance for binary trees.

ordered A binary tree is *ordered* if for each node with element x holds that all elements in the left subtree are *smaller than or equal to* x **and** that all elements in the right subtree are *greater than* x. The ordering relation of elements is determined by the `<`-instance of the element's type. Implement the function `is_ordered` that only results in `True` if a tree is *ordered*. As such (with trees $t_0 \ldots t_7$ from exercise 7.1):

```
is_ordered t0  =  True  || is_ordered t4  =  False
is_ordered t1  =  True  || is_ordered t5  =  True
is_ordered t2  =  True  || is_ordered t6  =  False
is_ordered t3  =  True  || is_ordered t7  =  False
```

balanced A binary tree is *balanced* if for each node in the tree holds that the difference in *depth* (see exercise 7.1) of its left and right subtrees is at most 1. Implement the function `is_balanced` that only results in `True` if a tree is *balanced*. As such (with trees $t_0 \ldots t_7$ from exercise 7.1):

```
is_balanced t0  =  True  || is_balanced t4  =  True
is_balanced t1  =  True  || is_balanced t5  =  False
is_balanced t2  =  True  || is_balanced t6  =  False
is_balanced t3  =  False || is_balanced t7  =  True
```

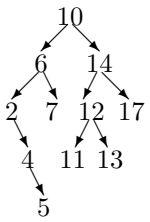
7.4 Traversing trees

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

In this exercise we use the following data type for binary search trees:

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

An example of a binary search tree is (`Node`s and `Leaf`s are not shown):



Search trees can be converted to lists in different ways. Implement the three functions `listAscending`, `listDescending`, and `listToLeaves`, all of type `(Tree a) -> [a]`. The requirements of the functions are:

- `listAscending`: returns the elements in ascending order, starting at the smallest element.

  Example: the tree above is converted to `[2,4,5,6,7,10,11,12,13,14,17]`.

  Note: do not implement `listAscending` as `reverse listDescending`.

- `listDescending`: returns the elements in descending order of magnitude, starting at the greatest element.

  Example: the tree above is converted to `[17,14,13,12,11,10,7,6,5,4,2]`.

  Note: do not implement `listDescending` as `reverse listAscending`.

- `listToLeaves`: returns the elements in order of distance to the root of the tree, starting at the root element, then all elements below (left to right), then all elements below those (left to right), etc.

  Example: the tree above is converted to `[10,6,14,2,7,12,17,4,11,13,5]`.

7.5 Map and fold over trees

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

Consider the following type for binary trees, and a number of functions:

```
::  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
```

For each of the functions below, give the most general type and describe what the functions calculate:

```
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]))
```

7.6 Generalized trees(*used in 7.7,7.8*)

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

In this exercise you work with generalized trees:

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

They are more general than the binary trees we have seen so far, because the type of elements in the `Node`s can differ from the type of elements in the `Leaf`s. Additionally, a node can have 0 or more subtrees. Implement the following functions for these trees:

```
::  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
```

with the following specifications:

- `root` returns the element of the root. The type depends on the fact whether it is a node or a leaf element. Consequently, the `Either` type is used.
- `trees` returns the direct subtrees of the tree. `Leaf` has no subtrees, and the subtrees of (`Node _ ts`) are `ts`.
- `is(Node/Leaf)Member` tests whether the first argument occurs as node/leaf element.
- `all(Nodes/Leaves)` returns all node/leaf elements. If nodes and leaves are of the same type, then `allMembers` can be used to return all elements.
- `map2 (f,g)` applies `f` on all node elements, and `g` on all leaf elements.

7.7 Displaying generalized trees(*used in 7.8*)

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

In this exercise you develop two possible algorithms to print generalized trees, in a manner analogous to that in exercise 7.2:

1. `indentTree` that prints the tree by systematically indenting nodes with the same depth with the same level of indentation; and
2. `tree2D` that prints the tree by displaying nodes with the same depth side by side, utilizing `TextCompose` from exercise 4.34.

As an example, we use the following, somewhat strangely formed, tree structure:

```
:: 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]]
```

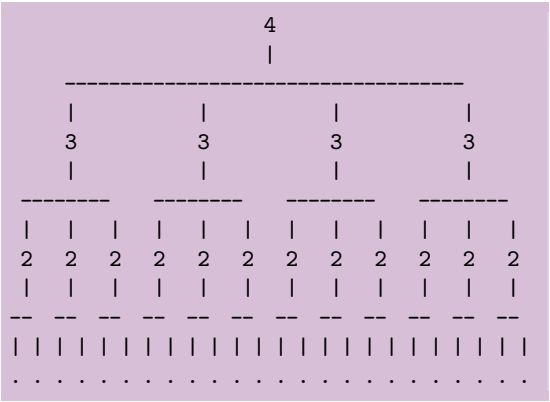
This generalized tree structure has subtrees that each have one fewer subtree. At the bottom of the tree you find leafs that do not contain any further useful information. That is represented by the `Void` type, using `.` to print it.

7.7.1 Printing with indentation

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

Develop, in analogous manner as in exercise 7.2.1, an algorithm that does the same for generalized trees. An example output of `pyramid 3` is shown on the left. The output of `pyramid 4` is not shown here: this is more suitable as alternative toilet paper decoration, try it and see for yourself.

7.7.2 Printing in 2D



Develop, in analogous manner as in exercise 7.2.2, an algorithm that does the same for generalized trees. An example output of `pyramid 4` is shown on the left. In contrast with the indentation algorithm, this results in a better distributed output.

7.8 Family trees(*used in 7.9*)

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

Family trees can be defined as a special case of generalized trees as defined in exercise 7.6:

```
:: 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
```

For convenience we assume the following, very conservative situation: people get children only after they are married, i.e. when they form a couple. People do not remarry. A person p is born as a (`Single` p) and after marrying with a person q goes through life as the couple (`Couple` p q). The person q is *related by marriage* and does not belong to the offspring of someone.

Implement the following functions for the family tree:

```
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 checks whether a family tree t satisfies the following criteria:
  - children are younger than their parents;
  - siblings are ordered from oldest to youngest;
  - no person occurs more than once in the family tree.
- `rootAncestor` t returns the family elder of tree t.
- `inFamilyTree` p t tests whether a given person p (descendant or related by marriage) occurs in the family tree t.
- `marry` p q t changes an unmarried person (`Single` p) in t into the married couple (`Couple` p q). If person p does not occur in t, or if p is already married, t is unchanged.
- `addChild` p c t adds a child p to the parents (couple) c in t. If the couple c does not occur in t, t is unchanged.
- `children` p t returns all children of a given person p in t; p can be a descendant as well as a person related by marriage. If p does not occur in t, then no people are returned.

- `offspring` p t returns all descendant of a given person p in t; p can be a descendant as well as related by marriage, but p's offspring are never related by marriage.

Determine a convenient order of defining the above functions. If you have performed exercise 7.6, it is smart to make use of the functionality implemented there as much as possible.

7.9 Displaying family trees

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

Use the printing functionality of the generalized trees from exercise 7.7 to also be able to print family trees. This quickly leads to a wide output. The output can be redirected to a text file in a straightforward manner from the command prompt:

```
> FamilyTreePrint.exe >output.txt
```

7.10 AVL trees

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

Implement the module `StdAVLTree.icl` that realizes *AVL* trees as worked out in the lecture. The AVL tree should store the depth in the tree itself, such that it does not have to be recomputed each time. The corresponding `StdAVLTree.dcl` is as follows:

```
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
```

The functions `mkAVL(Leaf/Node)` create an empty *AVL* tree / node with a given value and empty subtrees; the function `isMemberAVLTree` tests whether a given value occurs in the *AVL* tree; `insertAVLTree` adds an element to the *AVL* tree; `deleteAVLTree` removes an element from the *AVL* tree; `isAVLTree` is a predicate testing whether the given *AVL* tree is constructed correctly. This last function would not usually be incorporated in the abstract signature of correctly constructed *AVL* trees, but it is useful during the development process of the functions to test whether no errors have occurred.

Optional

If you have performed exercise 7.2, which can print binary trees, then it can be used as a convenient tool to be able to *see* whether your *AVL* trees have been constructed correctly.

7.11 Sorted files and trees(*used in 7.12*)

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

Files are often sorted. Take, for example, the file `EnglishWords.txt` from the *Exercises folder*.

Write the function `readSortedFile`, which gets as its argument the path to a sorted text file, and reads it into a *balanced binary search tree*.

Every line (ended by `'\n'`) in the text file is one element (without `'\n'`). If you have done exercise 7.10 (*AVL* trees), you may use it. Otherwise, use the fact that the file is sorted.

Test your function on the file mentioned above. You may have to increase the *Maximum Heap Size* of your project. Otherwise you may get the *runtime* error *Heap full*.

When this works, write the inverse function `writeSortedFile`, which writes a binary search tree to a sorted text file.

7.12 Interactive Boggle

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

In this exercise, you develop a one-player interactive variant of the game *Boggle*. See exercise 4.36 for a description of *Boggle*. Officially, the words have to be found in an English dictionary. For this exercise, you may use the file `EnglishWords.txt` in the *Exercises folder*. Modify the *Maximum Heap Size* of your project if you get the runtime error *Heap full*.

To be able to find words efficiently, you can best store them in a balanced binary search tree. Use the module you developed in exercise 7.11 for this.

The English version of *Boggle* uses the following dice:

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

Your console program should implement the following commands:

1. *new*: This command starts a new game for the player. The dice are shuffled using the random functions from exercise 5.18, placed in the 4×4 matrix and shown to the player.

2. A game consists of ten rounds. In every round, the player enters one word. The program checks if the word consists of at least three letters, that it can be found in the matrix, that it exists in the dictionary, and that it has not been entered before. If all checks pass, the player gets points for the word: for words with 3 or 4 letters 1 point, 5 letters 2 points, 6 letters 3 points, 7 letters 5 points and 8 or more letters 11 points.

3. If all ten rounds are finished, the score of the player is shown.

4. *stop*: this command terminates the program in a clean way.

7.13 Huffman coding

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

In this exercise, you develop a compression and decompression program that uses *Huffman coding*[1]. Huffman-coding was invented in 1952 by David Huffman[2]. Characters in a ASCII file are each represented by a code of equal length. This entails that rare characters take up as much representation space as frequent characters. The basic idea of Huffman coding is to represent frequent characters by a short code, and rare characters by a longer code. This means that the method is dependent of a *frequency table*. These are different for each language (e.g. Dutch vs. English). Therefore, you will also have to save the frequency table in the compressed data. It also means that the resulting file will be binary. When the application area is fixed, it is not necessary to store the frequency table: it is then possible to agree on a fixed table. For English, the frequency table from table 7.1 can be used. This table can also be found in `freqEN.txt` in the *Exercises folder*.

E	12.02%	R	6.02%	F	2.30%	K	0.69%
T	9.10%	H	5.92%	Y	2.11%	X	0.17%
A	8.12%	D	4.32%	W	2.09%	Q	0.11%
O	7.68%	L	3.98%	G	2.03%	J	0.10%
I	7.31%	U	2.88%	P	1.82%	Z	0.07%
N	6.95%	C	2.71%	B	1.49%		
S	6.28%	M	2.61%	V	1.11%		

Table 7.1: English letter frequency (source: https://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html)

Another consequence of using codes of different lengths, is that the code of one character cannot be the *prefix* of the code of another character. This requirement is called *prefix code* (sometimes: *prefix-free code*). The algorithm that was developed by Huffman to generate a prefix code is explained on https://en.wikipedia.org/wiki/Huffman_coding. Examine this algorithm.

In this exercise, you write a console I/O program that implements the commands below. Use the `SimpleFileIO` module from exercise 6.6 to deal with reading and writing files.

Frequency table → Huffman code table

The command *table* `_freq`*XX*`.txt` reads a frequency table from the file `freq`*XX*`.txt` (in the same format as `freqEN.txt` in the *Exercises folder*), generates a Huffman code table and writes this to the file `huffman`*XX*`.txt`. To check your implementation, verify that your output for `freqEN.txt` is equal to the file `huffmanEN.txt` in the *Exercises folder*.

Huffman code table × text file → compressed file

The command *encode* `_huffman`*XX*`.txt_`*file*`.txt` reads the Huffman code from

[1] Source: https://en.wikipedia.org/wiki/Huffman_coding

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

`huffman`XX`.txt`, compresses the text file $file$`.txt` using that code and writes the result to $file$`.huf`.

Huffman code table × compressed file → text file
The command *decode* `huffman`XX`.txt` $file$`.huf` reads the Huffman code from `huffman`XX`.txt`, decompresses the compressed file $file$`.huf` using that code and writes the result to $file$`.txt`.

7.14 Propositional logic

Main module:	*PropositionalLogic2.icl*
Environment:	*StdEnv*

7.14.1 Ground terms

Terms in propositional logic without variables (known as *ground terms*) are inductively defined as follows:

1. *true* and *false* are terms;

2. if t is a term, then $not(t)$ is also a term;

3. if t_1 and t_2 are terms, then $and(t_1, t_2)$ and $or(t_1, t_2)$ are also terms.

An example of a term that is constructed following these three rules is:

$$not(or(and(false, true), or(false, and(true, true))))$$

A boolean value can be assigned to every term t. This is done utilizing an interpretation $[\![t]\!]$. This interpretation is also defined inductively:

A. $[\![true]\!] = true$ and $[\![false]\!] = false$.

B. The value of $not(t)$ is the negation of the value of t.
As such: $[\![not(t)]\!] = \neg[\![t]\!]$.

C. The value of $and(t_1, t_2)$ is only true if t_1 and t_2 are true.
As such: $[\![and(t_1, t_2)]\!] = [\![t_1]\!] \wedge [\![t_2]\!]$.
The value of $or(t_1, t_2)$ is only false if t_1 and t_2 are false.
As such: $[\![or(t_1, t_2)]\!] = [\![t_1]\!] \vee [\![t_2]\!]$.

The term above has the following interpretation:

$$\begin{aligned}
& [\![not(or(and(false, true), or(false, and(true, true))))]\!] \\
= \; & \neg[\![or(and(false, true), or(false, and(true, true)))]\!] \\
= \; & \neg([\![and(false, true)]\!] \vee [\![or(false, and(true, true))]\!]) \\
= \; & \neg(([\![false]\!] \wedge [\![true]\!]) \vee ([\![false]\!] \vee ([\![and(true, true)]\!])) \\
= \; & \neg(([\![false]\!] \wedge [\![true]\!]) \vee ([\![false]\!] \vee ([\![true]\!] \wedge [\![true]\!]))) \\
= \; & \neg((false \wedge true) \vee (false \vee (true \wedge true)))
\end{aligned}$$

This is a "normal" *boolean* expression that can be calculated as follows:

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

Representation Utilizing algebraic types, develop a suitable representation of the propositional logic terms described above. Call this representation `PropL`.

Printing Implement an *instance* of the `toString` *type class* for the type `PropL`. This converts `PropL` values to a `String`.

Evaluation Implement a function `eval1` that interprets a `PropL` term as described above, and yields the corresponding `Bool` value.

7.14.2 Variables

We now add *variables* to the inductive definition of terms. The following rule is added:

4. the *variables* $v_i (i \in \{1, 2, 3, \ldots\})$ are terms.

An example of a term *with* variables is:

$$not(or(and(v_1, v_2), or(v_2, and(v_2, true))))$$

To interpret a term t with variables you need extra information. A *valuation* of variables is an image that assigns a single value to each variable. For the above example, $val = \{(v_1, true), (v_2, false)\}$ is a possible valuation of the variables $\{v_1, v_2\}$. We extend the interpretation $[\![t]\!]$ with this valuation: $[\![t]\!]_{val}$ and the following rule:

D. The value of v_i, given a valuation $val = \{\ldots (v_i, b_i) \ldots\}$ is b_i. This is written as $val(v_i)$. As such: $[\![v_i]\!]_{val} = val(v_i)$.

In the rules **A** up to and including **C**, the valuation is passed through unaltered.

Representation Adapt `PropL` such that variables can be represented.

Printing Adapt the `PropL` *instance* of the `toString` *type class* such that variables can also be converted to `String`s.

Evaluation Implement the function `eval2` that extends `eval` with a valuation to be able to interpret variables. Determine a suitable representation for valuations, and call this type `Valuation`.

Filtering variables Implement a function `vars` that receives a term t of type `PropL` and yields a list of all variables in t. The list should not contain duplicates. The variables of above example are v_1 and v_2.

Valuations Implement a function `vals` that yields all possible valuations for variables in a list of variables *without duplicates*.

When is it true? Finally, use the above functions to implement a function `truths` that receives a term t of type `PropL`, and yields all valuations that make t true. For the term defined above, the following valuations should be yielded: $\{(v_1, true), (v_2, false)\}$ and $\{(v_1, false), (v_2, false)\}$.

7.15 Three-valued propositional logic

Main module:	*PropositionalLogic3.icl*
Environment:	*StdEnv*

In *three-valued* propositional logic, terms are built in the same manner as in exercise 7.14, which is about *two-valued* logic. The difference is that variables can take any one of *three* values, namely *true*, *false*, and *unknown*. Calculating with the values *true*, *false*, and the logic operations is unchanged. Calculation with *unknown* is as follows:

t_1	t_2	$and(t_1, t_2)$	$or(t_1, t_2)$	$not(t_1)$
true	*true*	*true*	*true*	*false*
	false	*false*	*true*	
	unknown	*unknown*	*true*	
false	*true*	*false*	*true*	*true*
	false	*false*	*false*	
	unknown	*false*	*unknown*	
unknown	*true*	*unknown*	*true*	*unknown*
	false	*false*	*unknown*	
	unknown	*unknown*	*unknown*	

Develop data structures and functions as in exercise 7.14. However, adapt the data structures and functions such that these are suitable for both two-valued as well as three-valued propositional logic. To do this, make use of overloading.

7.16 Refactoring expressions *(used in 7.19)*

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

In this exercise you modify representations of expressions of the following form:

```
E1 = ( let x = 42 - 3 in x / 0 ) + ( let y = 6 in y * y )
                          // syntactic correct Clean expression with run-time error
E2 = let x = 42 in x + ( let x = 58 in x )
                          // syntactic correct Clean expression with result 100
E3 = let x = 1 in let y = 2 in let x = 3 in 4
                          // syntactic correct Clean expression with result 4
E4 = let x = 1 in x + y   // syntactic incorrect Clean expression (y unknown)
E5 = ( let x = 1 in x ) * x  // syntactic incorrect Clean expression (outer x unknown)
```

These expressions can be represented by utilizing the following types:

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

As such, an expression can be a number n (represented as `(NR` n`)`), a variable with the name x (represented as `(VAR` x`)`), an arithmetical operation (e_1 *op* e_2) (represented as `(OP` e_1 *op* e_2`)`, or a declaration (**let** n `=` e_1 **in** e_2) (represented as `(LET` n e_1 e_2`)`). The above expressions E_i will then be represented as `Expr` values `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")
```

7.16.1 Printing expressions

Make an *instance* of the overloaded function `toString` for `Expr`:

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

This instance should display `Expr` values as shown above. The result of `Start = map toString [E1,E2,E3,E4,E5]` is the list with subsequent strings E_1, E_2, E_3, E_4, E_5.

Note that *no* parenthesis may be placed around arguments of calculations consisting only of a number or a variable.

7.16.2 Free variables

Implement the function `free :: Expr -> [Name]` that yields all free variables present in an expression. A variable x is bound by a **let** x = ... **in** e. A variable is free if it is not bound. The result of `Start = map free [E1,E2,E3,E4,E5]` is `[[], [], [], ["y"], ["x"]]`. Note that in `E5` the variable with the name `"x"` *is* defined in the first argument of the multiplication, but not in the second argument.

7.16.3 Unused variables

In expression `E3` variables with the names `"x"` and `"y"` are defined, but not used. This expression can be simplified to `(NR 4)` while maintaining meaning. In general an expression (**let** x = e_1 **in** e_2) can be simplified to e_2 if x does not occur free in e_2. Implement the function `remove_unused_lets :: Expr -> Expr` that performs this transformation. The result of `map remove_unused_lets [E1,E2,E3,E4,E5]` is `[E1,E2,NR 4,E4,E5]`.

7.16.4 Evaluator

The value of an expression is calculated by an evaluator. The value of a number is the number itself. The value of the use of a variable is its definition. That is only possible for defined variables, and as such expressions E_4 and E_5 have no value. The value of an arithmetical operation is the arithmetical operation on the values of its arguments. We use calculations on integers. Zero division is not allowed, and as such expression E_1 has no value. The value of a variable definition is the use of its new definition in the remainder of the expression. Note that this means the value of E_2 is 10, and not 84 or 116.

The evaluation of expressions can succeed – in which case it yields an integer value – or fail (due to use of an undefined variable or division by zero) – and yield an undefined value. This is represented as follows:

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

Implement the function `eval :: Expr -> Val` calculating the value n of an expression and, if it exists, yielding that value as `(Result` n`)`. If the expression is not a valid calculation, `Undef` should be yielded. The result of `Start = map eval [E1,E2,E3,E4,E5]` is `[Undef,Result 100,Result 4,Undef,Undef]`.

7.17 A λ-reducer

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

The λ-calculus is a strongly simplified form of a functional programming language. Terms in the λ-calculus conform to the following syntax:

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

A term is a constant (*con*), a variable (*var*), an application of two terms t_1 and t_2 (t_1 t_2) or a λ-abstraction ($\lambda x_i.t$), in which the variable x_i may or may not occur in t. A variable that is introduced by a λ-abstraction is called *bound*. The occurrences of these variables in t are *not free*. Variables that are not introduced by a λ-abstraction are called *free* variables. Examples of λ-terms are:

$$\begin{array}{lcl} T_0 & \equiv & 42 \\ T_1 & \equiv & x_0 \\ T_2 & \equiv & (\lambda x_0.x_0) \\ T_3 & \equiv & ((\lambda x_0.x_0)\ 42) \\ T_4 & \equiv & ((\lambda x_0.(x_0x_0))(\lambda x_1.(x_1x_1))) \\ T_5 & \equiv & (\lambda x_0.(\lambda x_1.x_0)) \\ T_6 & \equiv & ((\lambda x_0.(\lambda x_1.x_0))\ 42)\ ((\lambda x_0.(x_0x_0))(\lambda x_1.(x_1x_1))) \end{array}$$

λ-terms can be represented in Clean utilizing an algebraic type:

```
:: Term    = C Value                  // constant v (C v)
           | X Index                  // variable x_i (X i)
           | (@.) infixl 7 Term  Term // application t1 t2 (t1 @. t2)
           |  \.           Index Term // abstraction λx_i.t (\.i t)
:: Value :== Int                      // arbitrary integer value
:: Index :== Int                      // index i (usually i ≥ 0)
```

We limit ourself to integer constants v (`C` v). We denote the variable x_i with index i as (`X` i). The application of two terms t_1 and t_2 is (t_1 `@.` t_2). The λ-abstraction $\lambda x_i.t$ is denoted as (`\.`i t). The above terms T_i are represented by the following Clean terms t_i:

```
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
```

It is important to place parentheses in the Clean representations. For example, if you define `t5` as `(\.0 \.1 (X 0))` or `(\.0 \.1 X 0)`, you will encounter the following error message from the Clean compiler:

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

7.17.1 Printing

Implement an *instance* of the `toString` *type class* for the type `Term`. This instance converts `Term` values to a `String`. Avoid displaying unnecessary parentheses around constants and variables. Display `(X` i`)` as `"x_`i`"`. The Clean terms t_i can be converted to `String`s as follows (your solutions are allowed to differ somewhat):

```
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))))"
```

Just like in Clean every (sub)expression of a λ-term of the form $((\lambda x.t_1)\ t_2)$ is a *redex* (*red*ucable *ex*pression). A λ-term *without* redex is in *normal form*.

7.17.2 Normal form

Implement the predicate `nf :: Term -> Bool` that determines whether a Clean representation of a λ-term is in normal form. Of the terms T_i, the terms T_0, T_1, T_2, and T_5 are in normal form.

Rewriting a redex $((\lambda x.t_1)\ t_2)$ proceeds, just like in Clean, by substituting all *free* occurrences of the variable x in t_1 by t_2. This is known as *uniform substitution*. We denote this as $t_1 <: (x, t_2)$. A variable x occurs free in t_1 if it is not bound by a λ-abstraction within t_1. For example, in the term $((\lambda x_0.x_0)x_0)$ only the last occurrence of x_0 is free, because the first occurrence is bound by the λ-abstraction. So, the result of $((\lambda x_0.x_0)x_0) <: (x_0, 42)$ is $((\lambda x_0.x_0)42)$.

7.17.3 Variables

Implement the function `vars :: Term -> [Index]` that yields all free and bound variables *without duplicates* in a Clean representation of a λ-term.

7.17.4 Fresh variable

Implement the function `fresh :: [Term] -> Index` that, when applied on a series of Clean representations of λ-terms $T_0 \ldots T_n$, yields the index i of a variable x_i that does *not* occur in $T_0 \ldots T_n$.

Uniform substitution $t_1 <: (x, t_2)$ replaces each *free* occurrence of x in t_1 by t_2. This function is defined as follows:

$$
\begin{array}{lll}
con & <: (x,t) = & con \\
x & <: (x,t) = & t \\
y & <: (x,t) = & y \\
(t_1\ t_2) & <: (x,t) = & ((t_1 <: (x,t))\ (t_2 <: (x,t))) \\
(\lambda x.t_1) & <: (x,t) = & (\lambda x.t_1) \\
(\lambda y.t_1) & <: (x,t) = & (\lambda y.(t_1 <: (x,t))) \qquad \textbf{if } y \notin vars(t) \\
(\lambda y.t_1) & <: (x,t) = & (\lambda z.((t_1 <: (y,z)) <: (x,t))) \qquad \textbf{with } z = fresh\{t, t_1\}
\end{array}
$$

7.17.5 Substitution

Implement an operator `(<:) infixl 6 :: Term (Index,Term) -> Term` that performs uniform substitution as described above.

A redex $((\lambda x.t_1)\ t_2)$ is *reduced* by uniform substitution of the free occurrences of x in t_1 by t_2. This is known as β-reduction ($\longrightarrow_\beta$):

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

7.17.6 Reduction

Implement the function `beta_reduce :: Term Term -> Term` that takes the Clean representations of the λ-terms $(\lambda x.t_1)$ and t_2 as arguments, and yields the Clean representation of term t such that $(\lambda x.t_1)t_2 \longrightarrow_\beta t$. If the first argument is not a λ-abstraction, `beta_reduce` should generate an error message.

A λ-term t is reduced by repeatedly applying the $\longrightarrow_\beta$ rule until the resulting term is in normal form. In the lecture, two reduction strategies were covered: *normal order* and *applicative order*. Both strategies choose the *left-most, outermost* redex in t (the left-most redex in the `Term` representation, that is highest in the data structure). *Normal order* reduction rewrites the redex, while *applicate order* reduction first rewrites the argument until it is in normal form, and only then rewrites the resulting redex.

7.17.7 Strategy

Implement the functions `normal_order :: Term -> Term` and `applicative_order :: Term -> Term` that search for the left-most, outermost redex in a term t and rewrite it according to the normal order and applicate order evaluation strategies. Rewriting of terms itself is handled by the `beta_reduce` function.
Example:

```
                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)
```

7.17.8 Rewriting to normal form

Implement the higher-order function `rewrite :: (Term -> Term) Term -> Term` that receives as argument a strategy function f as implemented in 7.17.7 and a term t. The term t will be rewritten using f until it reaches normal form.
Example:

```
                      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
```

Example:

```
                      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
```

7.18 `map` and type constructor classes

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

Study the following data structures and functions:

```
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)
```

Develop a *type constructor class* with the name `Map` (`map` is already in use) inside the `Map.dcl` and `Map.icl` modules such that the functions `map`, `mapMaybe` and `mapTree` are the implementations of instances of the type constructor class `Map` of the respective type constructors `[]`, `Maybe`, and `Tree`. The main module `Mapping.icl` contains a call for each of these instances.

7.19 RefactorXX, monadic

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

In exercise 7.16.4 you implemented an evaluator for *let expressions* that are represented according to the following recursive data structure:

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

7.19.1 Monadic evaluator

In this exercise you transform your evaluator to a *monadic* variant, namely the `MonadFail` type constructor class:

```
eval :: Expr -> c Int | MonadFail c
eval ...
```

For completeness the definitions of the `MonadFail` type constructor class are added in the `RefactorXX.icl` module:

```
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 your monadic evaluator for the *list* instances of the `MonadFail` type constructor class.

7.19.2 Monadic values

Create an instance of each of the `MonadFail` type constructor classes for the `Val` type. I.e., complete the following definitions:

```
::  Val a = Result a | Undef

instance fail   Val where ...
instance return Val where ...
instance >>=    Val where ...
```

Test your monadic evaluator for the `Val` instances of the `MonadFail` type constructor class.

7.20 Connect Four

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

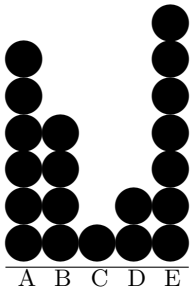
In the children's game *Connect Four* two players play against each other. They use an upright playing board consisting of a grid of m columns ($m \geq 4$, often 7) and n rows ($n \geq 4$, often 6 – 8). The players should alternate dropping discs of their own colour into a column. The first player to succeed placing 4 of their discs uninterrupted *next to each other*, *above each other* or *diagonally* wins.

Write a program that implements *connect four*. The user is able to choose the dimensions of the playing field, and can choose whether they can make the first move; the program should implement an opponent. Implement the logic of the opponent utilizing a *game tree*: i.e., develop a `GameState` data type and `moves` and `worth` functions.

7.21 Nim

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

In the game *nim* two players play against each other. They begin with a series of stacks that each consist of an arbitrary number of objects (e.g., 6-4-1-2-7 in the figure on the right).The players alternate taking *at least* one object from a single stack (i.e., they can take at most the number of objects on that stack). The player that removes the last object is the winner.

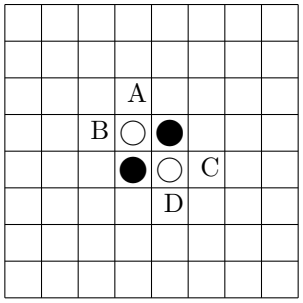


Write a program that implements *nim*. The user can choose whether they make the first move; the program should implement an opponent. Implement the logic of the opponent utilizing a *game tree*: i.e., develop a `GameState` data type and `moves` and `worth` functions.

7.22 Othello

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

The two-player game *Othello* (also known as *Reversi*) is played on a board consisting of 8×8 grid cells. In a cell, a disc can be placed that is *black* on one side, and *white* on the other. Each player plays a game with the same colour. The starting configuration is found in the figure on the side.



At the start of the game, the colours are assigned to the players. The players alternate placing one disc of their own colour on the board. They should ensure that in doing this they enclose at least *one* row (horizontal, vertical or diagonal) of their opponent: i.e., on both sides of the row of stones of the opponent there now is at least *one* stone of the player making the move. In the starting configuration, the player with the black discs can enclose the white discs by playing on the fields marked by an A, B, C or D.

The result is that all newly enclosed discs of the opponent change colour, and now belong to the player that made the move. Placing a disc can cause multiple rows (horizontal, vertical, diagonal) to become enclosed, and all these discs are conquered. If a player cannot place a disc with which at least one disc is conquered, their turn is

over and the opponent may play. The game is finished once the board consists of discs that are all the same colour, or if the board is full. Once the game is finished, the player with the most discs of their colour on the board is the winner.

Write a program that implements *Othello*. The user can choose whether they make the first move; the program should implement an opponent. The user always plays with black discs. Implement the logic of the opponent utilizing a *game tree*: i.e., develop a `GameState` data type and `moves` and `worth` functions.

7.23 Blokus

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

Blokus is a strategy game that can be played by two and four players. In this exercise we will assume there are two players. See http://www.blokus.com for more information about blokus.

The blokus game consists of a playing field of 14×14 grid cells, of which two are marked, and each player has an identical collection of 21 blocks in their own colour (see Figure 7.1).

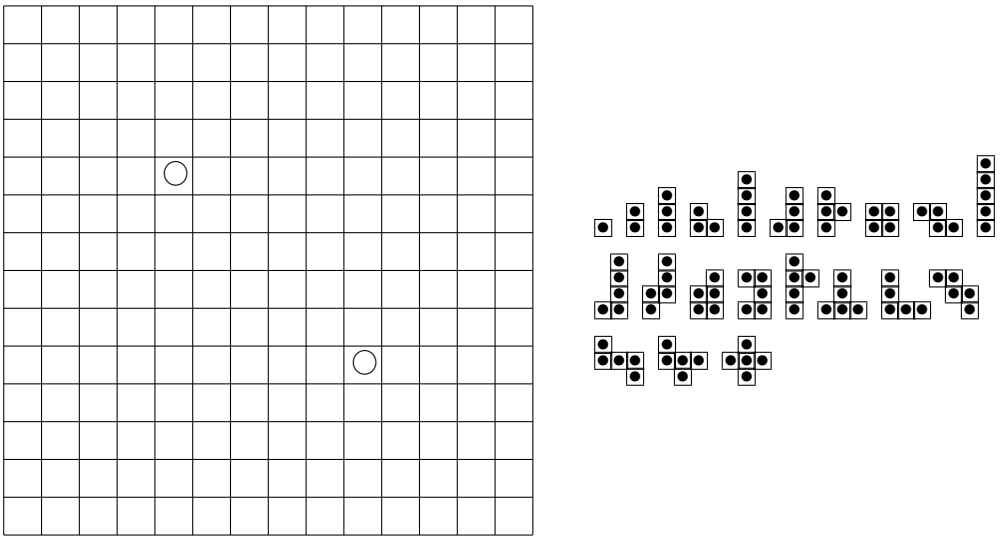


Figure 7.1: The blokus playing field and the 21 blokus playing blocks

The players alternate place one of the playing blocks on the field. The first block of each player should be placed on the marked cell on the board. The next block of a certain colour should touch a block that has already been placed on the field. The blocks may only teach on the *corners*, and *not* on the faces. The block to be placed may touch faces of blocks of another colour. If a player cannot place a block, their turn is over. The game ends if no players can place a block, or if all blocks have been placed.

The scoring is determined by the blocks that could not be placed on the field. Each block counts as -1. If a player placed all their blocks on the board, they earn $+15$ points. If a player placed the smallest block last, they earn $+5$ points. The player with the highest score wins.

Write a program that implements blokus. The user can choose whether they make the first move; the program should implement an opponent. Implement the logic of the opponent utilizing a *game tree*: i.e., develop a `GameState` data type and `moves` and `worth` functions.

7.24 Abalone

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

Abalone (see `https://en.wikipedia.org/wiki/Abalone_(board_game)`) is a strategic board game for two players. The six-sided board consists of 61 holes in which white and black marbles can be placed. Figure 7.2**a.**[3] shows the board and the standard starting configuration.

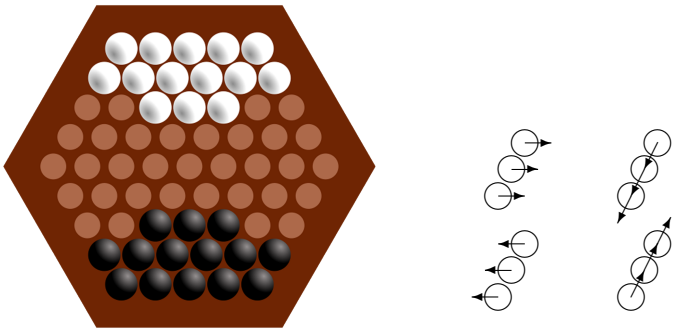


Figure 7.2: **a.** The starting configuration. **b.** The movement directions.

The player with the black marbles starts. The players alternate their turns. A turn consists of moving one, two or three marbles of the same colour connected in a single direction with one step in the same direction. I.e., this can be forwards or backwards (in the length direction) or sidewards. See Figure 7.2**b**. Marbles of one colour may push marbles of the other colour, but not of the same colour (otherwise you would move too many of your own marbles). In this case, the pushing marbles should be in the majority: to push one marble of your opponent, you need two or three of your own marbles, and pushing two marbles of your opponent is only possible with three of your own marbles. As a consequence, you can never push marbles of your opponent when moving sidewards.

The goal of the game is to be the first to push six marbles of your opponent off the board.

Write a program that implements Abalone. The user can choose whether they make the first move; the program should implement an opponent. Implement the logic of the opponent utilizing a *game tree*: i.e., develop a `GameState` data type and `moves` and `worth` functions.

[3] Source: `http://nl.wikipedia.org/wiki/Bestand:Abalone_standard.svg`

Chapter 8

Correctness proofs

In these exercises you prove a number of propositions correct. These propositions use function definitions. Every alternative in a function definition is numbered. In the proofs, use these numbers to indicate which alternative you applied and underline that part of the expression which you applied it to. If you use the equivalence from right to left, indicate that using a $\Leftarrow$. **Do not skip any steps.**

First, complete the proof on paper. You will certainly need to provide one or two proofs at the exam, so doing proofs on paper makes for good practice.

When you completed the proof, you can enter it as ASCII text or with a word processor of your preference. In the latter case, you must upload your solution as a *pdf* document. For example, let's say you need to prove for all finite lists `xs` that: `xs ++ [] = xs` by the given definition of `++`:

```
(++) :: [a] [a] -> [a]
(++) []       ys = ys              (1)
(++) [x : xs] ys = [x : xs ++ ys]  (2)
```

Then your proof **must** look as follows (otherwise it is not graded!):

```
Prove: for all xs :: [a]: xs ++ [] = xs
Proof:
    By induction on xs

    Base case:
    assume xs = []

    Prove:  xs ++ [] = xs
    Proof:
        (assumption) xs ++ [] = xs  (assumption)
                     **         **
        (1)      <=> [] ++ [] = []
                     ********
                 <=> []       = []  .

    Induction case:
    assume property holds for certain xs:
        xs ++ [] = xs        (IH)
    Prove:   [x : xs] ++ [] = [x : xs] for all x
```

```
Proof:
    (2)       [x : xs] ++ []   = [x : xs]
              **************
    (IH)  <=> [x : xs ++ []] = [x : xs]
                    ********
          <=> [x : xs]       = [x : xs]  .

Base + Induction: proof complete.
```

8.1 map and o

Main module:	*ProofMapO.icl*
Environment:	—

Use the following function definitions:

```
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)
```

Prove the following proposition for all *finite* lists xs and functions f and g:

```
map (f o g) xs = map f (map g xs)
```

8.2 init and take

Main module:	*ProofInitTake.icl*
Environment:	—

Use the following function definitions:

```
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)
```

Prove the following proposition for all *finite, non-empty* lists xs:

```
init xs = take (length xs - 1) xs
```

For convenience, assume that the *Int*eger range is unbounded.

8.3 Peano arithmetic

Main module:	*ProofPeano.icl*
Environment:	—

We are able to represent natural numbers as follows (the so-called *Peano* arithmetic):

```
:: Nat = Zero | Suc Nat
```

In other words, 0 is a natural number (`Zero`), and if x is a natural number, then the successor of x (`Suc` x) is also a natural number. The function that shows the relation between `Nat` and `Int` is the following:

```
( ## ) :: Nat -> Int
( ## ) Zero    = 0            (1)
( ## ) (Suc n) = 1 + ##n      (2)
```

Assume that the *Int*eger range is unbounded.

Addition Given the function `add`:

```
add :: Nat Nat -> Nat
add Zero    n = n             (3)
add (Suc m) n = Suc (add m n) (4)
```

Prove the following proposition for all m and n:

```
##(add m n) = ##m + ##n
```

Multiplication Given the function `mul`:

```
mul :: Nat Nat -> Nat
mul m Zero    = Zero              (5)
mul m (Suc n) = add (mul m n) m   (6)
```

Prove the following proposition for all m and n:

```
##(mul m n) = ##m * ##n
```

8.4 `map`, `flatten` and ++

Main module:	*ProofMapFlatten.icl*
Environment:	—

Consider the two following function definitions:

```
(++) :: [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)
```

8.4.1 `map` and `++`(used in 8.5)

Prove the following proposition for all *finite* lists as and bs and functions f (carefully consider on what list you apply the induction!).

```
map f (as ++ bs) = (map f as) ++ (map f bs)      (8.4.1)
```

8.4.2 `map` and `flatten`

Consider the following function definition:

```
flatten :: [[a]] -> [a]
flatten [ ]    = [ ]                    (5)
flatten [x:xs] = x ++ (flatten xs)      (6)
```

Prove the following proposition using complete induction on the length of list xs. Assume that xs is a finite list:

```
flatten (map (map f) xs) = map f (flatten xs)
```

Make use of property 8.4.1. **Note:** if you haven't proven 8.4.1 you may assume it holds.

8.5 Lists and trees

Main module:	*ProofMapsAndTips.icl*
Environment:	—

In this exercise we use the following type and function definitions:

```
::  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)
```

Explain what the functinos `foldbtree` and `tips` do. Prove the following proposition for any function f and any finite binary tree t:

```
map f (tips t) = tips (mapbtree f t)
```

Use lemma 8.4.1.

8.6 subs and map

Main module:	*ProofSubsAndMap.icl*
Environment:	—

Given the following function definitions:

```
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)
```

Explain what `subs` computes and give the result of the expression (`subs ['abcd']`). Prove the following proposition for all functions f and finite lists xs:

`subs` (`map` f xs) = `map` (`map` f) (`subs` xs).

You may use the following lemmas that hold for all functions f, g and finite lists xs and 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)

Chapter 9

Dynamics

Dynamics enable you to wrap calculations and pass them on. During wrapping, not only the calculation is stored in some way, but a representation of its *type* is also stored, i.e.: a dynamic value is an encapsulation of an object together with its type. An encapsulated calculation can be passed to other functions as per usual, but it can also be written to a file to be read at a later point by the same or another application. The application unpacking such a dynamic should indicate which type is expected. Only if the expected type can be made to conform with the actual type the application can use the concrete content. These checks take place during the execution of the program.

9.1 Notations

Main module:	*NotationDynamics.icl*
Environment:	*Experimental*

A number of functions are given below. Derive the most general type of each of the functions. Explain what each function does.

```
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

f6 x (y :: a^)                   = x == y
f6 _ _                           = False

f7 x                             = dynamic (x, toString x)

f8 ((x,eq_x) :: (a, a -> Bool)) ((y,_) :: (a, b))
                                 = eq_x y
```

9.2 Guessing numbers

Main module:	*GuessingNumbers.icl*
Environment:	*Experimental*

In this exercise you develop a console-I/O program that reads a dynamic file (with a nondescript file name, such as A, B, C, etc., to be selected by the user). Suppose that the user chooses the file with name *filename* (in the left image in Figure 9.1 you can see the user selected dynamic file `"A.dyn"`). The goal is for the user to guess, in as few steps as possible, which number sequence is stored in the dynamic. A session is shown in the right image in Figure 9.1. If the user inputs *enter*, the program reacts by showing the next number in the sequence. If the user inputs a number that does not correspond with the next number, they do not see which number it should have been (in the session shown, this occurs at the fourth number). If the number is correct, this is confirmed by displaying it. If the user has correctly predicted the number a few times (5 times in the session shown), then they have discovered the sequence.

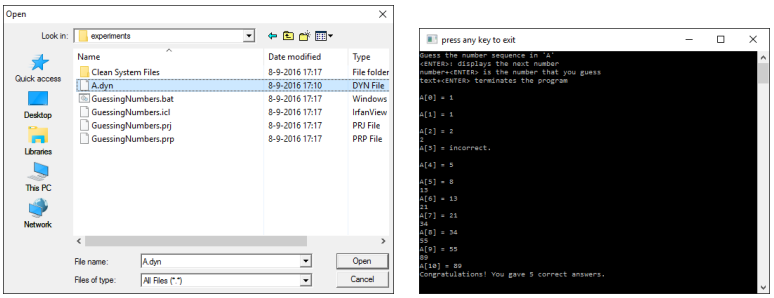


Figure 9.1: The number guessing program in action.

9.3 Heterogeneous sets

Main module:	*StdDynSet.icl*
Environment:	*Experimental*

In exercise 4.25 you are asked to develop a module for sets of elements with *identical* type (*homogeneous sets*). In this exercise we go a step further, and use *dynamics* to create sets of (potentially) *different* types (*heterogeneous sets*). The elements that can be present in these sets should be comparable; as such, they should support the overloaded `==`. To print sets, an overloaded `toString` instance is necessary (not essential, but convenient for testing). We abbreviate this as follows:

```
class Set a | TC, ==, toString a
```

Complete the implementation that corresponds with `StdDynSet.dcl`. The following properties should hold for operations on heterogeneous sets:

- The `zero` instance gives the *empty set*.
- The `toString` instance displays all elements of the set between `"{"` and `"}"`, and separated by `,`.

- The `==` instance only yields `True` if two sets contain exactly the same elements.
- `nrOfElts` yields the number of elements that are in the set; `isEmptySet` only yields `True` for the *empty set* (the empty set has zero elements).
- `memberOfSet` determines whether the first argument is an element in the set; `isSubset` determines whether the first argument is a subset of the second argument (each element in the first argument is also present in the second argument); `isStrictSubset` determines whether the first argument is a *strict* subset of the second argument (each element in the first argument is also present in the second argument, but the second argument contains elements that are not present in the first argument);
- `union` calculates the union of the two sets; `intersection` calculations the intersection of the two sets; `without` removes all elements present in the second set from the first set.

`Set` is an instance of `==` and `toString`, and thus also of class `Set`. This module enables you to create sets of sets.

9.4 An IKS interpreter

Main module:	*IKS.icl*
Environment:	*Experimental*

In this exercise you use dynamics to create an *interpreter* for an extremely simple programming language: X. X is a so-called *combinator* language. A X program is an expression, consisting of:

1. A symbol `I`, `K` or `S`.
2. The integers.
3. If e is an X expression, then `(`e`)` is also an X expression.
4. If e_1 and e_2 are X expression, then $e_1\ e_2$ is also an X expression.

The following are all valid X expressions: `I`, `K`, `I(42)`, `K(42)(100)`, `(I)`, `(K)`, `(S)`, `SI`, `(SI)`, `SKK(100)`, etc. To prevent this exercise from being needlessly complicated, you may assume that expressions do not contain *white-space* characters. In the above examples, this is indicated by separating each individual number argument of an X expression with parentheses. The semantics of an X expression are given by the following interpretation $[\![\cdot]\!]$:

- $[\![I]\!] \stackrel{def}{=} \lambda x.x$; $[\![K]\!] \stackrel{def}{=} \lambda xy.x$; $[\![S]\!] \stackrel{def}{=} \lambda xyz.xz(yz)$.
- $[\![n]\!] \stackrel{def}{=} n$, if n is an integer.
- $[\![(e)]\!] \stackrel{def}{=} [\![e]\!]$.
- $[\![e_1\ e_2]\!] \stackrel{def}{=} ([\![e_1]\!]\ [\![e_2]\!])$.

Although this language looks extremely simple, it is as powerful as the pure λ-calculus.

Create using dynamics an interpreter for X expressions. To be able to work with dynamics, you should set the *environment* to *Experimental* after clearing a Clean project. Additionally, you should check the flag *Enable dynamics* in *Project Options*.

Dynamics

Write a program that yields three file-dynamics in the same directory as the application:

1. A file with name "I" and content **dynamic** $[\![I]\!]$.
2. A file with name "K" and content **dynamic** $[\![K]\!]$.
3. A file with name "S" and content **dynamic** $[\![S]\!]$.

To this end, use the function `writeDynamic`. Ensure you give the correct type for each of the dynamics, including $\forall$ quantification. E.g.:

```
Start :: *World -> *World
Start world
    # (ok,world) = writeDynamic "I" (dynamic i :: A. a: a -> a) world
    ...
    = world

i :: a -> a
i x = x
...
```

After execution, the files "I.dyn", "K.dyn", and "S.dyn" are added to the directory.

Parsing

The next step is *parsing* a line of input. Introduce the following algebraic data type `IKS`:

```
:: IKS = I | K | S | N Int | App IKS IKS
```

This type represents the *syntax tree* of the expression that was input. The symbols `'I'`, `'K'`, and `'S'` correspond with the alternatives `I`, `K`, and `S`. The integers correspond with the alternative `N`. Note that while parsing, X expressions, like λ expressions, are left-associative. For example, this means that the expressions `SKI`, `(SK)I`, and `((S)K)I` have the same meaning.

Some hints for this part:

- Convert each `String` that is to be parsed to a `[Char]`. To do this, use the overloaded function `fromString`. Do not forget to eliminate the closing *newline* character!
- Write the function `pIKS :: [Char] -> IKS` that parses one line of input. If you want to handle erroneous input, you can define the type of this function as `pIKS :: [Char] -> Maybe IKS`. This is optional: you are allowed to assume only valid input is given.
- Due to the left-associative nature of X expressions it is most convenient to parse input from back to front, instead of front to back. Do not decompose the input using `hd` and `tl` (or `[x:xs]` pattern matches), but use functions `init` and `last`.
- Because X expressions contain parentheses, it is useful to define a helper function to extract a group of balanced parentheses from an expression:

  ```
  split_bracket :: [Char] -> ([Char],[Char])
  ```

For example:

```
split_bracket ['(SK)I']   ⇒ (['(SK)'],['I']).
split_bracket ['((S)K)I'] ⇒ (['((S)K)'],['I']).
```

- A list of digits can straightforwardly be converted to an integer:

```
number :: [Char] -> Int
number chars = toInt (toString chars)
```

Interpreting

A `IKS` syntax tree can be interpreted according to the $[\![\cdot]\!]$ function described in the introduction of this exercise. Write an interpreter function `interp` that receives as arguments dynamics corresponding with the combinators `I`, `K`, and `S`; and the `IKS` syntax tree of the parsed expression. The result of `interp` is a dynamic corresponding to the interpreted function. Thus, its type is:

```
interp :: (Dynamic,Dynamic,Dynamic) IKS -> Dynamic
```

Use the function `dynApply` for application of symbols ($[\![e_1\ e_2]\!] \stackrel{def}{=} ([\![e_1]\!]\ [\![e_2]\!])$):

```
dynApply :: Dynamic Dynamic -> Dynamic
dynApply (f :: a -> b) (x :: a) = dynamic f x :: b
dynApply     _             _    = dynamic "dynamic␣type␣error"
```

Console

In this part you combine the parts above into an I/O console program. The user can input X expressions in the console. The program prompts with the interpretation of the X expression that was input. To this end, the program first reads the three dynamics that were written to disk in 9.4. To read these dynamics, use the function `readDynamic`. E.g.:

```
# (ok,dyn_I,world) = readDynamic "I" world
```

should successfully read the dynamic that was written in 9.4.

The prompt should distinguish between a function result, an integer result, and a string result. An example of a session is given in Figure 9.2.

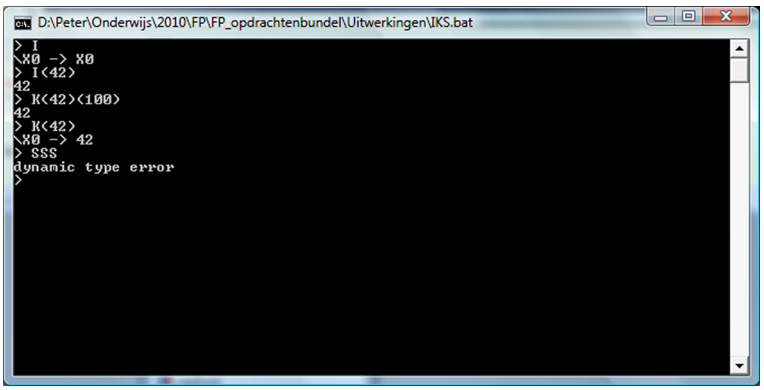


Figure 9.2: The IKS interpreter in action.

Chapter 10

SoccerFun

The SoccerFun environment is needed to complete the exercises in this chapter.

10.1 Training: walking circles

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implement a soccer player walking clockwise circles around the field. The player should not walk further than 5 meters from the edge of the soccer field. If they are not at the edge of the soccer field, the player should first walk perpendicular to the nearest edge of the soccer field (e.g., if they are 'most' north, they should walk straight to the north, or if they are 'most' west, they should walk straight to the west, etc.).

10.2 Training: slaloming

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implement a soccer player who slaloms around a number of opponents. If the player starts on the `West` half, they should run to `East`. If the player starts on the `East` half, they should run to `West`. On the other side of the soccer field the ball is waiting to be kicked into the nearest goal. Use the *RefereeCoach_Slalom* and *Opp_Slalom* to test your *Student Slalom*. This last one is implemented in module `Team_Student_Slalom_Assignment`.

10.3 Training: passing

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implement a soccer brain for players who pass the ball to each other from `West` to `East` and kick the ball into the goal (or in reverse from `East` to `West`). Use the *RefereeCoach Passing* and *Opp_Passing* to test your *Student Passing*. This last one is implemented in module `Team_Student_Passing_Assignment`.

10.4 Training: deep passing

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implement soccer brains for two players standing on opposite sides of a group of opponents. The first player should kick the ball over the ground to the other player without granting the opponents the opportunity to take possession of the ball. This is a matter of timing. Use the *RefereeCoach DeepPass* and *Opp_Deep_Pass* to test your *Student_Deep_Pass*. The last one is implemented in module `Team_Student_DeepPass_Assignment`.

10.5 Training: goalkeeper

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

Implement a brain for a goalkeeper defending their goal. The goalkeeper is surrounded by opponents passing the ball to each other. The goalkeeper should ensure the ball can not be played to the centre of the goal without the goalkeeper standing in the way. The goalkeeper should stand *in front of* the goal line. Use *RefereeCoach Keeper* and *Opp_Keeper* to test your *Student Keeper*. This last one is implemented in module `Team_Student_Keeper_`Clean`Assignment`.

10.6 Final assignment

Main module:	*SoccerFun.icl*
Environment:	*SoccerFun*

In this final exercise you develop a complete soccer team consisting of one goalkeeper and ten field players. Create a Clean module with your own name (e.g. `PeterAchten.icl`) and implement and export the team function:

```
implementation module PeterAchten

import Footballer

TeamPeterAchten :: !Home !FootballField -> Team
TeamPeterAchten home field = ...
```

The corresponding *definition module* should look as follows:

```
definition module PeterAchten

import Footballer

TeamPeterAchten :: !Home !FootballField -> Team
```

as has already been done in the framework for `TeamMiniEffie`. You may choose the starting configuration yourself. The argument of type `Home` indicates which field half you are assigned (`West` or `East`). You make your team accessible by adding the following lines to the frame in module `Team.icl` (here in correspondence with module `PeterAchten.icl`):

```
implementation module Team
```

```
...
import PeterAchten                          // do not forget to import your module
...
allAvailableTeams = [ Team_MiniEffies
                    , TeamPeterAchten       // this makes your team known
                    ]
```

That is all. Your team is now selectable in the framework.

You may decide yourself whether you implement an individual brain function for each soccer player, or that you implement a function for categories such as the goalkeeper, defenders, and attackers, or that you write a universal brain. The soccer players *must* satisfy the following rules:

Goalkeeper: This player is not allowed to leave the *penalty area*. If the ball is reasonably within reach, the goalkeeper should intercept it. "Reasonably within reach" means that the goalkeeper should take into account her distance to the ball and its speed, and the distance and speed of other players to the ball. If the goalkeeper is able to reach the ball before an opponent, the player is obligated to play the ball. This depends on the strategy of the goalkeeper's team players: the goalkeeper is not obligated to get in the way of team players.

Fielder: The field players may not all chase the ball. Instead, they should assume a reasonable field division. "Reasonable field division" means that all fielders strive to play a certain area of the field, and that the shared overlap between these areas is small. Depending on the game situation (e.g. attacking and defending) these positions should be assumed. Fielders may not possess the ball continuously; they are required to play the ball to other players if this is reasonably sensible. "Reasonably sensible" means that when a team player is in a sufficiently better area than the player themselves, and if that team player is able to receive the ball, the ball should be passed.

Rules of the game: Soccer players should respect decisions of the referee. This means that your team is *not* allowed to play the ball or take possession of it if the opposing team has right to it (e.g. during a throw-in, goal kick, corner, etc.). This also means that a team is *required* to play the ball or take possession of it if the referee indicates as such (similar situations).

Efficiency: For all soccer players it should hold that the brain function is sufficiently efficient, meaning that if the team you created were to play against itself, the calculation of the soccer actions of all 22 player together should not take more time than one twentieth of a second. This can be checked by means of the *frame-rate indicator*: the number behind the text `Rounds/sec:`. This number, during normal speed, may not drop below 20 (unless a referee dialogue interrupts te game). Note that you can set the game speed from the menu.